HELLENIC REPUBLIC

National and Kapodistrian University of Athens

Department of Computer Science

A dissertation submitted for the degree of Doctor of Philosophy

Decentralized Blockchain Interoperability

Dionysis Zindros

Athens

April 2020

HELLENIC REPUBLIC

National and Kapodistrian University of Athens

Department of Computer Science

A dissertation submitted for the degree of Doctor of Philosophy

Decentralized Blockchain Interoperability

Dionysis Zindros

Athens

Defended on: April 3, 2020
Last revision: May 23, 2021

PhD Thesis

Decentralized Blockchain Interoperability

Dionysis S. Zindros

Examination Date
April 3rd, 2020

For my parents, Efi and Spyros, and my sister Natalia.

Abstract

We put forth the first decentralized communication mechanism between proof-of-work and proof-of-stake blockchains, or combinations thereof. To construct it, we propose two new cryptographic primitives which function as cross-chain certificates. For proof-of-stake sources, the ATMS primitive (Ad-Hoc Threshold Multisignatures) allows attesting to the shifting of stake from epoch to epoch. For proof-of-work sources, the NIPoPoWs primitive (Non-Interactive Proofs of Proof-of-Work) allows compressing proof-of-work into succinct strings that shrink a long blockchain into a succinct polylogarithmic proof. We provide the first ATMS and NIPoPoWs constructions. For work, we prove our constructions are secure in both the static and the variable difficulty setting and we achieve security in the synchronous and bounded delay settings with concrete adversary bounds in each case. We put forth the first definition of sidechain security and formally prove our constructions secure. Our proofs are in the Backbone model for work and in the Ouroboros model for stake.

Our cross-chain certificates allow the transmission of generic information between blockchains. We describe multiple applications of our sidechains, including proof-of-burn-based one-way pegs and two-way pegs. In addition to interoperability, our protocols enable the transmission of blockchain information for its own internal use. This allows the construction of *superlight* clients with exponentially smaller communication complexity than traditional clients. These superlight clients are the first asymptotic improvement upon SPV. Additionally, our protocols can be used in the work setting to create *superlight miners*, which need only logarithmic state to mine and are an exponential improvement over standard proof-of-work blockchain protocols. We demonstrate the feasibility of our schemes with experiments, simulations, and implementations, including measurements of security and performance metrics. We give concrete proposals for deployment security parameters. We have worked with the industry to implement our schemes in practice: Our protocols have been implemented and deployed in the real world proof-of-work blockchains ERGO, nimiq, WebDollar, and Midnight and the proof-of-stake blockchain Cardano.

Contents

Preface

Acknowledgements

My parents, Efi and Spyros, and my sister Natalia have always inspired me and been patient and loving. Thank you.

I could not have completed this thesis without the continuous exchange of ideas with my computer scientist colleagues who read my papers and this thesis and provided feedback: My co-authors Andrew Miller, Peter Gaži, Kostis Karantias, Orfeas Stefanos Thyfronitis Litos, Dimitris Karakostas, Nikos Leonardos, Zeta Avarikioti, Roman Brunner, Christos Nasikas, Alexei Zamyatin, Roger Wattenhofer, Lefteris Kokoris-Kogias; my University of Athens colleagues, Katerina Samari, Pyrros Chaidos, Giorgos Panagiotakos; my colleagues Angel Leon, Sam Patterson, Brian Hoffman, Washington Sanchez; my mentors, Andrew McCann who taught me how to code, Andreas Moser who taught me to follow my heart, Dimitris Fotakis who taught me to count, Nikos Papaspyrou who taught me how to think, Aristidis Arageorgis who taught me how to speak, and Aris Pagourtzis, who supervised my master thesis; the people whom I've taught and been taught by and who provided a lot of helpful feedback for this thesis, Aleksis Brezas, Petros Angelatos, Themis Papameletiou, Dimitris Lamprinos, Vitalis Salis, Petros Markopoulos, Eva Sarafianou, Dimitris Grigoriou, Christos Porios, Andrianna Polydouri, Nikolas Kamarinakis, Giorgos Christoglou, and all my other teachers and students; and the people who enabled this work but were not in the spotlight, in particular Tatiana Kyrou and Mirjam Wester.

Last and most importantly, I want to thank my advisor Aggelos Kiayias for giving me the space and opportunity to learn on my own. But also closely and patiently mentoring me and teaching me modern cryptography, the importance of precise definitions, assumptions and proofs, *how to prove and how to refute*. Above all, for teaching me about the scientific method, the philosophy of science, the love for what we do, *the moral character of cryptographic work*, and for helping me navigate the labyrinthian and always increasing difficulty of completing a thesis. I am greatly honoured to be his student in the *paradigm shift* he is leading in the field of blockchain science, political cryptography, and decentralized financial cryptography.

I am indebted to IOHK who generously funded my salary and my numerous conference participations during the years of my PhD. Charles Hoskinson personally ensured I would never have to worry about funding during my doctoral work. I am also thankful OB1 and FrostWire for funding my work. Additional funding was provided by the European Union ERC project CODAMODA as well as H2020 project PRIVILEDGE #780477.

Publications

This thesis is based on the following papers.

1. *Non-Interactive Proofs of Proof-of-Work*
   Aggelos Kiayias, Andrew Miller, Dionysis Zindros[1]
   Financial Cryptography and Data Security 2020 (FC '20)

2. *Proof-of-Burn*
   Kostis Karantias, Aggelos Kiayias, Dionysis Zindros[1]
   Financial Cryptography and Data Security 2020 (FC '20)

3. *Proof-of-Stake Sidechains*
   Peter Gaži, Aggelos Kiayias, Dionysis Zindros[1]
   IEEE Symposium on Security and Privacy 2019 (Oakland '19)

4. *Proof-of-Work Sidechains*
   Aggelos Kiayias, Dionysis Zindros[1]
   Financial Cryptography Workshop on Trusted Smart Contracts 2019 (FC WTSC '19)

5. *Variable Difficulty Proofs of Proof-of-Work*
   Aggelos Kiayias, Nikos Leonardos, Dionysis Zindros[1]
   Unpublished Manuscript

6. *Mining in Logarithmic Space*
   Aggelos Kiayias, Nikos Leonardos, Dionysis Zindros[1]
   Unpublished Manuscript

7. *Smart Contract Derivatives*
   Kostis Karantias, Aggelos Kiayias, Dionysis Zindros[1]
   International Conference on Mathematical Research for Blockchain Economy 2020 (MARBLE '20)

The following works, which do not form an integral part of this thesis, were also completed during its preparation.

1. *Cryptocurrency Egalitarianism: A Quantitative Approach*
   Dimitris Karakostas, Aggelos Kiayias, Christos Nasikas, Dionysis Zindros[1]
   International Conference on Blockchain Economics, Security and Protocols 2019 (TOKENOMICS '19)

2. *Compact Storage of Superblocks for NIPoPoW Applications*
   Kostis Karantias, Aggelos Kiayias, Dionysis Zindros[1]
   International Conference on Mathematical Research for Blockchain Economy 2019 (MARBLE '19)
   Nominated for Best Paper Award

3. *Trust Is Risk: A Decentralized Financial Trust Platform*
   Orfeas Stefanos Thyfronitis Litos, Dionysis Zindros[1]
   Financial Cryptography and Data Security 2017 (FC '17)

[1]Authors are ordered alphabetically.

4. *Productizing TLS Attacks: The Rupture API*
   Aggelos Kiayias, Eva Sarafianou, Dionysis Zindros[1]
   Real World Crypto Symposium 2017 (RWC '17)

5. *CTX: Eliminating BREACH with Context Hiding*
   Dimitris Karakostas, Aggelos Kiayias, Eva Sarafianou, Dionysis Zindros[1]
   Black Hat Europe 2016

6. *SoK: Communication Across Distributed Ledgers*
   Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, William J Knottenbelt

7. *Structure and Content of the Visible Darknet*
   Georgia Avarikioti, Roman Brunner, Aggelos Kiayias, Roger Wattenhofer, Dionysis Zindros[1]

Structure

This thesis is structured as follows. Chapter 1 gives an introduction to the problem of Blockchain Interoperability which we treat in this work and places our work in the broader context of the related scientific literature. Chapter 2 introduces the reader to the prerequisites required to understand the work, including the cryptographic primitives we use, the Bitcoin Backbone and Ouroboros models in which we work, and the technical details of existing cryptocurrency implementations which we build upon, with a focus on Bitcoin and Ethereum. A reader familiar with these concepts can skip this chapter. We then devote the next three chapters to discuss how to disemminate information from open blockchains of both consensus mechanisms, which form the heart of this thesis. Chapter 3 puts forth our Proofs of Proof-of-Work construction, which allows consuming information from proof-of-work-based blockchains. In Chapter 4, we use them to build superlight clients, including SPV nodes, full nodes, and superlight miners which can replace full miners while maintaining only logarithmic consensus state. Chapter 5 extends our results in the Variable Difficulty and Bounded Delay model. Chapter 6 puts forth our Proofs of Proof-of-Stake construction, which allows consuming information from proof-of-stake-based blockchains. Chapter 7 discusses leveraging proofs-of-proofs to achieve interoperability between blockchains by allowing the free passing of information from one blockchain to another enabling proof-of-burn one-way pegs, two-way pegs, and smart contract derivatives. We give our conclusions and directions for future work in Chapter 8.

Chapter 1

Introduction

1.1 The Setting

1.1.1 Let there be Bitcoin

Before money, there was debt [64]. Money is a yardstick for measuring it. Sometimes it takes the form of a gold coin. Not useful in itself, one accepts it because one assumes other people will. Modern *fiat* money is not backed by gold, but takes the form of pieces of paper bills or, more often, bits in the computer systems of banks. Regardless of their manifestation, all forms of money are debt, which is a social relation [71].

Money functions as a *medium of exchange*, as a *common measure of value* or *unit of account*, as a *standard of value* or *standard of deferred payment*, and a *store of value* [73]. These functions of money rely on the relationship of the individual with the economic community that accepts money. Each monetary transaction between two parties is never a "private matter" between them, because it translates to a claim upon society [136].

This gives rise to the need of *consensus*. The economic community must be able to ascertain, in principle, whether a monetary transaction is *valid* according to its rules. In a good monetary system, parties of the economic community must globally agree on the conclusions of such deductions. In simple words, when someone pays me, I must know that they have sufficient money to do so, and that this money given to me will be accepted by the economic community when I later decide to spend it. This judgement of validity consists of two parts: First, that the money in use has been minted legitimately in the first place. Secondly, that this money rightfully belongs to the party who is about to spend it, and has not been spent before, to protect against *double spending*.

The problem of consensus is solved differently in different monetary systems. Gold coins had stamps whose veracity could be checked, while paper bills have watermarking features making them difficult to duplicate. Such physical features ensure the legitimacy of minting. The problem of double spending is trivial when it comes to physical matter: If I give a gold coin to someone, I no longer hold that gold coin and cannot also give it to someone else. When coins are digitized, the problem of *who owns what* is solved by the private bank and payment processors. A private bank centrally maintains the balance of a bank account to ensure a corresponding debit card cannot spend more money than it has. In this case, a vendor's terminal connects to the bank's servers to check the validity of the payment (and security can only be ensured while the terminal is online). These cases involve a *trusted third party*, the bank or the payment processor, to maintain a balance and make a judgement on whether a transaction is valid. The central bank is relied upon for the legitimacy of minting. Payment processors and banks who maintain account balances and make a judgement on whether a transaction is valid are relied upon to prevent double spending. The economic community depends on these third parties and trusts them for availability and truthfulness.

The cypherpunk political movement and the wave of cryptographers working on *protocols* in general have an inherent hatred for trusted third parties. For the former, they amount to centralization of political power which they wish to see eliminated. For the latter, it constitutes a technical challenge – if the role of the trusted third party is fully algorithmizable, why not replace the party by a protocol ran by the economic community themselves? It is somewhere in the intersection of the two that *blockchain* protocols appeared.

Bitcoin was invented by Satoshi Nakamoto in 2008. In a paper titled *A Peer-to-Peer Electronic Cash System* [116], Satoshi introduced the first *cryptocurrency* and the technology powering it, the *blockchain*. The paper accompanied an implementation of what has come to be known as *Bitcoin Core*, the first cryptocurrency wallet. Satoshi, whose name was soon deified within the blockchain community, was nothing but a Japanese pseudonym, the physical identity and whereabouts of the author or authors still unknown. Never short on drama, the space was shaken again when Satoshi mysteriously ceased all his online activity in 2011. His online persona disappeared without a trace, leaving the community leaderless, perhaps in an attempt to, in a sense, remove the last standing trusted third party from the picture.

As an electronic currency, Bitcoin enjoys many advantages. The transfer of a transaction takes a couple of seconds, while transactions become secure against chargebacks in about an hour. It is easy and cheap to move money across the world in large quantities, with transaction fees remaining constant regardless of amount moved (at the time of writing, approximately $0.05). The ecosystem is open, the reference client open source, and the community can develop their own software to work with it without requesting permission from anyone.

More interestingly, Bitcoin is the first *decentralized* electronic currency. Unlike all previous electronic currencies, it does not require any trusted third party for its operation. There is no company or operator with privileged access over Bitcoin – no CEO or corporate network. This makes the Bitcoin network *sovereign*: It cannot be shut down by a traditional court of law regardless of the desires of world governments, unless drastic measures are employed such as shutting down the Internet. There is no central entity to issue a subpoena to and it will continue to operate as long as there are users. This also makes the system uncensorable.

1.1.2 Coming to an agreement

Bitcoin is a peer-to-peer network. As such there are no servers and clients. Instead, similar to BitTorrent, there are simply nodes which connect to each other and exchange data. These form the global Bitcoin network, a connected graph of participants continuously exchanging financial data. They run open source code which can be inspected and modified by anyone, and are all treated as equals, none of them enjoying elevated privileges. Additionally, there is no implicit trust. It is accepted that some of them will behave adversarially and try to subvert the system by attempting to spend money they do not own, or by attempting to censor others.

At first glance, given that anyone can modify the code of their node, it seems that the situation is hopeless. Is it not possible for such a node to fake how much money they have? Additionally, even if they cannot conjure money out of nowhere, since money is represented as bits, is it not possible to simply duplicate their wallet and thus duplicate their money? In this setting, the consensus problem now becomes critical and particularly challenging. Bitcoin resolves such issues by making use of the *blockchain*. Bitcoin's security, and in turn our analysis, is for *arbitrary adversaries*. The powerful cryptographic setting gives us the tools not only to analyze particular adversarial strategies, but to prove our system can withstand *any* adversary, even ones we cannot imagine. As such, our security theorems will begin "For any adversary..."

To solve the consensus problem, Bitcoin makes a radical shift compared to previous systems. To allow participants to verify whether a transaction is valid and whether the payer has sufficient money to execute it, it reveals to *all* participants who owns how much money. This is done by dissemminating all transactions to all participants in the network in a gossiping fashion. These transactions are then stored by *every* other node in perpetuity. Any synchronizing node downloads these historical transactions and stores them, too. By looking at these transactions, it can deduce whether a party's claim to spend money is legitimate. The financial privacy of the parties is not trivially violated, because money is received into *public keys* instead of accounts corresponding to real names. Privacy can be significantly improved when a person employs a new public key for every transaction they are about to receive [2].

A transaction in Bitcoin is a payment order in the form of a string. It has a source, which corresponds to a previous transaction receiving the money, and a destination, which is a public key. In order for a transaction to be valid, the private key corresponding to the public key in the source must digitally sign the transaction, including the public key of the new owner. If Alice wishes to pay Bob, she looks for a transaction which has paid her money that she has not yet spent. She then creates a payment order in which she writes down the amount she wishes to pay to Bob, as well as his public key, and signs the order with her private key. She then broadcasts this transaction on the peer-to-peer network. Bob, who is connected to the same network, receives the transaction through one of his peers, and verifies its validity as well as the fact that his own public key has been used in the payment and that the amount is correct.

Each node organizes the transactions it sees on the network into a *ledger*, which is a sequence of transactions. This ledger determines which party has how much money. The node can then evaluate the validity of a new incoming transaction by assessing whether there are sufficient funds. For the monetary system to function, the parties must globally agree on a *common* ledger. If Alice receives a payment from Bob and her ledger depicts this, but Charlie's does not, then Alice's money received from Bob cannot be used to pay Charlie.

A malicious node can attempt to *double spend*. If Eve legitimately owns a coin, she can create a valid transaction tx paying Alice with that coin, as well as a valid transaction tx' paying Bob with that coin. If any of the two transactions are individually broadcast on the network, it will be considered valid. However, if both transactions are broadcast on the network, they cannot both be valid, because Eve only ever had one coin. As such, a strategy against double spending must be employed when such a sitatuation occurs.

Simple strategies do not work. For example, observing that Eve must be malicious to have created a double spend, the nodes could reject *both* of her transactions. However, this now enables Eve to first pay Alice for some service, but later invalidate that payment by also paying Bob. While Eve cannot profit from this, she can harm Alice. The alternative strategy of waiting a fixed time Δ to determine whether there have been any double spends prior to accepting a transaction also does not work. The reason is that Eve could broadcast her double spending transaction at a time just before Δ has elapsed. Because the network is not fully synchronized, some parties will receive the transaction prior to time Δ, while others will receive it after time Δ, causing the network to disagree about whether the first transaction is valid.

1.1.3 Chains of blocks

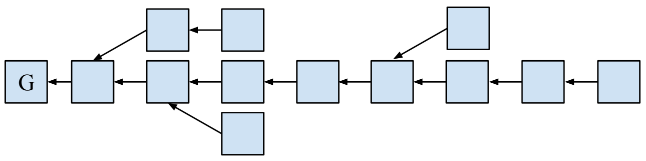
To solve this problem, Bitcoin mandates that nodes place transactions in chronological order. As Eve can lie about the timestamps of her transactions or broadcast them at a delayed time, timestamps cannot be trusted.

Transactions on the network are collected by nodes into bundles called *candidate blocks*. These blocks are simply strings concatenating the transactions together with some metadata. The nodes that collect them are known as *miners* and anyone can participate as a miner. Each miner attempts to *mine* the candidate block it has created, turning it into a *block*. Mining involves solving a moderately hard problem and has a small, but not negligible, probability of success. When the problem is successfully solved, we say that a block has been *found* or *mined*. The block contains the transactions, metadata, as well as the solution to the moderately hard problem. The moderate hardness of mining ensures blocks are created at a predetermined expected rate. For Bitcoin, this rate is 10 minutes. Once a block is found, it is broadcast to the rest of the network. A transaction is considered *confirmed* once it has been included in a block. Transactions inside blocks are placed in an arbitrary order determined by the party creating the block. Block validity mandates that no double spends appear within the same block. Once a transaction is confirmed in a block, it is considered to have been placed in order among other transactions.

A block includes a pointer to the most recent block seen by the node. This creates a *chain of blocks* or *blockchain*. As each block contains transactions, reading the transactions within the blockchain in order gives a unique ledger signifying which transaction happened before which other transaction. The validity of the blockchain requires that the transactions in the new block do not conflict with transactions in the past chain, and so reading the transaction sequence of a valid blockchain always gives a valid transaction sequence, without any double spends. Whenever a node receives a blockchain from the network, if it is longer than its own blockchain in the number of blocks it contains, then it abandons its own blockchain and adopts the longer one. This is known as the *longest chain rule*. The first block of the blockchain, called the *genesis* block, was fixed by the protocol at the time of initial deployment.

Surprisingly, this simple rule solves the consensus problem. Because blocks are

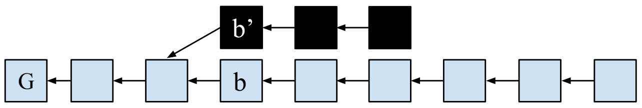
Figure 1.1: A blockchain with some temporary forks. Squares depict blocks.



broadcast once every 10 minutes on average, this leaves a sufficient window of silence between blocks, leaving enough time for the message of the newly found block to travel through the peer-to-peer network, so that honest parties will adopt the same chain. If it happens that two blocks are found almost simultaneously by chance, some parties will mine on top of one chain, while others will mine on top of another. Once a new block is found, one of the two chains will grow longer and the population will migrate to the longest chain, abandoning the shorter chain. If no other blocks are found and broadcast in close temporal proximity to this one block, all honest will adopt this newly generated block and converge. These situations are known as *temporary forks*. The chance of two blocks being found simultaneously again and again is small, and hence eventually the network will converge to a common view. This is illustrated in Figure 1.1. The honest parties may have some small disagreement about which few blocks are at the end of the blockchain. However, they will share a large *common prefix*. As such, a transaction is considered safe once it has been buried under a sufficient number of blocks.

Once a transaction tx is confirmed by becoming included in some block b and that block is buried under a sufficiently large number of blocks, it becomes difficult for the adversary to double spend using a conflicting transaction tx'. Consider the execution in Figure 1.2. As blockchains extending b do not validate if they contain tx', the adversary must *fork* the blockchain and start mining on top of the parent of b. Consider the first block b' of that fork in which tx' is confirmed. This becomes a mining race between the honest parties, who are mining on top of the longest chain extending b and confirming tx, and the adversary who is mining a new chain on top of b' and confirming tx'. If we assume the honest parties control the majority of the computational power on the network, the chain that the honest parties mine will grow faster than the one extended by the adversary. This is known as the *honest majority assumption*.

Figure 1.2: A minority adversary fails to double spend a deeply burried transaction.



Let us now see why blocks are generated approximately every 10 minutes. When a miner is asked to find a block, they collect the unconfirmed transactions into a string $\overline{x}$. They then attempt to find a nonce such that $H(\overline{x} \parallel \mathsf{nonce} \parallel h') \leq T$, where T is a constant *target*, H is a cryptographically secure hash function, and h' is the hash of the previous block. When we arithmetically compare hash values like this, we treat the bit output of the hash function as a binary number and interpret it numerically. In the case that T is a power of 2, this equation is equivalent to requiring every block hash must have its most significant bits set to 0. As the hash function has an unpredictable output, this problem becomes more difficult as T becomes smaller. The only known way to solve the problem is by brute forcing the nonce until a suitable value is found. This process is known as *proof-of-work.* While T is constant for short periods of time, it is adjusted over longer periods of time to achieve the desired block production rate. This is known as *variable difficulty.*

Proof-of-work systems take significant computational resources to secure, because they must ensure honest parties are putting in more effort than any possible adversary. While the computational resources go towards securing the network and are not wasted, a question is whether an alternative protocol can achieve comparable security with lower computational power consumption. Such a protocol would be more efficient and environmentally friendly. Towards this direction, the idea of *proof-of-stake* was introduced. The concept is the same as with proof-of-work in that blocks and chains are formed which confirm transactions and are issued at a controlled rate. However, the assumptions are changed. Instead of assuming honest majority by computational power, *honest majority by stake* is assumed. *Stake* denotes the amount of money one owns within the system. This assumption essentially mandates that the majority of the money in the system is owned by participants behaving honestly. As money changes hands with time, this assumption is presumed to hold throughout the execution despite *shifting stake.* Given that stake is owned by cryptographic keys, instead of having nodes attempt to solve the proof-of-work computational puzzle by brute forcing, the nodes are simply asked to sign off a block occasionally using the same key that signifies money ownership.

Therefore, a node is asked to sign off a block every once in a while. This node, known as a *leader*, is a representative among the rest of the stakeholders and is chosen at random among the stakeholders as follows. Among all the coins in the system, a coin is selected uniformly at random. This is known as *following the satoshi.* As richer participants own more coins, their probability of being selected is higher. While this sampling process will sometimes elect adversarial nodes, it elects nodes according to the underlying stake distribution. We know that following this election process leads to a blockchain which behaves similar to proof-of-work: While there can be short temporary forks due to adversarial behavior, soon enough the honest parties prevail and keep working on the longest chain. One challenge in the implementation of this protocol is how to do the sampling among the coins in a manner that is unpredictable. Due to technical reasons, the protocol *freezes* the stake distribution for a specific duration called an *epoch* during which an old view of the stake distribution is used for sampling. Once an epoch has passed, the altered stake distribution is observed and a new snapshot is taken. Under the assumption that stake shifting does not occur at a very large rate, the *bounded stake-shift assumption*, this sampling method is secure.

One important detail that is needed to make the proof-of-stake protocol secure is to ask parties to *evolve* their private keys once every epoch. In this process, an honest party uses their old private key to produce a new one in a way that allows anyone holding the respective public key to verify the evolution was truthful. The old key is then destroyed. This is useful to ensure that each key corresponds to a particular epoch. A key corresponding to an epoch cannot be used to create votes on blocks corresponding to older epochs, giving the system *forward security.* This protects against an event in which a majority stakeholder, due to stake shift-

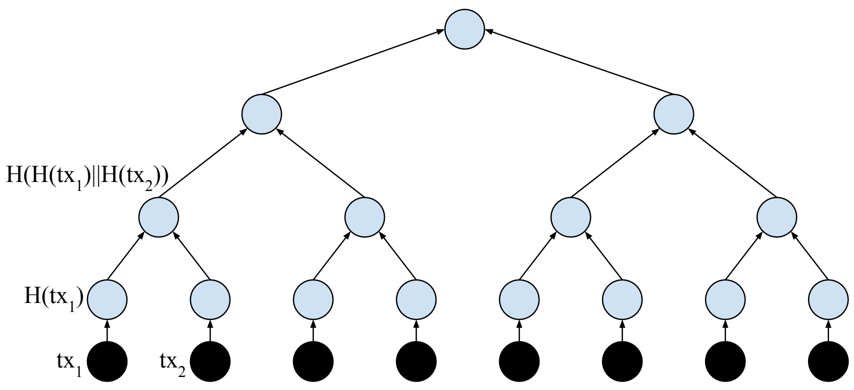


Figure 1.3: A Merkle Tree of transactions making use of a cryptographicallys secure hash function H. Black nodes indicate raw transactions. Lightly shaded nodes indicate hashes. The root is shown at the top.

ing, becomes a minority stakeholder. The honest majority assumption allows that stakeholder to then become corrupt. It is important that the adversary corrupting the minority stakeholder cannot reuse old keys that corresponded to the time when the stakeholder was holding the majority of the stake.

1.2 Motivation

1.2.1 Superlight Clients

Given an honestly adopted chain, consensus security is based on the premise that it is difficult to create a competing chain which deviates significantly and has more proof-of-work than the existing honestly adopted chain. To determine which financial history is the valid one, a *verifier* node booting for the first time into the network, connects anew to multiple *provers*, at least one of which is assumed to be honest. The verifier then downloads all available blockchains from its peers and adopts the longest one. The verifier must choose the chain that respects the protocol rules and corresponds to the most proof-of-work. In order to do that, the proof-of-work of every block in the chain must be presented and verified.

Nodes on the blockchain that maintain the whole history of transactions and chain are known as *full nodes*. Typically, miners are full nodes, but some non-mining nodes are also full nodes. Proof-of-work blockchain clients such as mobile wallets today are based on the Simplified Payment Verifications (SPV) protocol, which was described in the original Bitcoin paper [116], and allows them to sychronize with the network by downloading only block *headers* and not the entire blockchain with transactions. However, such initial synchronization still requires receiving all the block headers.

Instead of having each block contain all of its transactions and performing proof-of-work on it, transactions are first hashed into a special structure known as a Merkle Tree. A Merkle Tree of a transaction sequence $\overline{x}$ is structured as illustrated in Figure 1.3. Using a cryptographically secure hash function H, each transaction is hashed to obtain its transaction *id*. Subsequently, each two ids are concatenated together and hashed again to obtain their parent. These parents are again concatenated and hashed together to obtain *their* parent, until we arrive at a single *root* node x which contains a hash representation of every transaction in $\overline{x}$. The proof-of-work of the block is then computed by finding a nonce such that $H(x \,\|\, \mathsf{nonce} \,\|\, h') \leq T$. The transactions $\overline{x}$ are known as the *block content*, while the portion $x \,\|\, \mathsf{nonce} \,\|\, h'$ which is hashed for proof-of-work purposes is known as the *header*.

This chain C of block headers grows linearly in time. To put the problem in perspective and motivate the question, at the time of writing, Ethereum's blockchain (as measured by the amount of data that needs to be downloaded to synchronize a full node) is currently more than 250GB, and the size of block headers is 4.6GB. The latter amount is currently too large for a user-friendly mobile wallet.

The problem we concern ourselves with is whether it is possible to optimize this protocol, achieving communication $o(|\mathsf{C}|)$. We study the question of whether better protocols exist and in particular if downloading fewer block headers is sufficient to securely synchronize with the rest of the blockchain network. Our requirement is that the system remains decentralized and that useful facts about the blockchain (such as the Merkle root of current account balances in Ethereum) can be deduced from the downloaded data. This means that we don't want to utilize techniques such as checkpointing (in which the verifier software is patched by its trusted developer to include a later block as a *neon genesis* replacement to the genesis block) or trusting a server to give us the correct blockchain.

Our setting is as follows. A *superlight* verifier wakes up having only the first block of the blockchain, the genesis block. In the meantime, the blockchain has grown and contains many blocks. The verifier wishes to know whether a transaction is confirmed to decide if a payment has been made. It connects to multiple other *full nodes*, that we term *provers* throughout this work, who maintain the whole blockchain. At least one connection will be to an *honest node*. All the provers send claims to the verifier, some claiming that the transaction is confirmed, while others that it is not. The goal of the verifier is to decide which of these claims is true. The claims are accompanied by *evidence* to illustrate them, but must be short in length so that the client can synchronize quickly. In particular, we aim for exponentially shorter messages $\mathcal{O}(polylog(|\mathsf{C}|))$ instead of the standard $\mathcal{O}(|\mathsf{C}|)$ that an SPV node requires. These superlight nodes are useful when a client such as a mobile wallet wishes to synchronize with the network quickly. Other facts beyond transaction inclusion, such as account balances, are also useful to prove.

The question the verifier is trying to answer is not simply whether the transaction has been included in *some* valid block, but whether that block belongs to the longest chain and hence the economic participants will also agree that the transaction took place. For the proof-of-work case, this chain corresponds to the chain containing the most proof-of-work that respects the protocol rules (such as the occasional recalculation of the target T due to changes in the total mining power of the network). For proof-of-stake, this chain corresponds to the longest chain which has valid signatures by the majority of the stakeholders at each point in time, keeping in mind that stake is shifting from epoch to epoch. In both cases we are trying to create succinct proofs *about* the fact that proof-of-work or proof-of-stake took place without presenting every proof-of-work nonce or every proof-of-stake signature. This gives rise to *Proofs of Proof-of-Work* and stake.

While it is useful for mobile wallets to synchronize quickly, if a good protocol for superlight nodes is developed, one natural question that arises is whether the same protocol can also be used for full nodes and miners. This would help quickly create a new miner and have it working on the chain without needing a long synchronization time. Can miners, like superlight clients, also synchronize quickly with the rest of the network while achieving security comperable to a full node? And if so, do we need any extra assumptions and what are the security limitations of such a model?

1.2.2 Interoperability

Since the invention of Bitcoin in 2009, numerous other cryptocurrencies have followed, improving upon Bitcoin on several aspects. The most prolific of these is *Ethereum* [35], which explores the concept of *smart contracts*. These Turing-complete programs enable developers to define complex conditions which must be satisfied to spend money, beyond the simple signatures that Bitcoin allows as conditions for spending. They are programmed in specialized programming languages such as Solidity and run on top of the Ethereum Virtual Machine [151]. Each contract is a sovereign entity that can own money and define the rules under which this money can be spent. These contracts execute autonomously. Their correct execution is verified by miners on the network, in a similar way that signatures are verified in Bitcoin's case.

Other cryptocurrencies that include significant contributions and experimentation are *Litecoin* [101] which aims to be more egalitarian [74]; *Monero* [149] and *ZCash* [114, 69], which improve upon the generally poor [111, 132, 66] privacy of Bitcoin; *Namecoin* [105] which was a first attempt at creating a decentralized DNS alternative; *Dogecoin* [107], which experiments with inflationary economics in contrast to Bitcoin's deflationary nature; *Bitcoin Cash* [95], which experiments with larger block sizes; and *Cardano* [89], which uses proof-of-stake instead of proof-of-work.

It is possible to *trade* one coin for the other. For example, if one wishes to exchange Bitcoin for Ethereum, they need to find a counterparty who wishes to exchange Ethereum for Bitcoin (this is generally easy to do through centralized services). However, each of these blockchain systems remains isolated. The concept of a *cryptocurrency* and its respective *chain* remain intertwined: A Bitcoin lives in the Bitcoin chain, while an Ether lives in the Ethereum chain. The *interoperability* problem pertains to the ability to move a cryptocurrency from its native chain to a remote chain, a one-way peg, and then back, a two-way peg. If Bitcoin and Ethereum were interoperating, it would be possible to move one Bitcoin from the Bitcoin chain to the Ethereum chain and back. The Bitcoin would retain its *nature* during its lifespan within the Ethereum chain. It would always be a Bitcoin, not an Ether. For example, it would maintain the same exchange rate against other currencies. While on the Ethereum chain, it would also enjoy the benefits of the Ethereum ecosystem. For example, it would make itself subject to smart contracts and enjoy the faster confirmation speed of Ethereum. At the same time, it would make itself subject to the security assumptions of Ethereum. For example, it could subject itself to more limited security protection under a smaller amount of honest computational power devoted to the mining of the chain it lives on. The bitcoin would have the benefits and shortcomings of the Ethereum environment only as long as it lives on the foreign blockchain. When it returns back to its native chain, it should again behave within the context of the Bitcoin blockchain. The bitcoin

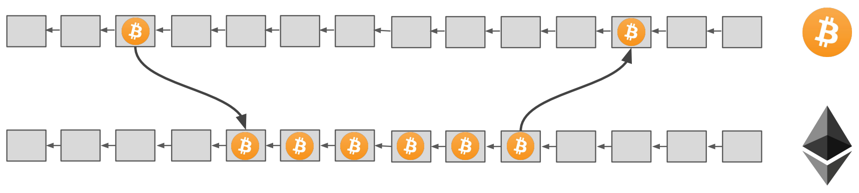


Figure 1.4: A motivating but imaginary two-way peg. A bitcoin is moved from its native Bitcoin chain to the Ethereum chain and back. While on the Ethereum chain, it is still a bitcoin.

has become *decoupled* from its blockchain. This is illustrated in Figure 1.4.

More generally, and beyond the transfer of a cryptocurrency asset from one blockchain to another, it is useful to allow one blockchain to consume information pertaining to another. Today, smart contracts are confined to access data only within the blockchain they run on, such as data maintained within the state of the smart contract itself. Access to external data requires a trusted third party or group thereof to vouch for the data validity [159]. As contracts can represent complex financial conditions and trust relationships, it is useful to give them the ability to take decisions based on *events* that take place on different chains. One example could be an Ethereum contract which gives out shares in a decentralized company (in the form of tokens) conditioned on the fact that a series of Bitcoin payments have been made in regular intervals during a predetermined period of time. Another example could be an insurance contract running on the Ethereum blockchain which, conditioned on the fact that a particular Bitcoin account fails to receive a predetermined amount of money until a particular point in time, releases an Ethereum payment to its policyholder. As one can readily see from these not so complex examples, such contract conditions that go beyond a simple payment cannot be readily encoded in the form of an atomic swap or a two-way peg and may not even have a definite counter-party in each chain.

As such, we are looking for a generic cross-chain communication mechanism which allows blockchains to interoperate, achieving one-way pegs, two-way pegs, as well as generic information transfers. Such a system should allow smart contracts on one blockchain to receive and react to events taking place on another blockchain without the need of trusted parties. We term this generic communication mechanism a *sidechain*[1]. Despite their widely agreed usefulness there exist currently no sidechain constructions that are decentralised, efficient, and generic at the same time. One critical challenge in desigining these protocols is that the aim is to allow the two chains to remain *miner isolated*. This means that, while users could maintain wallets in multiple chains and observe them for transactions of their interest, as well as forward information between the chains, the *miners* that run the consensus protocol in each chain cannot be required to connect to multiple networks. For example, it is desirable that Ethereum can react to Bitcoin events without requiring the Ethereum miners to connect to (or even know about the existence of) the Bitcoin network.

[1]We note here that previous work has used this term in a narrower context than the generic information transfer we propose here. This is one reason why the term *cross-chain communication* for a generic protocol may be preferable.

Interoperability can also be a useful mechanism to enable interfacing between decentralized blockchains and more traditional centralized systems such as accounts whose balances are held by custodians and are subject to the traditional laws of particular countries. Such a construction could allow for a smoother transition from the legacy monetary system to a blockchain-enabled system. The same mechanisms that allow a smart contract in Ethereum to verify a remote Bitcoin payment can be used to allow a *permissioned* chain (in which the consensus is achieved by majority voting between a centralized committee) to verify a Bitcoin payment without requiring its consensus maintainers to connect to the Bitcoin network. It is, of course, trivial to write an Ethereum verifier that checks whether a transaction has been confirmed within such a permissioned network (by simply verifying and counting signatures). Therefore in this thesis we will concentrate on consuming data from decentralized chains, as the other way around is straightforward.

While sidechains were not originally proposed for scalability purposes, they can also be used to off-load the load of a blockchain in terms of transactions processed. For this purpose, a particular blockchain, which we will call the *mainchain*, that wishes to off-load its load can create multiple separate sidechains, that maintain consensus independently. A cross-chain protocol can then be used to connect each side chain to the main chain, with the main chain functioning as a financial hub. While communication protocols between the main chain and the side chain can be symmetric, their use cases are different. In this application, the use of the main chain is as a permanent store of value as well as an interface between different side chains, while each side chain is used for everyday transactions. As long as 2-way pegs are enabled, a particular sidechain can offer specialization by, e.g., industry, in order to avoid requiring the mainchain to handle all the transactions occurring within a particular economic sector. As long as the majority of transactions does not cross economic sectors, this provides a straightforward and simple way to "shard" blockchains [106, 92, 155]. Another way of sharding could be by geographical location.

Lastly, a child side chain can be created from a parent main chain as a means of exploring a new feature, e.g., in the scripting language, or the consensus mechanism without requiring a soft, hard, or velvet fork [87, 158]. The side chain does not need to maintain its own separate currency, as value can be moved between the sidechain and the main chain at will. If the feature of the sidechain proves to be popular, the main chain can eventually be abandoned by moving all assets to the sidechain, which can become the new main chain. In fact, such experimentation with new features was one of the original motivators for sidechains [12].

Given the benefits listed above, there is a pressing need to address the question of sidechain security and feasibility, which so far has not received any formal treatment.

1.3 Our Contributions

1.3.1 Superblocks and NIPoPoWs

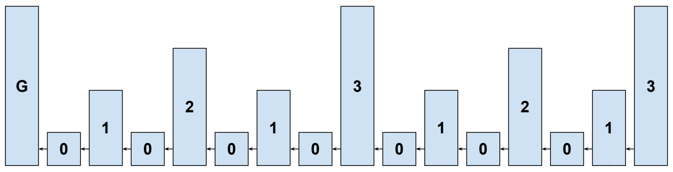
We create a decentralized blockchain client or verifier which, having only *genesis*, connects to multiple provers, at least one of which is honest, is able to ascertain the confirmation of a transaction. Using its full local chain, each prover generates a succinct proof and sends it to the verifier. Adversarial provers can send anything in the place of a proof. By comparing the proofs in terms of the amount of proof-of-work they encode, the verifier deduces which blockchain contains the most proof-of-work without receiving and validating every block header. The proofs the provers send are only generated once and do not require multiple interrogation questions from the verifier. As such, these proofs are non-interactive and we call them *Non-Interactive Proofs of Proof-of-Work* (NIPoPoWs).

To create such succinct representations of work, we look at the distribution of block hashes in the chain. Every valid block B satisfies the proof-of-work equation $H(B) \leq T$ where T is the mining target, but some blocks satisfy it better than others. Some blocks so happen to have a hash with value *much* lower than T, even though this is not required for validity and is not intentional. For example, some blocks will satisfy $H(B) \leq \frac{T}{2}$. Concretely, because the hash function is uniformly distributed, in expectation *half* the blocks will satisfy $H(B) \leq \frac{T}{2}$, a *quarter* of them will satisfy $H(B) \leq \frac{T}{4}$, an *eighth* will satisfy $H(B) \leq \frac{T}{8}$, and in general only a $\frac{1}{2^\mu}$ fraction of blocks will satisfy $H(B) \leq \frac{T}{2^\mu}$. If a block satisfies this inequality for some $\mu \in \mathbb{N}$, we say that it is of *level* μ and call it a μ-superblock (and note that a μ-superblock for $\mu > 0$ is also a $(\mu - 1)$-superblock). The probability of a valid block B being a μ-superblock is:

$$\Pr[H(B) \leq \frac{T}{2^\mu} | H(B) \leq T] = \frac{1}{2^\mu}$$

Under this light, the blockchain looks as illustrated in Figure 1.5. Of course, because block hashes behave randomly, this image will be probabilistic and blocks may not be precisely distributed as expected.

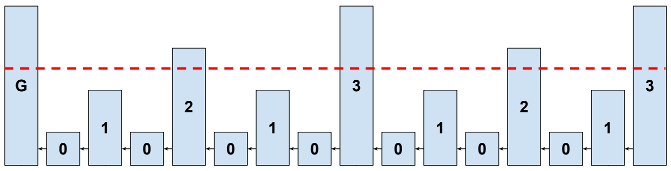
Figure 1.5: Superblocks distributed within a blockchain. Higher levels have achieved a higher difficulty during mining.



When a chain C is mined, there will therefore exist a subsequence of it, consisting only of blocks of level μ, which is going to be only $\frac{|\mathsf{C}|}{2^\mu}$ blocks long. The core idea of the construction is this: Instead of sending the full chain, the prover chooses a level μ and sends this as a *representative of the underlying work*. Presenting a block of level μ captures the fact that work of about 2^μ has happened around it without presenting that work itself. As such, a superblock is a way for the prover to *sample* the blockchain in a way that can convince the verifier: When the verifier receives m blocks of level μ, it can deduce that approximately $m2^\mu$ regular blocks must exist around those superblocks. Constructions based on this simple idea instantiate NIPoPoWs by leveraging superblocks and we call them *superblock NIPoPoWs*.

A simplified description of our protocol then works as follows. Initially, some value m is fixed, representing the number of blocks that the verifier wishes to receive to feel safe. This m is a constant parameter. The honest prover then chooses the highest level μ which has at least m blocks at that level. Choosing any level above μ would not satisfy the verifier, as fewer than m blocks would be transmitted. Choosing any level below m would be wasteful, as more blocks would have to be transmitted. This prover choice is illustrated in Figure 1.6. Suppose that the verifier receives two proofs from two provers, one of which is honest while the other adversarial, and wishes to compare them. The verifier first checks that all the blocks it has received really are μ-superblocks by verifying the proof-of-work equation parameterized by μ is satisfied, and that each proof it has received has at least m blocks. Then, just as an SPV verifier would compare the length of two full chains, the NIPoPoW verifier now simply *counts* the number of blocks it has received from the two provers and announces that the one with the most blocks is the winner. If the verifier receives the proofs π_1 and π_2, then the decision is simply the result of the comparison $|\pi_1| > |\pi_2|$.

Figure 1.6: A superblock NIPoPoW prover chooses the threshold (dashed line) corresponding to level $\mu = 2$ for the verifier requirement $m = 4$.



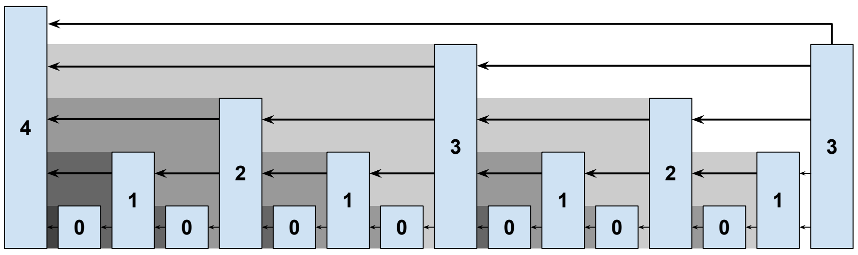
The crucial point in terms of security is that an adversary cannot fake this set of superblocks without actually putting in the work. An adversary that produces a μ-superblock will also in expectation generate some 2^μ regular blocks in the process (even though the adversary may of course choose to discard these). Because the adversary has minority mining power, an adversary cannot create a longer sequence of μ-superblocks faster than the honest parties create one, for the same reason that an adversary cannot create a longer regular blockchain faster than the honest parties create one.

Let us count how many different levels μ there are in a chain. If the chain has length $|\mathsf{C}|$, then going up to level 1 cuts the chain in half, and we expect to see only $\frac{|\mathsf{C}|}{2}$ superblocks of level 1. Moreover, this continues with subsequent increases in level, until at level $\mu = \log |\mathsf{C}|$ we expect to find only $\frac{|\mathsf{C}|}{2^\mu} = 1$ block. As soon as we get to level $\log |\mathsf{C}| + 1$, we expect to find no more blocks of that level or above. The number of levels is therefore $\log |\mathsf{C}|$.

To ensure that the blocks sent by the prover cannot be reordered in the wrong chronological order, just as in the regular underlying blockchain, we need to introduce pointers that point between consecutive blocks of the same level. As such, a μ-superblock must include in its contents a pointer to the most recent μ-superblock that was mined before it. Unfortunately, we cannot add just these exact pointers to the block contents, because the pointer data must be included in the contents that are hashed during the attempt to find proof-of-work. It seems that we must predict what level a block will have prior to it being mined, but its level depends on its hash, which is a product of its mining. Therefore, we will simply include all the pointers that could be needed regardless of what level the block achieves. Prior to mining any block, the miner collects a pointer to the most recent superblock of each level it has seen so far. It places these pointers in a list which it then includes in the block it is mining. When the block is mined, regardless of what level it achieves, it will have a pointer to the most recent block of its own level.

The number of pointers that need to be included in this manner is small because the number of levels the blockchain will ever reach is $\log|\mathsf{C}|$. This interlinking is illustrated in Figure 1.7. To avoid premining of superblocks (blocks that were mined prior to the creation of the genesis block), we require that the interlink vector of every block also contains a pointer to the genesis block. By having miners add these extra pointers to blocks, the verifier can check that the blocks in each proof presented have been mined in the order given. We see that NIPoPoWs are *subchains* of the blockchain in that they form subsequences of the block sequence and also maintain pointers across. The change of adding interlink pointers seems on the surface to require that miners change their behavior and so a hard fork (a breaking change of consensus protocol rules) is required. However, the upgrade can be deployed using a soft fork (a backwards-compatible change), or even without requiring any miner upgrade in what we introduce as a *velvet fork*.

Figure 1.7: The interlinked blockchain. Each superblock is drawn taller according to its level. A new block links to all previous blocks that have not been overshadowed by higher levels in the meantime.



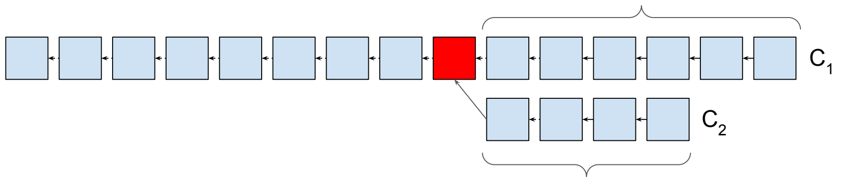
Because the two provers may send chains of different levels, it may be necessary for the verifier to compare the two proofs according to different levels. For this purpose, the verifier weights the proofs according to their level prior to comparison. The result of the comparison is then $2^{\mu_1}|\pi_1| > 2^{\mu_2}|\pi_2|$, where μ_1 and μ_2 indicate the level of proofs π_1 and π_2 respectively. If the two proofs are at the same level, this comparison reduces to a simple comparison by count. We term this comparison which can happen *across* levels the *charity* construction. As we will see in later chapters, the verifier can choose to compare the two proofs at a common level, simplifying the construction, albeit to some loss in security. We term the construction which compares across a common level the *distill* construction.

1.3.2 Comparing NIPoPoWs

While the protocol we have presented so far works *in expectation* against any adversary, we want to design a protocol that also works *with overwhelming probability.* Let us now discuss the role of the security parameter m that the verifier requires for safety. The property we can prove from the honest majority assumption is that, in a given period of time that is sufficiently long, the honest miners mining on their regular 0-level chain will generate more μ-superblocks than an adversary with any mining strategy. This "sufficiently long" period of time is ensured by the m security parameter that the verifier requires. If any of the two superchains consists of at least m superblocks, this must necessarily have required a long time to generate as long as m is sufficiently large (in later chapters, this will be made precise with a Negative Binomial Chernoff bound). If a sufficiently long time has passed, the length of the superchains will be close to their expectations and correspond to the mining power of the adversary and the honest parties respectively (this will later be made precise with a Binomial Chernoff bound). Because the adversary has minority mining power, their superchain will be shorter with overwhelming probability. As we increase the parameter m to some reasonable value (say $m = 128$), the probability that an adversary is able to attack our protocol drops exponentially in m (according to a function asymptotically similar to 2^{-m}).

If the honest and adversarial verifiers send chains that share some blocks, there is some probability that the adversary is able to convince an honest verifier of an invalid claim. What we wish to ensure is that the honest NIPoPoW verifier will never be made to disagree with a corresponding SPV verifier. For the argument outlined above to hold, we must ensure that the verifier requests from the two provers at least m superblocks that are *distinct* in each of their proofs. This ensures the adversary will not be able to reuse blocks from the honest chain in her proof. Towards this purpose, the verifier asks the two provers to choose a level μ such that there are at least m blocks in each of their proofs *after* the most recent block shared among their chains (the *lowest common ancestor block* or *LCA block*) as illustrated in Figure 1.8.

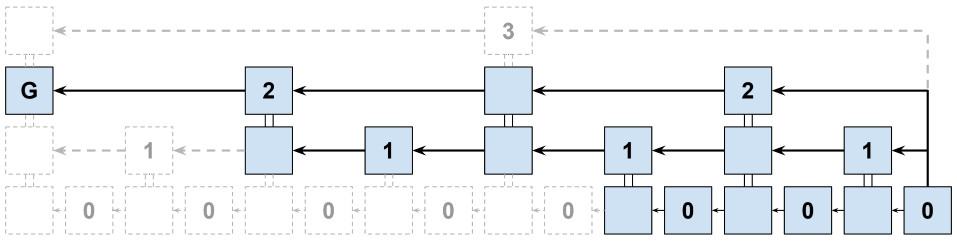
Figure 1.8: A comparison across two chains sharing an LCA. The comparison must be performed on the independent subchain suffixes after the highlighted block.



This can be solved by introducing interactivity in a protocol that works as follows. Initially, both provers send their highest level μ that has at least m blocks to the verifier. The verifier inspects both proofs and finds their LCA block b. He then sends the LCA back to the provers and requests more information. Each prover subsequently finds the highest level μ' that has at least m blocks *after block* b. He sends all blocks of level μ' that follow b. The verifier again inspects the two proofs and finds their new LCA block b'. The process continues until one of the two provers is not able to keep up with the interrogation. The number of interrogation steps will be logarithmic, as they are bounded by the number of levels. Because there can be no two chains at level 0 which are both long and significantly diverge, one of the two provers will necessarily fail.

While it seems that this requires some interactivity, in reality the provers can provide sufficient evidence upfront so that no interrogation is needed and the verifier can compare proofs completely offline. The prover must ensure that there are sufficient blocks in the proof to enable a comparison regardless of which block is deemed to be the LCA block (as the adversary can create a proof which essentially chooses the LCA in her favour). As such, no matter what block in his proof is chosen as the LCA block b after which the comparison will be performed, the honest prover wants to ensure he will be successful. The prover includes sufficient blocks so that for every block b in his proof (a candidate LCA), there exists some level μ for which

Figure 1.9: A Non-Interactive Proof of Proof-of-Work based on superblocks. Multiple squares stacked above one another illustrate the same block which spans multiple superblock levels. The solid blocks are included in the proof. The dashed blocks are not selected for proof inclusion at that level.



at least m superblocks of that level will appear after b. Furthermore, to ensure no work is missed, the honest prover wants *all* his blocks of the chosen level to appear after b.

This requires us to build a prover that sends blocks at multiple levels. A summary of the construction is then as follows. The prover first chooses the highest level μ which has at least m blocks. He sends all μ-superblocks. Then, for each lower level $j-1$, he sends sufficient blocks of level $j-1$ to cover the same range that the last m blocks of level j span. The blocks that are selected for sending in such a NIPoPoW are illustrated in Figure 1.9. In this example, the protocol is working with parameter $m = 3$. Initially, the prover chooses the *highest* level μ that has at least $m = 3$ blocks. In this case, he selects $\mu = 2$ which has $4 \geq m$ blocks, as level 3 would be insufficient with only $2 < m$ blocks. All 4 blocks of $\mu = 2$ are included. Subsequently, the last $m = 3$ blocks of level $\mu = 2$ are considered and the 6 blocks of level 1 that span the same range are also included. Lastly, the last $m = 3$ blocks of level $\mu = 1$ are considered and the last 6 blocks of level 0 that span the same range are also included. The final proof is 10 blocks long (because some blocks belong to multiple levels, but do not need to be repeated in the proof).

To compare two proofs π_1 and π_2 of this style, the verifier first finds the LCA block b among them. He then chooses a level μ such that at least one of the provers has provided m blocks of level μ after b and compares the count of distinct blocks at that level. Once we introduce our full chain traversal notation, this comparison can be expressed elegantly using the inequality

$$|\pi_1\{b{:}\}{\uparrow}^{\mu}\,| > |\pi_2\{b{:}\}{\uparrow}^{\mu}\,|\,.$$

Even though multiple levels have been included in the proof, the proofs only take logarithmic size and are therefore succinct. The reason is that the number of levels is logarithmic, while the number of blocks included in each level is constant and approximately $2m$ per level, bringing the total to $2m \log |\mathsf{C}|$.

1.3.3 Superlight Mining

Having created this construction, it is now possible to build *superlight wallets* that can synchronize exponentially faster than standard SPV wallets. This still requires provers to be full nodes and have access to a complete chain. A natural question that arises is whether it is possible to make a NIPoPoW verifier that has already received a verified NIPoPoW take the place of a standard prover. The answer is *yes*, because these proofs are deterministic and can be simply replayed by nodes that have downloaded them. Additionally, once a NIPoPoW verifier has synchronized with the network, any new block that is mined on top of the existing chain can also be (fully) verified by them and adopted by them. Thus, even though the verifier does not hold the whole chain, it can keep downloading and verifying new blocks from the network. The next question is whether a verifier who holds a NIPoPoW π and receives a series of new blocks $b_1, b_2, \cdots, b_k$ from the network can convince another verifier that these blocks now belong to the best chain. This is possible because superblock-based NIPoPoWs have the *online* property: Consider a NIPoPoW π created for a chain C. If b_1 is created on top of C, then it is possible to evolve π into a new NIPoPoW π_1 which pertains to the chain $\mathsf{C} \parallel b_1$ without ever seeing C. This is because the new NIPoPoW π_1 only needs to contain (some) blocks from π as well as b_1 itself. This can be used to keep evolving NIPoPoWs as blocks are added, always only keeping the latest among them. Previous blocks and old NIPoPoWs can be garbage collected, allowing the node to function with a state of logarithmic size.

The node that has synchronized using a NIPoPoW and only holds a NIPoPoW can function as both a prover and a verifier and even be able to deal with cases of temporary forks. In fact, the node can even *mine* on top of their NIPoPoW. Because it holds the most recent block of the blockchain, it simply creates a new candidate block that points to it and attempts to solve proof-of-work accordingly. If he succeeds, he can broadcast the new block to the rest of the network and update their local state by evolving their NIPoPoW in light of the new block. As long as the majority of the mining power is honest, even *all* miners can upgrade to this logarithmic protocol so that no one really has to ever save the whole chain. The idea here is that the full history of transactions is not necessary to achieve economic consensus (i.e., who holds how much money or how much money remains unspent); what is needed is to be able to determine whether a new transaction is valid, and for this it suffices to know the current state of the system, be it unspent transaction outputs or account balances. This technique, which provides exponential improvements in the size of state storage, we term *logarithmic space mining*.

1.3.4 Proofs of Proof-of-Stake

Different techniques can be used to compress consensus state in proof-of-stake blockchains. Unlike proof-of-work in which the work of a block is a stand-alone property that can be verified by the verifier, the verification of the proof-of-stake signature on a block requires the verifier to know the stake distribution of the current epoch so as to be able to decide whether the key making the signature corresponds to a correctly elected leader. However, having the verifier see the whole stake distribution would be inefficient. Instead, we propose a construction in which the verifier only sees a small sample of the stake distribution per epoch. The protocol looks at the leaders that were elected to mint blocks throughout the epoch, and assembles a number of these leaders into a *committee*. Since the block leaders are selected in a follow-the-satoshi fashion by respecting the evolving stake distribution, the committee members are selected in this way too. It has been shown that committees selected once per epoch in this fashion will have honest majority *by count* as long as the underlying stake distribution that was used for the selection process has honest majority by stake.

The light stake verifier looks at one such committee per epoch. For each epoch, the committee signs off a *certificate* which attests to the election of a new committee for the next epoch. The committee members themselves are full nodes who, in the honest protocol, only sign off such certificates attesting to the correct new committee. The verifier checks that the certificate has been signed by the majority of an epoch's commitee. By verifying these certificates, the verifier can ensure that he holds a representative sample of keys for each epoch. These keys corresponding to the committee of each epoch can then be used to sign statements alleging that a transaction took place during their respective epoch. As the verifier knows the committee for each epoch, they can verify these statements also.

While this construction has eliminated the need for the verifier to maintain the whole stake distribution as it shifts from epoch to epoch, something that would require the verifier to look at every transaction, it still requires the verifier to know a committee for each epoch. These committees can be quite large to ensure that the sampling of stake is representative. We optimize this need by introducing the Ad-Hoc Threshold Multi-Signatures (ATMS) primitive, in which a whole committee is represented by a single short public key. This public key, called the *aggregate public key*, encodes in it all the public keys of all committee members, and its size is comparable to a regular public key. There is one aggregate key per epoch. Furthermore, if multiple committee members create signatures on a message, these can be *aggregated* into a single *aggregate signature* which has a small size, comparable to a regular signature. These signature schemes are also *treshold signatures*: If a majority of the keys used to create the aggregate public key combine their signatures into an aggregate signature, the aggregate signature will pass verification with the corresponding public key in which more keys have been aggregated. On the other hand, any aggregation of signatures by a minority number of signatories will not verify. Lastly, the signature scheme is *ad hoc*, because each member of the committee can locally create their own signature without interactivity with the rest of the members. The signatures can then be aggregated by any party, even outside of the signatories themselves. Similarly, public keys can be generated locally without interactivity and aggregated later by any party.

The evolution of committees from epoch to epoch is then represented by asking the verifier to only hold a single aggregate key per epoch. For each transition from epoch to epoch, the majority of the committee of the previous epoch sign a certificate transitioning to the committee of the new epoch. The certificate's text contains the new aggregate public key. These signatures are combined into an aggregate signature and can be verified by the aggregate public key of the previous epoch. As such, the verifier simply sees one public key and one signature per epoch. The aggregate public key of each epoch can also vouch for events that took place during that epoch such as a particular transaction. For purposes of synchronization of a light client, the committee corresponding to the current epoch can vouch for any transactions occurring any time in the past, and the light verifier can use the current aggregate key to verify any such claims. The amount of data needed to download is therefore minuscule per epoch. However, as epochs evolve linearly with time, the asymptotic complexity is still $\Theta(|\mathsf{C}|)$, albeit with a very small constant.

1.3.5 Sidechains

Consensus compression is useful for superlight clients and superlight mining. One of its most interesting application beyond this pertains to cross-chain communication. Given two chains, the *source chain* and the *destination chain* which are maintained by independent mining populations, we want to have the destination chain react to an *event* that took place in the source chain. The miners on one chain are *isolated* from the other, and they do not connect to its network. However, users interested in the cross-chain events can connect to both networks. Because users can be adversarial, the destination chain miners cannot simply trust the users' claim that an event took place on the source chain – it needs to be verified.

Events that can be passed from blockchain to blockchain can be virtually any predicate pertaining to a small amount of data such as a constant number of transactions or blocks. In practice, these events can be, for example, about the fact that a transaction with particular metadata took place on the source blockchain, that a particular amount has been sent or received, that a particular account holds a certain amount of balance at some point in time, that some transaction output remains unspent, that a smart contract method was called with some particular arguments, or that a smart contract's state variables held certain values at some point in time.

We create support for cross-chain communication for blockchain systems in which the source blockchain can produce Non-Interactive Proofs of Proof-of-Work-or-Stake. As discussed, these systems may need some backwards compatible upgrades before they can be used as sources, such as introducing interlinks. On the receiving side, we require the blockchain to either have native support for consuming these Proofs of Proofs (and its miners to run a verifier), or to have smart contract support. Smart contract support is sufficient because a compressed consensus verifier (such as a NIPoPoW or ATMS verifier) can be written in the code of a smart contract.

The core idea of our sidechain construction is as follows. Whenever an event takes place within the source blockchain, information about the event as and its corresponding blockchain is compressed into a Proof of Proof. For proof-of-work sources, these are NIPoPoWs; for proof-of-stake sources, these are evolving ATMS. As the proof is a short string, it can be submitted to the destination blockchain for verification. This verification can be done by the destination chain miners natively (without ever connecting to the source blockchain), or by a smart contract. The latter case is the more interesting one, because it allows blockchains that were never designed to be interoperable to communicate. In this setting, the code for checking the validity of the NIPoPoW as well as the code for their comparison is written into smart contract format. This allows the smart contract running in one blockchain to consume NIPoPoWs generated about other blockchains. In this case, the miners of the smart blockchain simply execute the smart contract code as if it were any other smart contract, without any regard for its semantics or knowledge that inter-chain communication is taking place. Once the smart contract has verified the proof, it may be able to take a decision immediately (for proof-of-stake sources), or it may need to wait for a short period to allow for *contesting proofs* to appear (for proof-of-work sources). In case a fraudulent NIPoPoW pertaining to a shorter chain is submitted, a contesting proof allows any node monitoring both chains to make a counter-claim, via a new NIPoPoW, claiming that the original proof was fraudulent. The smart contract runs the NIPoPoW comparison algorithm and decides which of the two proofs is the legitimate one. To ensure users have incentives to submit such proofs, a successful contestation is accompanied by a reward which is paid out to the contester and is obtained by slashing a collateral deposited by the original prover. In case no proofs of fraudulence are submitted, the collateral is returned to the (honest) initial prover.

As we can have both proof-of-work and proof-of-stake sources as well as proof-of-work and proof-of-stake destinations, this construction allows us to create communication bridges between any combination of consensus mechanisms. If the information is passed only one-way, then the communication is unidirectional. This application is still useful as a mechanism to bootstrap a new cryptocurrency from an old one. We present one way of doing that by destroying money on the source blockchain and proving that this happened by providing a relevant proof submitted to the destination chain. On the other hand, nothing prevents two blockchains from both functioning as a source and a destination for each other. This allows us to build fully bidirectional communication channels between blockchains, giving rise to full two-way pegs.

In a two-way peg, the bidirectional communication can be used to move a coin from one blockchain to another while it retains its nature, decoupling the notion of a blockchain from that of a cryptocurrency. The lifecycle of the coin is as follows. Consider two blockchains A and B and an asset which is natively issued within chain A. Two smart contracts are deployed for interoperability purposes, one on chain A and one on chain B. Each of the two smart contracts records the address of its counterpart. Initially, a coin exists in its native blockchain A. The coin is locked into a special smart contract within the native blockchain, which ensures that it cannot be further spent. At this stage, blockchain A functions as a source blockchain for the communication protocol. The fact that the coin was locked is proven using a Proof of Proof-of-Work or Stake. This proof is created by the user that locked her coin. The proof is submitted to a smart contract which lives on blockchain B. At this stage, blockchain B functions as a destination chain. The receiving smart contract verifies the Proof of Proof-of-Work or Stake and waits to ensure there is no contestation. At this point, the receiving smart contract can be certain that the coin was locked into its counterpart which lives in chain A. Note that the smart contract never directly connected to the network of chain A. The receiving smart contract mints a new token coin within blockchain B, equal in nominal value to the value of the locked coin on chain A. The token on chain B is given the the public key that corresponds to the user who locked the initial coin on chain A. That token can then be used for exchange within chain B like a regular currency. However, it is dissimilar from the native currency of chain B (in fact, chain B may not even have a native currency of its own, as we will see).

When any user who ends up holding the token within chain B decided to move it back to chain A, the reverse process is initiated. Chain B now functions as a source chain, while chain A functions as a destination chain. The token is sent to the smart contract in chain B. The contract destroys the token. The user who destroyed their token in chain B then creates a Proof of Proof of this fact. Again this is a short string which can be submitted to the smart contract that lives on chain A. The smart contract on chain A verifies the proof attesting to the destruction of the token on chain B, and waits for potential contestation. At this point, the smart contract on chain A can be certain that the token has been destroyed on chain B and can now unlock the corresponding value in native currency that it currently holds. Because tokens appear on chain B only after they have been locked in the smart contract of chain A, the smart contract in chain A will always have sufficient balance to unlock to respond to any withdrawals.

Lastly, the process is voluntary and any users of A and B who do not wish to use it do not have to participate or even know about it. Importantly, the miners of the two chains can remain unaware that the cross-communication is taking place. Additionally, chain A is *firewalled* from chain B. This means that a catastrophic failure in chain B, such as an honest majority violation, does not propagate to chain A beyond the amount of money that was locked in the cross-chain smart contract. As such, any nodes who are not participating in the cross-chain protocol will incur no financial losses from such a catastrophic failure.

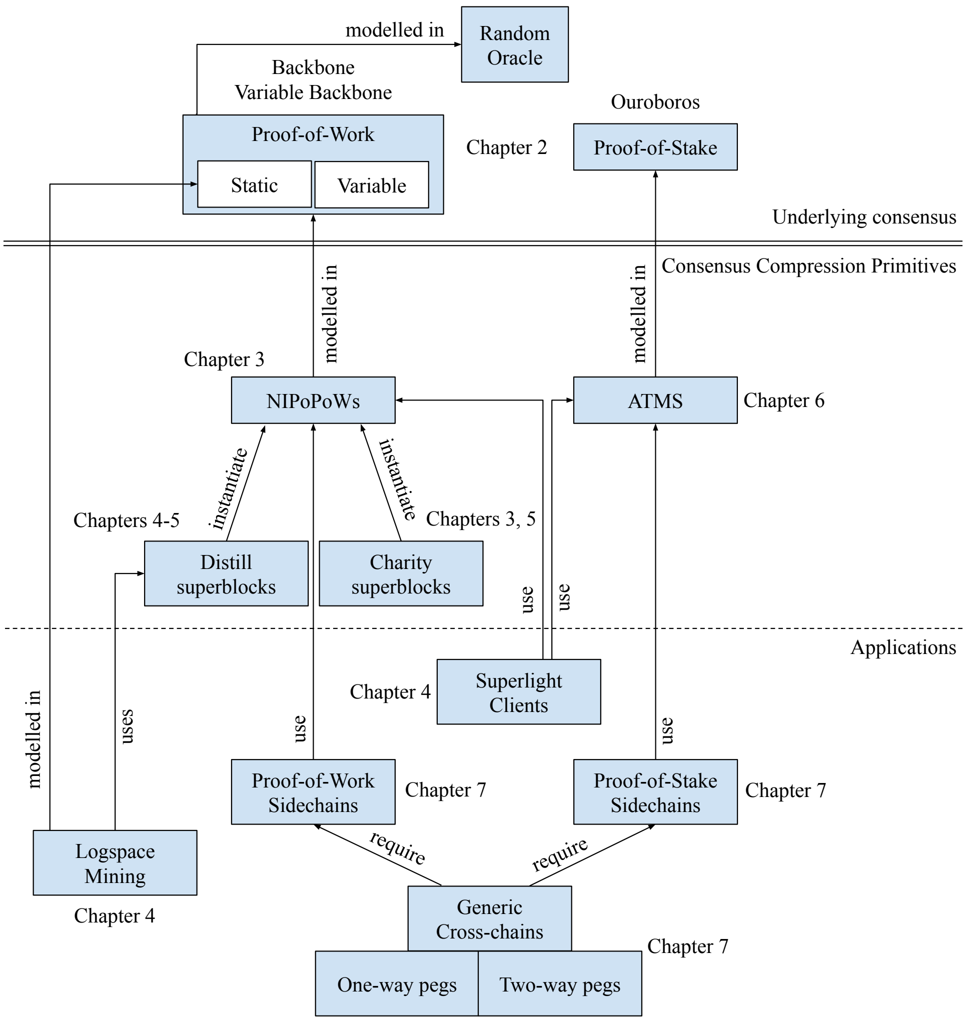
1.3.6 Summary of Contributions

A summary of our contributions and their dependencies, with annotations indicating where they are presented in this thesis, is visually illustrated in Figure 1.10.

In summary, in this thesis we solve the problem of *consensus compression* for all decentralized blockchain consensus mechanisms. **For proof-of-work**, we introduce the NIPoPoWs primitive (Chapter 3) and we give two superblock-based constructions of succinct NIPoPoWs protocols in the Backbone model: First the *charity* construction (Chapter 3), and second the *distill* construction (Chapter 4 and 5). In the static synchronous model (Chapter 3), we prove our charity construction with *goodness* secure against $\frac{1}{2}$ adversaries, but succinct only in the optimistic setting. Our charity construction *without goodness* as well as our distill construction are both secure and succinct against $\frac{1}{3}$ adversaries (Chapter 5). In the synchronous variable model (Chapter 5), our distill construction is secure against a $\frac{1}{3}$ adversary as long as difficulty is non-decreasing. Our charity without goodness construction is secure against a $\frac{1}{3}$ adversary even if difficulty is not limited to non-decreasing. Both are succinct as long as difficulty is not exponentially decreasing. Lastly, in the Δ-bounded delay setting (Chapter 5), both constructions are secure and succinct under the same limitations, but only against a $\frac{1}{4}$ adversary. We give concrete parameter recommendations and run experiments and simulations indicatively for the charity construction of Chapter 3. **For proof-of-stake**, we construct the ATMs primitive and give signature-based construction (Chapter 6). These are secure in the Ouroboros model, but offer only constant improvements over full clients and hence do not achieve asymptotic succinctness.

We make use of these primitives to build **cross-chain transfer** applications, which give rise to interoperability among blockchains, allowing generic information transfer among work/work, work/stake, and stake/stake chains. We give the definition of what constitutes a secure sidechain protocol (Chapter 7) and put forth a cross-chain protocol which we prove secure. Our protocols can work natively or by leveraging smart contract functionality. We show how they can be utilized to create one-way and two-way pegs and discuss several deployment mechanisms which allow them to be deployed as soft forks or better. Our protocols can also be used to build superlight clients. Lastly, we show that our proof-of-work protocols specifically can be utilized to build logarithmic-space miners (Chapter 4), providing exponential improvements over the state and communication complexity of existing proof-of-work blockchain protocols.

Figure 1.10: A roadmap of this thesis' structure. Our underlying model is shown above the double line. Our contributions are shown below the double line and comprise consensus compression primitives (above the dashed line) and their applications (below the dashed line). The respective chapters are indicated next to each topic.



1.4 Related Work

1.4.1 Consensus Compression

Nakamoto's original Bitcoin whitepaper [116] anticipated the rising costs of an evergrowing blockchain, and proposed a protocol for lightweight clients, called "Simplified Payment Verification" (SPV). Unlike "full nodes" which process and validate the entire blockchain (including all transactions and signatures), SPV clients only process the proof-of-work chain and transactions directly pertaining to them. SPV nodes only download the block headers (which in this thesis we denote C) of the blockchain and leave out transactions. An SPV node who has downloaded the chain with the most proof-of-work can validate the proof-of-work, as the headers are sufficient to do this. When an SPV client receives a payment, it requests a Merkle inclusion proof in order to confirm that the transaction is included in one of the blocks whose header is already stored by the client. This is possible because the block header contains a Merkle Tree Root of the transactions. This gives a significant improvement in communication complexity, the amount of data that an SPV node needs to download being $80|\mathsf{C}|$ for 80-byte headers in Bitcoin. While today a full node needs to download more than 250 GB of data, an SPV node only needs to download 50 MB of data. However, as the size of the block has a constant upper bound (currently about 1 MB), SPV constitutes a constant improvement over full nodes. In particular, the amount of data that needs to be downloaded from the network grows linearly in both the full node and the SPV node cases. They both download $\Theta(|\mathsf{C}|)$ data. Most widely used clients today, among others mobile wallets based on BitcoinJ, implement SPV.

As an alternative to downloading all block headers at first startup, the SPV client software could embed a hardcoded checkpoint; blocks prior to which are ignored.[2] Although this approach is very efficient, it introduces additional trust assumptions on software maintainers.

The first attempt to compress consensus state to something sublinear without additional trust assumptions was put forth in 2012 in a Bitcointalk forum post by Andrew Miller in which he proposes what he calls the *High-Value-Hash Highway* [115]. In that post, the idea of *superblocks* is introduced and intuition is given about superchains that capture proof-of-work without presenting it. Albeit incomplete, some first ideas are also discussed with regards to the interlink data structure. In 2016, Aggelos Kiayias, Nikolaos Lamprou and Aikaterini-Panagiota Stouka coined the term *Proofs of Proof-of-Work* and proposed a well-defined protocol leveraging superblocks in their paper *Proofs of Proofs of Work with Sublinear Complexity* [83]. The paper shows that such protocols can be succinct and presents a concrete protocol with complexity $\mathcal{O}(\log(|\mathsf{C}|))$. That paper also properly introduces the superblocks structure, which we make heavy use of in our work. Both the *highway* forum post and the *PoPoW* paper have been important inspiration for our work. These ideas form the basis for the creation of superlight clients.

The protocol defined in these works, the *KLS* protocol, has some significant shortcomings. First, KLS is *interactive.* This means that multiple network messages between the prover (a full node peer in the network) and the verifier (a superlight node or client attempting to synchronize) are required in order for the superlight node to arrive at a conclusion about which blockchain is the longest. In these

[2]See `https://bitcoinj.github.io/speeding-up-chain-sync`

interactions, the superlight node communicates with multiple provers, at least one of which is honest, and *interrogates* them by asking questions which are adapted based on the other proofs that they have received. After sufficient interrogation, the verifier can draw the correct conclusion. The number of rounds in this interrogation process can grow to be $\mathcal{O}(\log(|\mathsf{C}|))$. This has impact on the performance of the superlight client, but also limits the applicability of the scheme when it comes to applications such as cross-chain certification and logspace mining. In this thesis, our protocols improve upon theirs to achieve *non-interactivity* (see Chapter 3), enabling cross-chain and logspace mining applications in addition to superlight clients (see Chapter 4).

Secondly, while the KLS protocol allows a superlight client to deduce that most recent k blocks (where k denotes a configurable constant *common prefix parameter*) of a chain which would be admitted by an honest full node, it gives no further ability to query the chain for information buried deep within it. In particular, it does not offer the ability to prove that a transaction took place in the past, unless k is set to be large. In that case, if $k \in \Omega(|\mathsf{C}|)$, the protocol ceases to be succinct. Therefore, the main application of a superlight client, which involves verifying whether a transaction took place, is not possible by the KLS protocol, except in limited circumstances. As such, it constitutes a *suffix proof* protocol, but falls short of constructing *infix proofs*. In this thesis, we generalize their construction to allow for the deduction of a quite *generic* class of predicates about the chain, including old transaction confirmation (see Chapter 3). Intuitively, we extend their work to allow any fact about the blockchain which depends on a polylogarithmic number of blocks to be decided.

Lastly, the security treatment of the KLS construction is incorrect. More specifically, due to a subtle but critical mistake in the proof of the security theorem, their conclusion that their protocol achieves security for a 1/2 adversary with overwhelming probability is false. In fact, there exists a minority adversary which is able to break security with overwhelming probability. These issues are explored in this thesis in our development of *superchain quality*. We subsequently use this property to both build an attack which we prove works with overwhelming probability against their scheme, as well as to create a scheme which bypasses the issue for a 1/2 adversary. Their initial construction can be proven secure against a 1/3 adversary, albeit using a different proof strategy (leveraging our results in Chapter 5). We also remark that our schemes are succinct against all possible adversaries and are not limited to optimistic succinctness.

In addition to treating the above shortcomings, our results generalize in the more refined model of Δ-bounded delay and variable difficulty. We also provide experimental results, simulations, concrete security parameters, and smooth upgrade recommendations.

The practical feasibility of superblock-based constructions and their empirical analysis has been studied by a series of other works [75, 77].

Our work has inspired a vibrant and challenging line of research by other groups. Following our NIPoPoW paradigm and subchains approach, a different construction named FlyClient [32] allows the succinct and secure construction of proofs. Instead of including an interlink vector which just contains links to the most recent superblock of every level, they make use of Merkle Mountain Ranges to reference the whole previous blockchain from every block. This enables them to create a probabilistic proof which is made non-interactive using the Fiat–Shamir [55] heuristic.

The structure of their scheme is inspired by similar probabilistic challenge-response protocols that exist in the zero knowledge setting [134]. Their protocol is elegant and simple and has seen some adoption in practice. Additionally, they propose an extension to Merkle Mountain Ranges, namely Segment Merkle Mountain Ranges which are useful for variable difficulty constructions (a different approach to ours). While we are convinced their protocols are secure in principle, contrary to our work in this thesis, their security analysis so far is only done *in expectation* and not *with high probability.* A more complete analysis is pending. Lastly, because the construction of their NIPoPoWs is not superblock-based, but probabilistic, this means that these proofs are not *online*. Therefore, if a full node has a proof and mines a block on top of it, they cannot create a new proof without holding the whole chain. This limitation is inherent in their construction and means that applications such as logspace mining which rely on online NIPoPoWs are not possible using their protocol.

A different approach to our superblock-based constructions stems from leveraging SNARKs and their recursive composition [24, 25] as a way to compress to polylogarithmic size and update a given blockchain. One such construction is Coda [110]. This construction allows the compression of both proof-of-work and proof-of-stake consensus. A significant advantage and distinguishing feature of our approach is however the fact that it does not rely on a common reference string as SNARK-based protocols require, and so does not require a trusted setup. Moreover, no additional assumptions are introduced beyond those necessary for the security of the Bitcoin blockchain in the random oracle model. Lastly, we remark that our approach is simpler in its construction, as it makes use of only proof-of-work and hash primitives. We believe this is a great advantage for protocols that will be implemented in practice. Additionally, it enables a wider audience to audit the implementation and has a smaller attack surface. We note, however, that our proof-of-stake construction (in Chapter 6) is $\Omega(|\mathsf{C}|)$, while they achieve the exponentially better result of $\mathcal{O}(\log(|\mathsf{C}|))$. We only achieve a comparable result in the proof-of-work case. However, it is unclear whether the Coda construction is practical. Although the proofs are indeed succinct, the constant factors in their sizes have not been calculated. As with many zero knowledge schemes, they can very quickly turn out to be prohibitively large. The practicality of this scheme remains to be illustrated.

Our proof-of-stake compression scheme makes heavy use of the Ad-Hoc Threshold Multisignatures primitive, which we introduce in this work. Threshold multisignatures were considered before [122], but without the ad-hoc characteristic we consider here.

Our work focuses on compressing *consensus data*, i.e., the block headers exchanged and stored. There has been significant work in compressing *application data* in a way that maintains consensus. Such examples include moving transactions and smart contract execution off-chain in Layer 2 constructions such as payment channels [126, 86, 10, 11] and networks, rollups of the optimistic [142] or zero-knowledge [20] kind, and sidechains [125, 125, 90, 82]. Other systems allow (quite successfully) compressing multiple transactions into fewer or smaller, such as in the case of EDRAX [38], bulletproofs [31], or Mimblewimble [124]. These systems do not compress consensus state; all block headers must still be sent and stored, even though the actual *application data* is reduced.

1.4.2 Interoperability

Blockchain interoperability is a problem that has attracted a lot of interest in the space. Since 2011, the Namecoin [105] blockchain used the concept of *merged mining* in which proof-of-work is performed simultaneously in two different chains. While this does not allow the exchange of blockchain application data (such as transactions), it constitutes the first instance of a chain co-existing with another in a symbiotic relationship.

Proposals to allow blockchains to exchange application layer data were first discussed in 2014 in the work *Enabling Blockchain Innovations with Pegged Sidechains* [12]. This also constitutes the first occurrence of the term *sidechain*. While they do not propose a concrete mechanism which allows for decentralized blockchain communication, they describe the need for interoperability among blockchains. An interesting observation in the work appears in Appendix B in which the authors discuss the need for secure compact Proofs of Proof-of-Work (which they call *compact DMMS*) as a prerequisite for the construction of secure sidechains. The need for a Proofs of Proof-of-Work construction as a prerequisite for sidechains was pointed out earlier that year by Friedenbach in the Bitcoin Development mailing list [57]. Both of these mentions refer to superblock-based constructions. These references highlight that our work in this thesis solves a long-standing open problem with a multitude of previously known applications. A significant contribution of their work is the introduction of the *firewall* property, which we formalize in this thesis.

These works explore one particular application of sidechains, the ability of a sidechain to offer smooth software upgrades. They propose that features can be trialed on a sidechain prior to being adopted in the main chain. Their definition of a *sidechain* is limited to being a *child* chain of a particular parent chain; as such, the sidechain's lifecycle fully depends on that of its parent. In our work, in addition to providing a concrete implementation of cross-chain proofs which are needed for parent-child-style sidechains, we generalize the notion of sidechains to allow pre-existing and stand alone blockchains to communicate as well (see Chapter 7). Additionally, their exchange of information is limited to the two-way pegged transfer of value. Our construction allows conveying *any* information, not just transfers of value. This distinguishing property becomes more pronounced when applied to smart contract blockchains in which more generic information of interest can appear. In our work, a blockchain is a sidechain of another chain if it can generically react to events on that chain, and so the relationship can be symmetric. The events which we allow reacting to can be any of the generic predicates that lend themselves to succinct proofs-of-work or stake.

The first practical implementation of cross-chain information transfer involves the use of *atomic swaps*. Atomic swaps allow the exchange of two assets between a party on one chain and a counter-party on another chain atomically. Atomicity is achieved through Hash Time Locked Contracts [126] and ensures that either both assets are exchanged, or neither. This makes atomic swaps useful for decentralized cross-chain trading and makes decentralized exchanges (DEXes) possible. Atomic swaps were first introduced by Tier Nolan [118]. Multi-asset atomic swaps were studied by Herlihy [68]. Contrary to atomic swaps, our cross-chain communication protocols allow the generic transfer of information and are not limited to asset swapping. Additionally, we do not require a counter-party to trade with.

One construction which allows the generic transfer of information between chains without additional trust assumptions is called *BTCRelay* [41]. The premise is to copy all block headers of one chain into the other using a smart contract. While secure, this structure is not succinct (it is $\Theta(|\mathsf{C}|)$). It has been implemented as a smart contract in production which allows the transfer of information from Bitcoin to Ethereum.

Drivechains [141, 102] and *rootstock* are sidechain proposals which require miners of both chains to be aware of both networks. This follows our definition of *direct observation* (Chapter 7). While challenging from an engineering point of view, mutual direct observation makes the problem theoretically trivial. In practice, mutual direct observation means that deploying these sidechain schemes on stand-alone chains would require modifying their miners and full nodes to connect to multiple networks, which would be unfeasible. In our scheme, miners remain agnostic to the existence of other chains and connect only to one network.

Plasma [125] (of which Plasma Cash [93] is one instance) creates a child sidechain which is under the control of a centralized custodian account. The custodian is responsible for creating blocks on the sidechain. The custodian commits his blocks once every epoch to the parent chain, which implements a decentralized consensus mechanism. While the custodian is a centralized entitity, it is not trusted with security, as the construction allows the participants who hold money on the child sidechain to *abandon ship* in case foul play is detected. Foul play includes any violation of *common prefix* on the child sidechain; this means that participants must wait for the child chain to commit to the parent chain in order to trust that their transaction has been confirmed. Foul play also includes any *violation of liveness* in which the custodian fails to create blocks. In the latter case, the parent chain allows the withdrawal of money from the stale child chain. One shortcoming of the scheme is that the commitments between the child chain and the parent chain grow linear in time. However, we believe this scheme is particularly interesting and practical, as linear cross-communication will always be necessary between chains that have active cross-chain activity. While this mechanism cannot connect stand-alone decentralized blockchains, it can be used to offload some of the burden from an overloaded parent chain, assisting in scalability.

Polkadot [152], *Tendermint*, *Cosmos* [30], *Liquid* [47] and *Interledger* [144] also build cross-chain transfers. Their validation relies on a trusted committees, federations or is left unspecified. None of the aforementioned constructions include proofs of security. Other related work includes XCLAIM [157], PeaceRelay, COMIT [70], NOCUST [81] and Dogethereum [143]. In all cases, there is a lack of a formal security model and analysis, a gap we aim to fill here.

Chapter 2

Background

The cryptographic treatment has three core characteristics [103].

1. **Formal definitions** play a central role. They specify the desirable properties of our protocols. As we will see, these can often be quite tricky to develop. One such example is the definition of *pegging security* in Chapter 7.
2. **Clearly articulated assumptions** allow us to understand the limitations of our protocols. Our protocols never work unconditionally, and we must restrict our model to obtain security. One such example is the computing power of the adversary. In the case of Chapter 3, we can withstand a 1/3 adversary, but the extended model of Chapter 5 can only withstand a 1/4 adversary.
3. **Rigorous proofs of security** give us the *guarantee* that our protocols are secure, as long as our assumptions hold. Instead of employing *ad hoc* arguments, the proofs are mathematical theorems employing computational reductions, and they assert that the protocols are secure *for all* adversaries.

This chapter gives an overview of prerequisites upon which we build our protocols. Blockchain science is a new field. As such, many of the elements we employ here are folklore knowledge in the community, and some of them have never been written down precisely before. Thus, this chapter may be of independent interest as reference. Two interesting examples are our own security proof for Merkle Trees, as well as an explicit description of the static difficulty, variable difficulty, synchronous, and Δ-bounded delay environments in the form of pseudocode, which has previously only appeared in imprecise textual descriptions.

2.1 Notation

We use standard mathematical notation throughout this work. We define all the non-standard or unusual notation in this section. We use standard cryptographic notation, which is also introduced here for reference. The reader unfamiliar with this notation can consult some reference book in the subject such as [103] for a more complete treatment.

Given a distribution $\mathcal{M}$, we denote by $m \leftarrow \mathcal{M}$ the experiment by which the random variable m is chosen according to the distribution $\mathcal{M}$. Given a finite set M, we use $\mathcal{U}(M)$ to denote the uniform distribution which assigns probability $1/2^{|M|}$ to each element $m \in M$. We will use $m \overset{\$}{\leftarrow} M$ to denote the experiment in which m is sampled from $\mathcal{U}(M)$.

As a shorthand for probabilities and to avoid excessive subscripting, we will write the experimental set up (such as the sampling of random variables) within the $\Pr[\cdot]$ prior to the predicate of interest and separated by ; . For example, $\Pr[x \leftarrow \mathcal{D}_1, y \leftarrow \mathcal{D}_2; x + y = 1]$ denotes the experiment of independently sampling two random variables, x and y, from the distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively, summing their values, and observing whether their sum is equal to 1.

2.1.1 Asymptotic probabilistic security

Following the extended Church–Turing definition, we consider the class of problems in P to be *easy*, and we call *hard* those which are not easy [137]. We will talk about *honest parties*, Turing Machines [147] which run our code, and the *adversary*, denoted $\mathcal{A}$, which is an arbitrary Turing Machine that can run any code. It is assumed that all honest parties and the adversary have polynomial available time. Both the honest parties and the adversary have access to true randomness and are thus probabilistic Turing Machines. The shorthand *PPT* is used to denote a probabilistic polynomial-time Turing machine.

Our theorems are by computational reduction, in which we show that bad events happen only with negligible probability in some security parameter. We use $\lambda \in \mathbb{N}$ to denote this security parameter. This parameter allows all our cryptographic primitives to be instantiated with the required level of security; for example, it provides the number of bits in the output of our hash function.

Definition 1 (Negligible)**.** *A function $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ is* negligible *if for all $k \in \mathbb{N}$ it holds that $f \in \mathcal{O}(\frac{1}{\lambda^k})$.*

We will use the notation negl to denote any negligible function.

Definition 2 (Overwhelming)**.** *A function $f : \mathbb{N} \longrightarrow \mathbb{R}^+$ is* overwhelming *if it can be written as $f(n) = 1 - \mathsf{negl}(n)$ for some negligible function negl.*

We will define the *security* of various cryptographic protocols by making use of challenger-adversary games in which the challenger is a known Turing Machine defined by us, but the adversary is an *arbitrary* Turing Machine. Herein lies the beauty of cryptography as a science; it allows us to create protocols which we prove secure against *any* adversary, even those we cannot conceive. A protocol will be considered *secure* if no PPT adversary can win the respective game, except with negligible probability.

2.1.2 Sequences

We use $[n]$ to denote the set of natural numbers from 0 up to and including n. We also use $[\mathcal{M}]$ to denote the support of a distribution $\mathcal{M}$; the distinction between the two notations will be clear from the context. We use ϵ to denote the empty sequence (or empty string). We will later also use $\epsilon \in \mathbb{R}^+$ to denote deviations from expectation, but it will be clear from the context whether this is a string or a real number. We write one sequence next to another to denote string concatenation. Likewise, we concatenate sequences to symbols by juxtaposition. If for clarity this needs to be made explicit, we will use the symbol $\|$ for concatenation. We make the implicit assumption that concatenation encodes values into a string unambiguously, for example by separating them by a special character that never appears in the operands. This allows them to be uniquely extracted again.

Our sequences are indexed starting at 0. Given a sequence C, we use $|\mathsf{C}|$ to denote its length. We use Python notation to denote sequence addressing. For $i \in [n-1]$, we denote the i^{th} element from the beginning as $\mathsf{C}[i]$. The first element of the sequence is thus $\mathsf{C}[0]$. For $i \in [n] \setminus \{0\}$, we denote the i^{th} element from the end as $\mathsf{C}[-i]$. The last element of the sequence is thus $\mathsf{C}[-1]$. We call this the *tip* of the sequence. Given $i \in \mathbb{Z}, j \in \mathbb{Z}$ with $i \leq j$, we denote $\mathsf{C}[i{:}j]$ the subsequence from i (inclusive) to j (exclusive), that is the sequence which contains exactly the elements $\mathsf{C}[i], \mathsf{C}[i+1], \ldots, \mathsf{C}[j-1]$. If $i > j$, then by convention we set $\mathsf{C}[i{:}j] = \epsilon$. We allow this *range* notation to be used with negative indices as well, indicating ranges starting or ending (or both) in indices considered from the end of the sequence, hence allowing for $\mathsf{C}[-i{:}j], \mathsf{C}[i{:}-j]$, and $\mathsf{C}[-i{:}-j]$. The left end of a range can omitted if it is $i = 0$. The right end of a range can be omitted if it is $j = |\mathsf{C}|$. For example, $\mathsf{C}[:-k]$ is the sequence C with the last k elements excluded. In this example, if $|\mathsf{C}| < k$, then $\mathsf{C}[:-k] = \epsilon$. Given $A, Z \in \mathsf{C}$ such that A and Z exist only once in C, we denote by $\mathsf{C}\{A{:}Z\}$ the subsequence of C starting from A (inclusive) and ending in Z (exclusive). If $A = \mathsf{C}[0]$, it can be omitted. Omitting Z denotes the sequence starting with A and containing all subsequent elements until the end of the sequence. We use the notation $A \preceq B$ to indicate that sequence A is a prefix of B, namely that $B[:|A|] = A$.

We will reuse some set notation when talking about sequences[1]. Given a sequence A, we write $x \in A$ to mean $\exists s_1, s_2 : A = s_1 \| x \| s_2$. We write $A \subseteq B$ to mean that A is a *subsequence* of B, that is it has all the same elements in the same order, but it could have more intertwined. More precisely, $\epsilon \subseteq B$ is true; and if $A \subseteq B$ is true, then $x \| A \subseteq x \| B$ as well as $A \subseteq x \| B$ are both true. We will use set builder notation to filter sequences. Namely, $\{x \in A : p(x)\}$ is interpreted as ϵ if $A = \epsilon$; otherwise, if A is non-empty and we have $A = uB$ for some element u and sequence B, then $\{x \in A : p(x)\} = p(u) \| \{x \in B : p(x)\}$. We use $A \cap B$ to mean the sequence $\{x \in A : x \in B\}$. Because in all of its uses throughout this work A and B will have their shared elements appear in the same order, this operator will be symmetric in our case.

2.2 Mathematical Background

The following well-known theorem is due to Rubin [40, 39]. It will help us derive negligible bounds for probabilities of bad events.

[1]Formally speaking, in the traditional build-up of mathematical foundations using axiomatic set theory, sequences are defined as sets, and this would be deemed somewhat abusive in light of the following fact. If A is a sequence and x is a candidate element, $x \in A$ could have a different truth value when A is interpreted as set and when A is interpreted as a sequence. For this reason, the symbols $\in, \subseteq$ and so on, when referring to sequences (as is made clear in the surrounding text), are meant to be read as predicates $\in', \subseteq'$ and so on different from their set-theoretic counterparts. For conciseness, and because it does not affect clarity in our treatment, we will not mark them with the prime symbol.

Theorem 1 (Chernoff Bounds)**.** *Let $\{X_i : i \in [n]\}$ be mutually independent Boolean random variables such that for all $i \in [n]$ it holds that $\Pr[X_i = 1] = p$. Let $X = \sum_{i=1}^{n} X_i$ and $\mu = \mathbb{E}[X] = pn$. Then, for all $\delta \in (0, 1]$ it holds that:*

$$\Pr[X \leq (1-\delta)\mu] \leq \exp(-\frac{\delta^2\mu}{2}) \text{ and } \Pr[X \geq (1+\delta)\mu] \leq \exp(-\frac{\delta^2\mu}{3})$$

A similar tail bound can be stated [109, Theorem 3.15] for martingales.

Theorem 2 (Martingale Tail)**.** *Let $X_0, X_1, \dots$ be a martingale with respect to the sequence $Y_0, Y_1, \dots$. For $n \geq 0$, let*

$$V = \sum_{i=1}^{n} \operatorname{var}(X_i - X_{i-1} | Y_0, \dots, Y_{i-1}) \text{ and } b = \max_{1 \leq i \leq n} \sup(X_i - X_{i-1} | Y_0, \dots, Y_{i-1}),$$

where sup is taken over all possible assignments to $Y_0, \dots, Y_{i-1}$. Then, for any $t, v \geq 0$,

$$\Pr\big[(X_n \geq X_0 + t) \wedge (V \leq v)\big] \leq \exp\Big\{-\frac{t^2}{2v + 2bt/3}\Big\}.$$

2.3 Cryptographic Primitives

We now overview the cryptographic primitives we will make use of. In particular, cryptographically secure hash functions, public-key signatures, and proof-of-work. This section is a review. For a full treatment, refer to any introductory textbook in the subject [103, 80, 62, 63].

2.3.1 Hash Functions

A hash function $H^s : \mathcal{M} \longrightarrow \{0,1\}^\lambda$ is a function parameterized by the security parameter λ which takes any string from the distribution of input strings $\mathcal{M}$ and outputs a string of constant size λ. To capture the fact that the hash function behaves like a randomly chosen function, the hash function is instantiated using a key-generating function $\mathsf{Gen}(1^\lambda)$ which generates a hash key s. The hash function itself is then H^s, a different function for each value of the key s. As hash functions are the building blocks and workhoses of cryptography, other protocols are designed on top of them that make use of them. We will do so in this work. In practice, the key s is assumed to have been generated by the designers of the higher level protocol that makes use of the hash function and is typically fixed and publicly known. The hash protocol is the tuple $\Pi = (\mathsf{Gen}, H)$.

Practical hash functions allow us to map any message x of arbitrary length $x \in \{0,1\}^*$ to a fixed-length bitstring $\{0,1\}^\lambda$. Hash functions are easy to compute, but hard to invert. In applications, it is assumed that a hash uniquely represents its preimage (it is *binding*) and that the preimage cannot be discovered from the image given sufficient entropy (it is *hiding*). This makes them ideal for constructing *commitment schemes.*

These intuitive ideas are captured by the difficulty of finding collisions in hash functions. This is formalized in the next definition.

Algorithm 1 The collision-finding experiment $\mathsf{hash\text{-}collision}_{\Pi,\mathcal{A}}$.

```
function $\mathsf{hash\text{-}collision}_{\Pi,\mathcal{A}}(\lambda)$
    $s \leftarrow \mathsf{Gen}(1^\lambda)$
    $x_1, x_2 \leftarrow \mathcal{A}(s)$
    if $H^s(x_1) = H^s(x_2) \wedge x_1 \neq x_2$ then
        return true
    end if
    return false
end function
```

Algorithm 2 The preimage-finding experiment $\mathsf{hash\text{-}preimage}_{\Pi,\mathcal{A}}$.

```
function $\mathsf{hash\text{-}preimage}_{\Pi,\mathcal{A}}(\lambda)$
    $s \leftarrow \mathsf{Gen}(1^\lambda)$
    $x \leftarrow \mathcal{M}$
    $x' \leftarrow \mathcal{A}(s, H^s(x))$
    if $H^s(x) = H^s(x')$ then
        return true
    end if
    return false
end function
```

Definition 3 (Collision resistance)**.** *A hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^\lambda$ is called* collision resistant *if for all PPT adversaries $\mathcal{A}$ there is a negligible function* negl *such that*

$$\Pr[\mathit{hash\text{-}collision}_{\Pi,\mathcal{A}} = 1] \leq \mathsf{negl}(\lambda)\,.$$

A weaker notion of security mandates that no adversary can reverse the function (pre-image resistance) or that no adversary can find a second value giving the same output as a given random value. The two cryptographic games and definitions are illustrated in Algorithms 2 and 3.

Definition 4 (Pre-image resistance)**.** *A hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^\lambda$ is called* pre-image resistant *if for all PPT adversaries $\mathcal{A}$ there is a negligible function* negl *such that*

$$\Pr[\mathit{hash\text{-}preimage}_{\Pi,\mathcal{A}} = 1] \leq \mathsf{negl}(\lambda)\,.$$

Definition 5 (Second pre-image resistance)**.** *A hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^\lambda$ is called* second pre-image resistant *if for all PPT adversaries $\mathcal{A}$ there is a negligible function* negl *such that*

$$\Pr[\mathit{hash\text{-}second\text{-}preimage}_{\Pi,\mathcal{A}} = 1] \leq \mathsf{negl}(\lambda)\,.$$

Algorithm 3 The second-preimage-finding experiment $\mathsf{hash\text{-}second\text{-}preimage}_{\Pi,\mathcal{A}}$.

```
1: function $\mathsf{hash\text{-}second\text{-}preimage}_{\Pi,\mathcal{A}}(\lambda)$
2:     $s \leftarrow \mathsf{Gen}(1^\lambda)$
3:     $x \leftarrow \mathcal{M}$
4:     $x' \leftarrow \mathcal{A}(s, x)$
5:     if $H^s(x) = H^s(x') \wedge x \neq x'$ then
6:         return true
7:     end if
8:     return false
9: end function
```

A hash function that is collision-resistant is also pre-image resistant; additionally, if it is pre-image resistant, then it must also be second pre-image resistant, as long as it provides sufficient compression [131].

Protocols deployed in practice make use of fixed hash functions; that is, hash functions with a fixed security parameter and a fixed key. In Bitcoin, the hash function `SHA256` [119] is used for both commitments and proof-of-work. Its domain and range are `SHA256` $: \{0,1\}^* \longrightarrow \{0,1\}^{256}$. In Ethereum, the hash function `keccak256` [23], a variant of `SHA3`, is used for commitments. Its domain and range are `keccak256` $: \{0,1\}^* \longrightarrow \{0,1\}^{256}$. The function used for proof-of-work is a variant of this.

2.3.2 Signatures

A *digital signature* allows parties to authenticate the origin of a message as well as its integrity [103]. If Alice signs a message m, she generates a signature σ which is uniquely associated with that message. That signature can then only be used to verify that particular message. In a secure signature scheme, an adversary cannot *forge* signatures that correctly verify for messages that have not been signed by the honest party.

Signing and verification are two separate tasks which are asymmetric. Only the authorized party can sign a message, but anyone can verify the signature. This is achieved by having each party generate their own *public-private key pair* (pk, sk) in which pk is the public key and sk is the secret key. Signatures are then generated using the secret (or signing) key sk and verified using the public (or verification) key sk. A key pair is generated using the polynomial-time key generation algorithm $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$. A signature is generated by invoking the polynomial-time signing algorithm $\sigma = \mathsf{Sig}(sk, m)$. Verification is done by checking whether the verification algorithm $\mathsf{Ver}(pk, m, \sigma)$ returns true or false. The signature scheme Π then is defined as the tuple $\Pi = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$.

Signature schemes must be *correct*.

Definition 6 (Signature correctness)**.** *A signature scheme is* correct *if there is a negligible function* negl *such that for all messages* $m \in \{0,1\}^*$ *it holds that*

$$\Pr[(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda); \mathsf{Ver}(pk, m, \mathsf{Sig}(sk, m)) = \mathit{false}] < \mathsf{negl}(\lambda)\,.$$

Algorithm 4 The forgery experiment $\mathsf{sig\text{-}forge}_{\Pi,\mathcal{A}}$.

```
function $\mathsf{sig\text{-}forge}_{\Pi,\mathcal{A}}(\lambda)$
    $(pk, sk) \leftarrow \mathsf{Gen}(1^\lambda)$
    $M \leftarrow \emptyset$
    function $\mathsf{SigO}(m)$
        $M \leftarrow M \cup \{m\}$
        return $\mathsf{Sig}_{sk}(m)$
    end function
    $(m, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sig}O(\cdot)}(pk)$
    if $Ver(pk, m, \sigma) \land m \notin M$ then
        return true
    end if
    return false
end function
```

A *secure* signature scheme requires that no adversary is able to forge signatures. This is captured in the game-based definition of Algorithm 4. In this game, the challenger first generates a public/private key pair by invoking $\mathsf{Gen}(1^\lambda)$. Subsequently, the challenger asks the adversary $\mathcal{A}$ to attempt to find a signature forgery. The adversary is allowed to ask the challenger to have any messages signed by invoking the $\mathsf{Sig}O$ oracle with messages of her choice. The adversary is allowed to make multiple adaptive queries to the oracle. When the adversary is ready, she presents a message m, which she must not have requested from the oracle $\mathsf{Sig}O$ and a signature σ. The adversary is successful if the signature verifies.

Definition 7 (Security)**.** *A signature scheme* $\Pi = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ *is* secure *if for all PPT adversaries* $\mathcal{A}$ *there is a negligible function* negl *such that*

$$\Pr[\text{sig-forge}_{\Pi,\mathcal{A}}(\lambda)] < \mathsf{negl}(\lambda)\,.$$

There are multiple ways to construct a secure signature scheme. Our signature schemes of interest make use of the *discrete logarithm* problem in a group. In such a construction, a cyclic group $\mathbb{G}$ with order close to 2^λ and a generator $G \in \mathbb{G}$ are fixed initially. A public key corresponds to an element $A \in G$ of the group, while the corresponding private key $a \in \mathbb{Z}_{|\mathbb{G}|}$ is the order of A with respect to G, that is $A = aG$. The keys are generated by first choosing a private key a uniformly at random and then computing its corresponding public key. The public key can be computed quickly from the private key using multiplication by doubling [135], but it is believed that the inverse problem is hard.

The problem of finding a from A is made formal in Algorithm 5. Here, we assume that an efficient algorithm $\mathcal{G}$ can be used to pick a suitable group of the appropriate order and output its description. Furthermore, we assume the group operator is efficiently computable. The challenger generates a group and chooses one of its elements at random. The adversary is then asked to find the *discrete logarithm* of that element.

Algorithm 5 The DLOG problem.

```
1: function $\mathsf{DLOG}_{\mathcal{G},\mathcal{A}}(\lambda)$
2:     $(\mathbb{G}, G, |\mathbb{G}|) \leftarrow \mathcal{G}(1^\lambda)$
3:     $a \xleftarrow{\$} \mathbb{Z}_{|\mathbb{G}|}$
4:     $A \leftarrow aG$
5:     $a^* \leftarrow \mathcal{A}(\mathbb{G}, G, |\mathbb{G}|, A)$
6:     return $a^*G = A$
7: end function
```

Definition 8 (Discrete Logarithm Problem)**.** *The* discrete logarithm problem *is hard in the family of groups* $\{\mathcal{G}(1^\lambda)\}_{\lambda\in\mathbb{N}}$ *if for all PPT adversaries* $\mathcal{A}$ *there is a negligible function* negl *such that*

$$\Pr[\mathit{DLOG}_{\mathcal{G},\mathcal{A}}(\lambda) = 1] < \mathsf{negl}(\lambda)\,.$$

The particular instantiation of signature schemes in the context of cryptocurrencies makes use of elliptic curves [91] in which the discrete logarithm problem is believed to be hard. More specifically, Bitcoin and Ethereum both use the `secp256k1` curve [128].

Remark 1. *We remark that, perhaps contrary to popular belief, blockchain protocols do not depend at all on encryption primitives. Therefore, we choose not to treat encryption at all in the present work.*

2.4 Authenticated Data Structures

2.4.1 Merkle Trees

Consider a set $S = \{s_1, s_2, \cdots, s_n\}$ of strings $s_i \in \{0,1\}^*$. At some initial time, this set is compressed into a *root* string s which is short ($|s| = \lambda$). This compressed string is produced honestly and is given to a party called the *verifier*. Given this short trusted root string, the verifier receives claims from untrusted *provers* which claim that a certain piece of data e existed in S. The verifier's job is to decide whether such claims are truthful or fraudulent.

This protocol is an *authenticated set*. It consists of four algorithms $\mathcal{G}$, compress, prove, verify. At the beginning of the execution, $\mathcal{G}(1^\lambda)$ is invoked to initialize the protocol parameters. These parameters can be shared among multiple invocations of the protocol. As these parameters are fixed by the protocol in its concrete implementations, we will make them implicit from now on. A set S is compressed by invoking compress(S) which produces the root s. When an honest prover wishes to prove that some element e exists in S, they produce an *inclusion proof* π = prove(S, e). When the verifier receives an element e together with an inclusion proof π, they check its veracity by invoking verify(π, e, s), which returns true or false.

Authenticated set protocols must be correct. This means that honest executions should always work.

Algorithm 6 The authenticated data structure challenger.

```
1: function $\mathsf{AUTH}_{\Pi,\mathcal{A}}(\lambda)$
2:     $(S, e, \pi) \leftarrow \mathcal{A}(1^\lambda)$
3:     return $e \notin S \wedge \text{verify}(\text{compress}(S), e, \pi)$
4: end function
```

Definition 9 (Correctness)**.** *Consider an authenticated set protocol* $\Pi = (\text{compress}, \text{prove}, \text{verify})$*. We say that* Π *is* correct *if*

$$\forall S : \forall e \in S : \text{verify}(\text{prove}(S, e), e, \text{compress}(S))\,.$$

Such protocols are useful when s and π are short.

Definition 10 (Succinctness)**.** *Consider an authenticated set protocol* $\Pi = (\text{compress}, \text{prove}, \text{verify})$*. We say that* Π *is* succinct *if for all* S *it holds that*

$$|\text{compress}(S)| \in \mathcal{O}(\mathit{polylog}(|S|)) \wedge \forall e \in S : |\text{prove}(S, e)| \in \mathcal{O}(\mathit{polylog}(|S|))\,.$$

In the protocols we will explore, we will have $|s| = \lambda \in \mathcal{O}(1)$ and $\pi \in \mathcal{O}(\log(|S|))$. Furthermore, $|S|$ will be polynomial in λ.

An authenticated set protocol is *secure* if no adversary can convince a verifier about the inclusion of an element which is not in the set. This is made formal in the game illustrated in Algorithm 6.

Definition 11 (Security)**.** *An authenticated set protocol* $\Pi = (\text{compress}, \text{prove}, \text{verify})$ *is* secure *if for all PPT adversaries* $\mathcal{A}$ *there is a negligible function* negl *such that*

$$\Pr[\mathit{AUTH}_{\Pi,\mathcal{A}}(\lambda)] < \mathit{negl}(\lambda)\,.$$

A construction that solves this problem which is used extensively in blockchain protocols is the *Merkle Tree* [112]. This construction is illustrated in Algorithm 7. It is parameterized by a hash function H. The construction presented works for $|S|$ equal to a power of 2.

It treats S as a sequence and organizes it into a complete binary tree Z using the $\mathsf{heapify}$ routine (for reference, see Figure 1.3 in Chapter 1). The routine places the hashes of the elements of S on the leaves of Z by storing them at locations $Z[|S|:]$. The value of each internal node is the hash of the concatenation of the values of its children. The $\mathsf{compress}$ function returns the value of the root which resides at $Z[1]$. To create a proof π, the prove routine takes an index of an element i and finds its position in the binary tree, namely the leaf stored at $Z[|S| + i]$. It then traverses the path from that leaf up to the root, maintaining the index of the current node in the variable i. In every iteration, it includes a bit indicating whether the current node is a left child ($b = 0$) or a right child ($b = 1$). For each node, it includes the value $Z[i \oplus 1]$ of the node's sibling, which has index $i \oplus 1$. To verify a proof, the verifier successively hashes the element whose inclusion is proven with the hashes h of the siblings provided in the proof π on the correct side indicated by b. In the end, it checks whether it has arrived at the trusted root s and this determines the result of the verification.

Correctness is achieved because the hash function is deterministic and verify applies hashes in the same manner as heapify does. Succinctness follows because $|s| = \lambda$ and, furthermore, the tree contains $2|S| - 1$ elements so the height of the tree is $\Theta(\log(|S|))$, making $|\pi| \in \Theta(\log(|S|))$.

The construction prefixes the value of each tree node with an indicator string marking it as a hash ("h:"). On the contrary, each of the elements of S is marked as content by prefixing it with a different indicator ("c:") prior to compressing. The verifier then begins by marking the claimed value as content by prefixing it with a "c:", in which case it can be certain there will be no confusion with a hash. This ensures that the adversary cannot claim inclusion of internal nodes as leafs.

Remark 2 (Length of a Merkle Tree)**.** *Instead of marking every node as a* hash *or* content *node, the* compress *function can also be made to return the count* $|S|$ *in addition to* s*. In that case, the verifier first asserts that* $|\pi| = \log|S|$ *before proceeding with verification. The compressed string has length* $\Theta(\log\log|S|)$*, but each proof is smaller by a constant factor. While this simplifies the security proof, most blockchain protocols require that* $|s| \in \Theta(1)$ *and so we adopt this formulation here.*

Security follows by a direct computational reduction from the collision resistance of H. We give a novel version of this proof here[2].

[2]The proof that appears in Modern Cryptography [103] shows something weaker: that a Merkle Tree is a collision resistant hash function. Dowling et al.[50] show in a different style of proof that collision resistance of the root implies path consistency on the leaves.

Algorithm 7 The Merkle Tree construction for $|S| = 2^k$ for some k.

```
function heapify$^H(S)$
    for $i \leftarrow 0$ to $|S| - 1$ do                    ▷ Fill in the tree leaves
        $Z[|S| + i] \leftarrow$ "h:" $\|$ $H($"c:" $\|$ $S[i])$
    end for
    for $i \leftarrow |S| - 1$ down to 1 do                 ▷ Fill in the tree internal nodes
        $Z[i] \leftarrow$ "h:" $\|$ $H(Z[2i] \| Z[2i + 1])$
    end for
    return $Z$
end function
function compress$^H(S)$
    return heapify$^H(S)[1]$
end function
function prove$^H(S, i)$
    $Z \leftarrow$ heapify$^H(S)$
    $i \leftarrow |S| + i$
    $\pi \leftarrow [\,]$
    while $i > 1$ do
        $b \leftarrow i \bmod 2$
        $\pi \leftarrow \pi \| (b, Z[i \oplus 1])$
        $i \leftarrow \lfloor i/2 \rfloor$
    end while
    return $\pi$
end function
function verify$^H(\pi, e, s)$
    $e \leftarrow$ "h:" $\|$ $H($"c:" $\|$ $e)$
    for $(b, h) \in \pi$ do
        if $b$ then
            $e \leftarrow$ "h:" $\|$ $H(e \| h)$
        else
            $e \leftarrow$ "h:" $\|$ $H(h \| e)$
        end if
    end for
    return $e = s$
end function
```

Theorem 3 (Security). *Let H be a collision resistant hash function. The Merkle Tree construction of Algorithm 7 parameterized by H is a secure authenticated set protocol for sets of size $|S| = 2^k$.*

Proof. Consider an arbitrary adversary $\mathcal{A}$ against AUTH. We construct a collision resistance adversary $\mathcal{A}^*$ for the hash function H. The adversary $\mathcal{A}^*$ invokes $\mathcal{A}$ and obtains a proof π, an element e and a set S. The adversary $\mathcal{A}^*$ checks that this proof is fraudulent by ensuring that it passes verify and that $e \notin S$ (if not, then $\mathcal{A}^*$ aborts). This proof implicitly encodes a position i in the tree, namely the position expressed by the binary number obtained by concatenating all the bits b in π.

First consider the case $|\pi| = \log |S|$. The adversary $\mathcal{A}^*$ checks whether $H(e) = H(S_i)$. If so, it returns the pair (e, S_i) as a collision. Otherwise it proceeds to find a collision as follows.

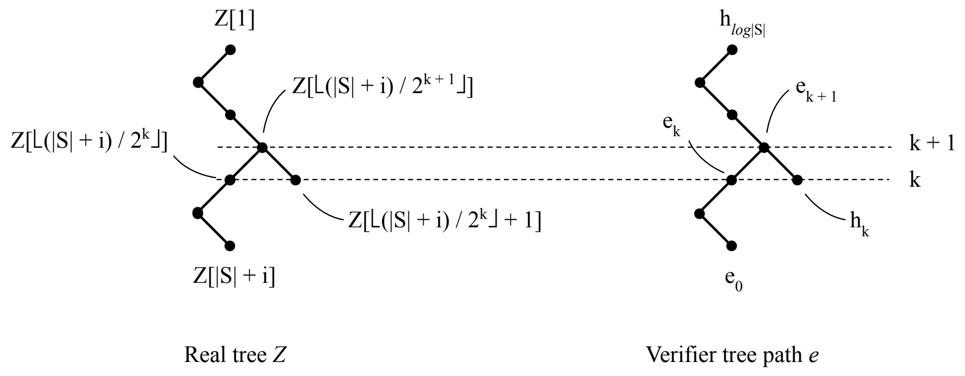


Figure 2.1: The path comparison between the real tree (left) Z constructed using heapify and the verifier tree path (right) h constructed by taking successive hashes with the sequence s_i finds a collision at level k.

The verifier who applies successive hashing will arrive at a sequence of hashes $e_0, e_1, \cdots, e_{\log|S|}$. The adversary $\mathcal{A}^*$ evaluates this sequence in the same manner as the verifier would and also looks at the values $Z[|S|+i], Z[\lfloor\frac{|S|+1}{2}\rfloor], \cdots, Z[1]$ as obtained by heapify(S). The adversary $\mathcal{A}^*$ compares the two sequences and finds the minimum $k \geq 0$ such that $Z[\lfloor\frac{|S|+i}{2^k}\rfloor] \neq e_k$, but $Z[\lfloor\frac{|S|+i}{2^{k+1}}\rfloor] = e_{k+1}$. This is illustrated in Figure 2.1. If $b_k = 0$, then it returns the pair $(Z[\lfloor\frac{|S|+i}{2^k}\rfloor] \,\|\, Z[\lfloor\frac{|S|+i}{2^k}\rfloor + 1], e_k \,\|\, h_k)$ as a collision. If $b_k = 1$ it returns the pair $(Z[\lfloor\frac{|S|+i}{2^k}\rfloor - 1] \,\|\, Z[\lfloor\frac{|S|+i}{2^k}\rfloor], h_k \,\|\, e_k)$. If no such k is found, it aborts.

We argue that $\Pr[\mathsf{hash\text{-}collision}_{H,\mathcal{A}^*}] \geq \Pr[\mathsf{auth}_{\Pi,\mathcal{A}}]$. To see this, consider the event that $\mathcal{A}$ is successful. It suffices to show that $\mathcal{A}^*$ is also successful. We distinguish two cases. In the first case, we have $H(e) = H(S_i)$. As $e \neq S_i$, therefore $\mathcal{A}^*$ has found a collision and the condition holds. Otherwise we have that $H(e) \neq H(S_i)$. As the verification is successful, we must have $h_{\log|S|} = s$. Therefore there must exist some k at which the condition holds. Without loss of generality, let $b_k = 0$. As $Z[\lfloor\frac{|S|+i}{2^k}\rfloor] \neq e_k$ we have that $Z[\lfloor\frac{|S|+i}{2^k}\rfloor] \,\|\, h_k \neq e_k \,\|\, h_k$. Additionally, $Z[\lfloor\frac{|S|+1}{2^{k+1}}\rfloor] = e_{k+1} = \text{“h:”} \,\|\, H(\lfloor Z[\frac{|S|+i}{2^k}\rfloor] \,\|\, Z[\lfloor\frac{|S|+i}{2^k}\rfloor + 1]) = \text{“h:”} \,\|\, H(e_k \,\|\, h_k)$, so a collision has occurred.

For the case $|\pi| < \log|S|$, the adversary $\mathcal{A}^*$ finds the internal node of the tree Z which lies at a distance $|\pi|$ from the root and its index at this level is again constructed from the binary number obtained by concatenating the bits b in π. Since it is the internal node of a complete tree, it has two children a and b and its value will be “h:” $\|\, H(a \,\|\, b)$ where a is also prefixed by “h:”. On the other hand, the claimed e is prefixed by “c:”, and so $e \neq a \,\|\, b$. The adversary $\mathcal{A}^*$ then proceeds to find the minimum k as before, but this time starting at a distance only $|\pi|$ from the root instead of $\log|S|$.

Lastly, for the case $|\pi| > \log|S|$, the adversary $\mathcal{A}^*$ applies $|\pi| - \log|S| - 1$ successive hashes with the sequence $s_0, s_1, \cdots, s_{|\pi|-\log|S|-1}$ on the sides $b_0, b_1, \cdots, b_{|\pi|-\log|S|-1}$ in the same manner that the verifier would by consuming the first $|\pi| - \log|S| - 1$ elements of π. At that point, it arrives at some value v which is prefixed by “h:”. It then computes i as before by using the remaining bits b in the proof that has not yet been consumed, and considers the element S_i for which it will hold that $v \neq S_i$ since it is prefixed by “c:”. It can then proceed as before to find k. □

Once the marking of nodes as leaves or internal nodes has been established, the limitation that $|S| = 2^k$ can also be relaxed by working with an incomplete binary tree instead of a complete binary tree.

Due to the way the Merkle Tree construction works, it can also be used to work with *authenticated sequences* in which positions are significant. To make this explicit, the verify function can be extended to also accept a positional argument i. Correctness and succinctness are defined as before. The security definition must then be altered to mandate that no polynomial adversary can produce arguments S, e, i, π such that $\mathsf{verify}(S, e, i, \pi) \wedge S[i] \neq e$, except with negligible probability. The construction and proof of security remain the same.

2.4.2 Sparse Merkle Trees

Merkle Trees allow *inclusion proofs*, but not *exclusion proofs*, i.e, proving that an element e is *not* in the S that was used to construct the tree root s. We can extend the authenticated set protocol to support such exclusion proofs. To do this, we allow the data structure to store key-value pairs. This generalizes the previous construction, as keys can be taken to be the indices within the sequence S. As $|S|$ is polynomial in λ, we can treat this new authenticated data structure as a means to encode a partial function $f : \{0,1\}^{p(\lambda)} \Rightarrow \{0,1\}^*$ for some fixed polynomial p. We will only concern ourselves with such functions defined only in a polynomial number of inputs, $|f| = p(\lambda)$. We call this primitive an *authenticated dictionary.*

The new protocol consists firstly of compress which takes some function f and returns a succinct root s. Secondly, of a function prove which takes a key k and a function f and produces a proof π that $f(k)$ is the value of f for k, or that $f(k) = \perp$. Lastly, verify takes a root s, a proof π, a key k and a value v (which can be $\perp$) and returns true or false indicating whether the proof is correct. Correctness, succinctness are defined as before. Security is also similar, keeping in mind that no adversary should be able to prove that the function is defined where it is not, or that the function is undefined where it is. Exclusion proofs for some key k are constructed by showing that $f(k) = \perp$.

Sparse Merkle Trees, introduced by Laurie and Kasper [99], are an authenticated dictionary construction.

The prover construction is illustrated in Algorithm 8. The heapify algorithm generates a Merkle Tree with $2^{p(\lambda)}$ leaves. At the leaf of location k, it places the value $H(\text{“c:”} \| f(k))$ if f is defined at k; otherwise, it places the value $H(\epsilon)$. This is done to distinguish the function f being defined and having the empty string as its value versus being undefined. We force keys to have a length of exactly $p(\lambda)$, so internal nodes are represented using shorter keys. Internal nodes are evaluated as before, noting that the internal node with key k has two children with keys $k0$ and $k1$. As the size of the tree is now fixed, there is no need to prefix internal nodes with “h:”. The root is the internal node keyed with the empty string ϵ, and so compress returns $Z[\epsilon]$.

Algorithm 8 The prover of the Sparse Merkle Tree construction for dictionaries of polynomial size $p(\lambda)$.

```
function $\mathsf{heapify}^H(f)$
    $Q \leftarrow [\,]$
    for $(k, v) \in f$ do
        $Z[k] \leftarrow H(\text{“c:”} \,\|\, v)$
        $Q \leftarrow Q \,\|\, k$                          ▷ Queue key for upward propagation
    end for
    $N \leftarrow [H(\epsilon)]$
    for $i \leftarrow 1$ to $p(\lambda)$ do      ▷ Compute hashes of empty subtree with height $i$
        $N[i] \leftarrow H(N[i-1] \,\|\, N[i-1])$
    end for
    while $Q \neq [\,]$ do
        $k \leftarrow Q[0]$
        $Q \leftarrow Q[1{:}]$
        if $Z[k \oplus 1] = \perp$ then
            $Z[k \oplus 1] \leftarrow N[p(\lambda) - |k|]$
        end if
        $k \leftarrow k[{:}-1]$                             ▷ Consume the last bit of the key
        $Z[k] \leftarrow H(Z[k0] \,\|\, Z[k1])$
        if $k \neq \epsilon$ then
            $Q \leftarrow Q \,\|\, k$
        end if
    end while
    return $Z$
end function
function $\mathsf{compress}^H(f)$
    return $\mathsf{heapify}^H(f)[\epsilon]$
end function
function $\mathsf{prove}^H(f, k)$
    $Z \leftarrow \mathsf{heapify}^H(S)$
    $\pi \leftarrow [\,]$
    while $k \neq \epsilon$ do
        $\pi \leftarrow \pi \,\|\, Z[k \oplus 1]$
        $k \leftarrow k[{:}-1]$
    end while
    return $\pi$
end function
```

This tree has an exponential number of nodes. The trick [44] to make this evaluation possible in polynomial time is to observe that f is only defined in a polynomial number $p(\lambda)$ of values and hence most of the leaves will have a value of $H(\epsilon)$. The root of any subtree of the same size whose leaves are all empty will have the same hash value, and so these can be precomputed. These are evaluated and cached in the array $N[i]$ which stores the hash of the empty subtree with height i. The computation is polynomial because, for each non-empty leaf, the number of nodes that need to be filled in is at most $p(\lambda)$, and the number of non-empty leaves is $p(\lambda)$, bounding the number of nodes that need to be filled in by $p^2(\lambda)$. The precomputed values N are used to fill in any direct children of those nodes which correspond to empty subtrees of a particular height.

Algorithm 9 The verifier of the Sparse Merkle Tree construction for dictionaries of polynomial size $p(\lambda)$.

```
function verify^H(s, π, k, v)
    if $|\pi| \neq k$ then
        return false
    end if
    if $v = \perp$ then
        $v \leftarrow \epsilon$
    else
        $v \leftarrow$ "c:" $\| v$
    end if
    $e \leftarrow H(v)$
    for $h \in \pi$ do
        if $k \bmod 2 = 0$ then
            $e \leftarrow H(e) \| h$
        else
            $e \leftarrow h \| H(e)$
        end if
        $k \leftarrow k[: -1]$
    end for
    return $e = s$
end function
```

The verifier is illustrated in Algorithm 9. It is identical to the simple Merkle Tree verifier, with the exception that it can now look at the bits of the key k to obtain the path it must follow from the leaf to the root.

Correctness follows because the evaluation is deterministic. For succinctness, note that proofs π have size $\Theta(\log(p(\lambda)))$. Lastly, security is similar to standard Merkle Trees and reduces from the collision resistance of H. One aspect that makes the security proof easier is that the size of Sparse Merkle Trees is known to be $2^{p(\lambda)}$, which means that the verifier can always check that $|\pi| = k = p(\lambda)$ prior to verification. As such, no cases need to be taken for $|\pi| \neq k$. Lastly, for the detail of the special value $\perp$, we note that any fraudulent claim of inclusion for points where the function is undefined or vice versa will cause a collision because "c:" $\| v \neq \epsilon$ for any v.

Sparse Merkle Tree proof sizes are $|k|\lambda$ where $|k|$ denotes the size of the key and $|\lambda|$ denotes the output of the hash function H. They can be optimized by leaving out hashes N corresponding to empty subtree siblings, which can be computed locally by the verifier, bringing the proof size down to $|k| + \lambda \log n$.

Sparse Merkle Trees support a wide range of useful operations, among others succinctly proving that an *assignment* has been made [38]. The authenticated dictionary primitive can include a prove-assign(f_1, k, v) which produces a proof π and a ver-assign(s_1, s_2, π, k, v) method which returns a boolean. Given a trusted root s_1 that the verifier knows about which corresponds to some function f_1, the prover can prove that a new root s_2 corresponds to the function f_2 which is identical to f_1, except with some key k assigned to value v (the value v can be $\perp$ if f_2 is now undefined at point k). Correctness and security are defined as expected. To implement proofs of assignment, the prover shows that it has replaced the log-sized path corresponding to k with the correct values by presenting all sibling hashes.

Remark 3 (Merkle–Patrcia Tries)**.** Merkle–Patricia Tries *are Sparse Merkle Trees which apply path compression akin to Patricia Tries [151]. However, despite increased implementation complexity, they do not achieve any improvement in performance that cannot be achieved in standard Sparse Merkle Trees. Proofs in Patricia Tries are* $|k| + \lambda \log n$ *bits long (and* $\log n$ *is* $\mathcal{O}(|k|)$*), but, as discussed above, this same size can also be achieved in Sparse Merkle Trees with the appropriate optimizations.*

2.4.3 Merkle Mountain Ranges

It is useful to use authenticated sequences as an append-only log. In that case, it becomes necessary to succinctly prove to verifiers who only hold a root that the underlying sequence has only been appended to and not altered.

Authenticated sequences can be augmented with an additional *extend* operation. The primitive is extended by two additional methods, prove-extend and ver-extend. The method prove-extend(S_1, U) takes two sequences S_1, U and produces a proof π. The method ver-extend(s_1, s_2, π, U) takes two roots s_1 and s_2 corresponding to sequences S_1 and $S_2 = S_1 \parallel U$ respectively, a proof π, and a sequence U and returns a boolean. Given a verifier who has a trusted root s_1 which corresponds to a sequence S_1, the prover wants to prove to the verifier that a new root s_2 corresponds to the sequence S_2. Succinctness here mandates that $|\pi| \in \mathcal{O}(|U| + polylog(|S_1|))$ and correctness and security as defined as expected.

An additional variant allows equipping authenticated sequences with a *prefix* operation. This is similar to *extend*, but the sequence U is unknown to the verifier. More specifically, the primitive is extended by two methods, prove-prefix and ver-pref. The method prove-prefix(S_1, S_2) takes two sequences S_1 and S_2 and produces a proof π. The method ver-pref(s_1, s_2, π) takes two roots s_1 and s_2 corresponding to sequences S_1 and S_2 respectively and returns a boolean. Here, the prover claims that there *is* some sequence U which extends the previous sequence S_1, but the verifier is indifferent as to what it is, beyond the fact that s_1 corresponds to a prefix of the sequence that s_2 corresponds to. Note that in this case, U can be $\mathcal{O}(\Theta)$, but succinctness requires that $|\pi| \in \mathcal{O}(polylog(S_2))$, and so U cannot be made part of the proof.

These two extensions to authenticated sets can be achieved by Merkle Mountain Ranges, which were introduced by Laurie, Langley and Kasper in the context of Certificate Transparency [100] and named by Todd [146].

The Merkle Mountain Range construction is illustrated in Algorithm 10. The compress algorithm makes use of the compress-MT function of a standard Merkle Tree. It takes as input a sequence S to compress. In each step, it takes the longest prefix of S which is a power of 2 and calculates its Merkle Tree Root using the standard Merkle Tree compression function compress-MT which is called a *peak*. It does this with successively smaller prefixes until the whole sequence is consumed. It then collects all of these peaks by concatenating them together and hashies them to produce the final root. The number of peaks produced will be equal to the number of ones in the bit representation of the number $|S|$ and so are $\mathcal{O}(\log |S|)$.

Algorithm 10 The Merkle Mountain Range construction.

```
function $\mathsf{peaks}^H(S)$
    $v \leftarrow \epsilon$
    while $S \neq \epsilon$ do
        $p \leftarrow 2^{\lfloor \log |S| \rfloor}$
        $L \leftarrow S[:p]$
        $S \leftarrow S[p:]$
        $v \leftarrow v \,\|\, \mathsf{compress\text{-}MT}^H(L)$
    end while
    return $v$
end function
function $\mathsf{compress}^H(S)$
    return $H(\mathsf{peaks}^H(S))$
end function
```

It is possible to implement prove and verify akin to Merkle Trees. The implementations are almost identical, with the only difference being that, at the level of the peaks, all $\mathcal{O}(\log |S|)$ peaks must be included in π to be able to arrive at the final root. The construction remains succinct.

To prove that a particular element e has been added to the Merkle Mountain Range, it is sufficient to only present the peaks to the verifier, a logarithmically sized proof. This allows the verifier to produce the new Merkle Mountain Range root as well as the new peaks. Therefore, to show extension by a sequence U, prove-extend suffices to send the peaks to the verifier and allow the verifier calculate the new peaks and root for every element in U in order.

On the other hand, to show prefix, it suffices to show the peaks as well as a logarithmic number of elements corresponding to U. More specifically, U is consumed in successively larger portions, each used to match one of the outstanding peaks of S. For each portion of U, only the root of the corresponding Merkle Tree that matches the peak in S needs to be sent in the proof. Once U has been consumed sufficiently to match all the peaks of S (for which $\mathcal{O}(\log |S|)$ hashes are needed), the remaining elements of U are consumed in powers of two (for which $\mathcal{O}(\log |U|)$ hashes are needed). Hence, prove-prefix is also succinct.

2.5 Model

2.5.1 The Random Oracle

Real protocols are instantiated using real hash functions. However, as concrete mathematical objects, these are not possible to analyze cryptographically, as they do not have a security parameter. For example, `SHA256` is a function with a fixed size of 256 bits. In these terms, one cannot talk about negligible probability of failure. Keyed hash functions in which the function depends on the security parameter are possible to analyze in this manner, but their use is limited in practice. Additionally, many of the guarantees provided by keyed hash functions discussed above are insufficient for more elaborate protocols. In particular, collision and preimage resistance are not enough for our needs. Nevertheless, our intuition is that practical concrete hash functions behave nicely, and we wish to include this notion in our model.

To bypass these limitations, we model our hash functions in the Random Oracle model [19]. In this model, the hash function behaves like an ideal random function. This is helpful, because we can argue all of its outputs will be unbiased and independent. This abstraction also signifies that we are interested in working in the realm of *protocol design*, and the intricacies of practical hash function design and implementation, which stands in the realm of efficient symmetric cryptography, are beyond the scope of our work.

In the Random Oracle model, we assume the existence of a global oracle machine H to which every party, adversarial or honest, has access to. The machine models the hash function and hence allows parties to ask for its evaluation at any input. The Random Oracle machine receives any input in $\{0,1\}^*$ and returns an output in $\{0,1\}^\kappa$.

Algorithm 11 The Random Oracle model parameterized by security parameter κ.

```
1: $T \leftarrow \emptyset$
2: function $\mathsf{H}_\kappa(x)$
3:     if $x \notin T$ then
4:         $T[x] \xleftarrow{\$} \{0,1\}^\kappa$
5:     end if
6:     return $T[x]$
7: end function
```

The output is chosen as illustrated in Algorithm 11. In detail, the Random Oracle is parameterized by the security parameter κ. Upon receiving some input x, if the input has not been encountered before, then it produces a fresh uniformly random κ-bit string which it stores in a dictionary T and returns. If on the other hand it receives an input it has seen before, it answers consistently with its previous answer, giving the same answer for the same query by consulting the dictionary T. Hence, this functionality is stateful.

It is imperative that the instance of the machine that all parties communicate with is the same. Hence, if a party makes a particular query x to the oracle and then another party asks the same query, the answer will be the same. If the random oracle is modelled as a stateful functionality, communication between the parties and the Random Oracle machine can be modelled as Interactive Turing Machines communicating.

Alterantively but equivalently, the Random Oracle can be defined as a shared oracle which answers queries according to a function selected uniformly at random at the beginning of the execution. As a random function can neither be sampled in polynomial time nor represented in polynomial space, this latter formulation means that, when the oracle is treated that way, it cannot be modelled as an Interactive Turing Machine, but must remain an oracle. In this formulation, at the beginning of the execution, the random oracle H is sampled uniformly at random from the function space $(\{0,1\}^\kappa)^{\{0,1\}^*}$, and the adversary and honest parties are invoked with oracle access to this same H. As all parties will be polynomial in our treatment, this sampling can also be understood to be done uniformly at random from the function space $(\{0,1\}^\kappa)^{\{0,1\}^{p(\kappa)}}$ where $p(\kappa)$ denotes the polynomial bounding the total execution of all parties together.

An important feature of the Random Oracle model is that the adversary cannot compute H locally, but must invoke the oracle to do so. This means that, in our mathematical treatment, we are allowed to speak of the queries the adversary has made, count them, look at their responses, and so on. This ability, termed *random oracle observability* will be critical in our analyses. On the other hand, we will not make use of the ability of a computational reduction to modify the outputs of the random oracle at will, termed *random oracle programmability*, which is at the heart of many security proofs in cryptography. Our results are therefore stronger, as we only assume a *non-programmable* Random Oracle [117, 56], which is a weaker assumption than usually made.

We note two features of the Random Oracle: First, it is *collision-resistant.* Concretely, if a polynomial number of queries is made to the Random Oracle, then the probability that a collision will occur is negligible. Secondly, an adversary can *predict* the output of the Random Oracle prior to making a new query to it with negligible probability.

2.5.2 The Environment

We will begin with a simple model and make it successively more nuanced until it is sufficiently sophisticated to satisfactorily capture the real world. First, we describe the simple environment in which the network is *synchronous* and the execution is with *static difficulty.* We then relax the synchrony assumption by introducing a Δ-bounded delay network. Subsequently, we relax the static assumption by introducing executions of *variable difficulty* in which populations can be adversarially evolved. Last, we give the adversary even more power by allowing her to adaptively corrupt parties of her choice. We now introduce these models in order.

In our setting, we will study executions of protocols in which some parties are *honest* while others are *adversarial.* All the honest parties typically run the same code termed *the honest protocol* (which is the protocol we will design), while the adversary can run any code she wishes, but is bounded by polynomial time bounds. Both the honest parties and the adversary are probabilistic Turing Machines. As we are working in distributed settings, our protocols will be long-lived and involve multiple parties running simultaneously and communicating over the network while maintaining local state. Among the parties in our execution, we will denote by n the total number of parties and with t the number of parties that are adversarial. To strengthen our adversary, we assume all the adversarial parties *collude* and are controlled by a single adversary. The situation where multiple adversaries are not colluding is also captured by our stronger model (this can be captured by a single adversary which simulates the multiple non-colluding adversaries).

To model the distributed setting, we must speak of executions concretely. Towards this purpose, we conjure an *environment* $\mathcal{Z}$ which is an Interactive Turing Machine [148] (ITMs) and is responsible for orchestrating the whole execution. An Interactive Turing Machine is a Turing Machine which models interactive computation by employing additional input and output tapes that can be written to by external machines. The machine can decide to pause computation by entering a special state and its computation is resumed by writing to its input tape.

The environment spawns $n - t$ honest parties running the honest protocol Π as n different Interactive Turing Machines. The environment also spawns one adversarial Interactive Turing Machine $\mathcal{A}$. The honest parties and the adversary can pass messages to the environment and receive messages from it by writing and/or reading from their interactive tapes. The environment takes as input the security parameter 1^λ and functions as an operating system scheduler to activate the honest parties and the adversary according to some schedule. The environment halts after polynomial time. The Interactive Turing Machine model is equivalent to having the environment faithfully simulate the execution of the honest parties and the adversary and correctly maintaining their state across pausing and resumption.

We study an execution by observing its transcript (the messages exchanged by the parties) as well as the internal state of the parties throughout the execution. This transcript, which we will denote $\text{view}^{n,t}_{\Pi,\mathcal{A}}$, is a random variable which is a function of the coins of the probabilistic Interactive Turing Machines that form the execution, namely the environment itself, the adversary, the honest parties, and the Random Oracle. We remark that this treatment is similar to the setting of Universal Composability [36]. Despite the similarities on the surface, we do not fully employ it and neither are our protocols composable, nor are our security proofs simulation-based. On the contrary, we use a direct property-based approach in our proofs instead of employing universally composable functionalities.

A skeleton for the environment is illustrated in Algorithm 12.

At the beginning of the execution, all the ITMs are booted by invoking their constructors. The environment spins up $n - t$ honest machines that run the protocol Π and one adversarial machine that runs the protocol $\mathcal{A}$ and represents the t adversarial parties. These machines are stateful, and so we denote the respective ITM (which can be paused and resumed) by P_i for the honest parties (running protocol Π) and by A for the adversary (running protocol $\mathcal{A}$). The machines are given time polynomial in λ by invoking their constructors with the parameter 1^λ. During construction, the adversary learns of the number of honest parties $n - t$ and adversarial parties t. Importantly, the honest parties do not have this privilege. We call this setting *permissionless setting* (also known as the *anonymous byzantine* or *open setting*), because honest parties are not informed of each others' identities nor their count. Contrary to our treatment throughout this thesis, there also exists a *permissioned* [17] setting in the blockchain literature. In that setting, the n nodes are given authenticated channels between each other and the quantity n is known to all parties. We will not make this assumption here. The fact that we are working in the permissionless setting gives rise to the *decentralized* title of this thesis.

Time is quantized into discrete *rounds* [60] (or *slots* [89]) numbered $r = 1, 2, 3, \cdots$. The environment contains a main loop which executes one iteration per round r for a total polynomial number of rounds $p(\lambda)$. During every round, it first activates every honest party P_i by invoking its `execute` method. Subsequently, at the end of the round, it actives the adversary A. The fact that the adversary is activated at the end of every round is an advantage for the adversary. We call such an adversary a *rushing adversary*, because it can use its computational power for the round after it has observed what the honest parties have done during the same round. Both the honest party and the adversary can know the index of the current round by counting how many times they have been activated so far in their persistent state. Because both the honest parties and the adversary are PPT machines, they will

Algorithm 12 The environment and network model running for a polynomial number of rounds $p(\lambda)$.

```
function $\mathcal{Z}^{n,t}_{\Pi,\mathcal{A}}(1^\lambda)$
    for $i \leftarrow 1$ to $n - t$ do                          ▷ Boot honest ITMs
        $P_i \leftarrow \mathsf{new}\ \Pi(1^\lambda)$
    end for
    $A \leftarrow \mathsf{new}\ \mathcal{A}(1^\lambda, n, t)$              ▷ Boot adversarial ITM
    $\overline{C} \leftarrow [\,]$
    for $i \leftarrow 1$ to $n - t$ do
        $\overline{C}[i] \leftarrow [\,]$
    end for
    for $r \leftarrow 1$ to $p(\lambda)$ do
        $C \leftarrow \emptyset$
        for $i \leftarrow 1$ to $n - t$ do
            $C \leftarrow C \cup \{P_i.\mathsf{execute}(\overline{C}[i])\}$   ▷ Execute honest party $i$ for round $r$
        end for
        $\overline{C} \leftarrow A.\mathsf{execute}(C)$              ▷ Execute rushing adversary for round $r$
        for $c \in C$ do                        ▷ Ensure all parties will receive message $c$
            for $i \leftarrow 1$ to $n - t$ do
                $\mathsf{assert}(c \in \overline{C}[i])$
            end for
        end for
    end for
end function
```

run for polynomial time every time they are activated. Additionally, we assume n is polynomial, thus ensuring the total execution time is polynomial.

2.5.3 The Network

When an honest party is activated, it is given messages from the network to *read*, which are written to a special location within its input tape by the environment. Here, we denote the network messages received by party i as $\overline{C}[i]$ and pass them as input to the $\mathsf{execute}$ method. The party can then *write* messages to the network during the round, which we denote by the $\mathsf{execute}$ method returning a value. We say that such messages are *diffused* to the network and we will use the $\mathsf{Diffuse}$ notation within the implementation of honest protocols to signify that a message needs to be diffused to the rest of the parties. At the end of the round, the adversary can see all the messages C that have been diffused by the honest parties during the same round. The adversary can then decide what will appear in the network tape of every honest party at the beginning of the next round by outputting an array $\overline{C}$ that contains a list of messages $\overline{C}[i]$ for every party i. The adversary can reorder the messages and insert as many of her own as she wishes. That is, it is possible that $\overline{C}[i]$ will contain more messages than $\overline{C}$ and that the messages in $\overline{C}[i]$ will appear in a different order than in $\overline{C}$. As such, communication is not authenticated.

However, she must ensure that all messages diffused by any honest party during the previous round appear in the network tape of every other honest party at the beginning of the next round. This is ensured by the assertion in Line 17. This means that no messages can be dropped by the adversary. This *connectivity* assumption is equivalent to assuming that each honest party is not *eclipsed* [67, 153] from the rest of the network. In practice, this is achieved by ensuring that every honest party is connected to every other honest party through some path, although not necessarily directly. Practical peer-to-peer protocols use gossiping [65] to ensure messages reach every participant of the network. The network model abstracts out such details and treats a round as the unit of time which is needed for a message to reach from every honest party to every other honest party.

Crucially, because the adversary can reorder messages and inject as many additional messages as she pleases, she is a *sybil adversary* [49]. This means that the adversary can fake multiple identities and pretend to produce messages by multiple parties, potentially more than t. It will be the job of our honest protocol to produce a *sybil resilient* mechanism in which such attacks have no impact on the protocol's security. Furthermore, the adversary can *split* the view of the honest parties because she can communicate different messages to different honest parties and have $C[i] \neq C[j]$ for $i \neq j$ on the same round. For example, the order in which messages are delivered can be different for every honest party and the adversary may inject different messages of her own for every honest party.

The requirement that messages diffused at the end of one round are delivered at the beginning of the next is the *synchronous model*. A large part of our analysis will be made there.

In addition to the details specified in the environment of Algorithm 12, we allow the honest parties to receive auxiliary *input*, distinguished from the network tape. This input is adversarially chosen and can influence the decisions of the honest parties. Once these notions have been defined, such input will correspond to *transactions* that the honest parties will wish to include in their blockchains.

A relaxation of the synchronous model is the Δ-bounded delay model and is illustrated in Algorithm 13. In this model, the adversary may delay messages up to Δ rounds before finally delivering them. Any message diffused by any honest party at round r must appear in the network tapes of all other honest parties prior to round $r + \Delta$. This Δ is unknown to the honest parties, although the security of the protocol requires that Δ, together with other protocol parameters, satisfies a number of conditions. As such, this model stands between the *synchronous* setting (where Δ is known by all honest parties beforehand or, equivalently, $\Delta = 1$) and the *semi-synchronous* setting (where Δ is completely unknown) in that, while Δ itself is unknown, it is governed by equations which express a trade-off between Δ and other quantities, and these equations are known.

Chapters 3, 6, 4, 7 are explored in the synchronous model. We extend our protocol to the Δ-bounded delay model in Chapter 5.

2.5.4 Evolving Population

So far, we have defined the environment for executions in which n and t are fixed throughout the execution. We call this setting the *static difficulty* [60, 58] setting. The model can be relaxed to allow the population to evolve with time. This gives rise to the *variable difficulty* [61] model. Instead of two fixed values n and t, we instead consider two *sequences* of values $\{n_r\}_{r \in [p(\lambda)]}$ and $\{t_r\}_{r \in [p(\lambda)]}$. At round

Algorithm 13 The environment and network model in the Δ-bounded delay setting.

```
function $\mathcal{Z}^{n,t}_{\Pi,\mathcal{A}}(1^\lambda)$
    for $i \leftarrow 1$ to $n - t$ do                          ▷ Boot honest ITMs
        $P_i \leftarrow \mathsf{new}\ \Pi(1^\lambda)$
    end for
    $A \leftarrow \mathsf{new}\ \mathcal{A}(1^\lambda, n, t)$            ▷ Boot adversarial ITM
    $\overline{C} \leftarrow [\,]$
    for $i \leftarrow 1$ to $n - t$ do
        $\overline{C}[i] \leftarrow [\,]$
    end for
    $\mathsf{seen} \leftarrow [\,]$
    $\mathsf{diffused} \leftarrow [\,]$
    for $r \leftarrow 1$ to $p(\lambda)$ do
        $C \leftarrow \emptyset$
        for $i \leftarrow 1$ to $n - t$ do
            $\mathsf{seen}[i] \leftarrow \mathsf{seen}[i] \cup \overline{C}[i]$
            $C \leftarrow C \cup \{P_i.\mathsf{execute}(\overline{C}[i])\}$     ▷ Execute honest party $i$ for round $r$
        end for
        $\mathsf{diffused}[r] \leftarrow C$
        $\overline{C} \leftarrow A.\mathsf{execute}(C)$          ▷ Execute rushing adversary for round $r$
        for $c \in \bigcup_{1 \le r' \le r-\Delta} \mathsf{diffused}[r']$ do          ▷ Ensure $\Delta$-delay
            for $i \leftarrow 1$ to $n - t$ do
                $\mathsf{assert}(c \in \mathsf{seen}[i])$
            end for
        end for
    end for
end function
```

r, when $n_r - t_r < n_{r-1} - t_{r-1}$, the number of honest parties has decreased and $(n_{r-1} - t_{r-1}) - (n_r - t_r)$ honest ITM instances are killed by the environment. The choice of which instances will be killed is made by the adversary. When $n_r - t_r > n_{r-1} - t_{r-1}$, the number of honest parties has increased and $(n_r - t_r) - (n_{r-1} - t_{r-1})$ new honest instances are spawned up by cloning the state of some existing honest instances. The choice of which instances to clone is made by the adversary. An increasing or decreasing t_r does not affect the spawned instances. Finally, the choice of how n_r and t_r evolve is made *adaptively* by the adversary [59] based on the execution so far.

In the variable difficulty setting, we will concern ourselves with protocols in which the population evolution is gradual and observes certain bounds [61].

Definition 12 (Bounded demographic)**.** *For $\gamma \in \mathbb{R}^+$, a population sequence $(n_r)_{r\in\mathbb{N}}$ is called (γ, s)*-respecting *if for any S of at most s consecutive rounds,* $\max_{r\in S} n_r \le \gamma \cdot \min_{r\in S} n_r$.

Algorithm 14 The *variable difficulty* environment in the Δ-bounded delay setting.

```
function $\mathcal{Z}_{\Pi,\mathcal{A}}(1^\lambda)$    ▷ Boot adversarial ITM
    $A \leftarrow \mathsf{new}\ \mathcal{A}(1^\lambda)$
    $(n_1, t_1) \leftarrow A.\mathsf{init}()$
    $I \leftarrow \emptyset$
    for $i \leftarrow 1$ to $n_1 - t_1$ do    ▷ Boot honest ITMs
        $P_i \leftarrow \mathsf{new}\ \Pi(1^\lambda)$
        $I \leftarrow I \cup \{i\}$
    end for
    $\overline{C} \leftarrow [\,]$
    for $i \in I$ do
        $\overline{C}[i] \leftarrow [\,]$
    end for
    $\mathsf{seen} \leftarrow [\,]$
    $\mathsf{diffused} \leftarrow [\,]$
    for $r \leftarrow 1$ to $p(\lambda)$ do
        $C \leftarrow \emptyset$
        for $i \in I$ do
            $\mathsf{seen}[i] \leftarrow \mathsf{seen}[i] \cup \overline{C}[i]$
            $C \leftarrow C \cup \{P_i.\mathsf{execute}(\overline{C}[i])\}$    ▷ Execute honest party $i$ for round $r$
        end for
        $\mathsf{diffused}[r] \leftarrow C$
        $(\overline{C}, \mathsf{kill}, \mathsf{spawn}, t_r) \leftarrow A.\mathsf{execute}(C)$
        for $(i, j) \in \mathsf{spawn}$ do    ▷ Spawn new honest parties as clones of old ones
            $\mathsf{assert}(P_j \neq \bot)$
            $P_i \leftarrow P_j$
            $\mathsf{seen}[i] \leftarrow \mathsf{seen}[j]$
            $I \leftarrow I \cup \{i\}$
        end for
        for $i \in \mathsf{kill}$ do    ▷ Kill honest parties of the adversary's choice
            $P_i \leftarrow \bot$
            $I \leftarrow I \setminus \{i\}$
        end for
        $n_r \leftarrow |I| + t_r$
        for $c \in \bigcup_{1 \leq r' \leq r-\Delta} \mathsf{diffused}[r']$ do    ▷ Ensure Δ-delay
            for $i \in I$ do
                $\mathsf{assert}(c \in \mathsf{seen}[i])$
            end for
        end for
    end for
end function
```

2.5.5 Adaptive Corruption

The environment already allows the adversary to control a certain number of parties. If the identities of these parties are fixed at the beginning of the execution, as presented before, we speak of *static corruption*. This is sufficient to treat Proof-of-Work blockchains (Chapters 3, 5 and 4), but a more nuanced model is required for Proof-of-Stake (Chapters 6 and 7). In Proof-of-Stake protocols, just *which* parties are corrupted will be significant, because the corrupted parties will be maintaining important secrets in their local state (namely, keys controlling money). As such, it is not sufficient to capture the notion that t of n parties are corrupted, but the model will require the adversary to specify which parties are corrupted. At the point of corruption, the honest party P_i relinquishes its entire state to the adversary and is killed, while t is incremented by 1. In this more detailed model, the adversary first attempts to corrupt an honest party P_i by requesting to do so from the environment. This permission is granted after a certain delay of Λ rounds, where Λ is a parameter of our model (and can be different from the network delay Δ). In particular, if $\Lambda = 0$ we talk about *fully adaptive corruptions* and the corruption is immediate. The model with $\Lambda > 0$ is referred to as allowing *semi-adaptive corruptions*.

2.6 The Application Layer

In creating a decentralized cryptocurrency, the goal is to build a monetary system which is not reliant on any third parties. Money is moved around by issuing *transactions*, which instruct the transfer of a certain amount from one party to another. If Alice holds a certain amount of money and she wishes to give it to Bob, she creates a transaction which encodes, in some form, the instruction to pay Bob that certain amount. That transaction is encoded into a string that is then signed by Alice and transmitted to the network.

Contrary to centrally controlled currencies in which banks or payment processors are responsible for maintaining balances, decentralized cryptocurrencies allow any participant to verify the validity of a transaction. In order for this to be possible, every transaction is transmitted to every interested party on the network, a so-called *full node*, who validates it. By recording all past transactions, every participant is aware of *who owns what* and can thereby determine if an attempt to spend money is legitimate. No special privileged or trusted nodes exist on the network.

We now formally define what a transaction is and look at the transaction formats for Bitcoin and Ethereum. In addition to being the largest cryptocurrencies, these two systems define two prototypal transaction formats known as the *UTXO model* and the *Account model*. All other cryptocurrencies adopt either model, or a hybrid of the two [154].

2.6.1 Transactions

Transactions are part of the *application layer*. As this thesis concerns itself with the *consensus layer* which organizes transactions into sequences, we will generally not concern ourselves about their format, and we will allow the application layer to specify any transaction format it wishes. Therefore, transactions can be any strings that are deemed valid by the application layer.

Definition 13 (Transaction). *A predefined language $\mathcal{T}$ of strings in $\{0,1\}^*$ is called a* transaction language. *Elements $tx \in \mathcal{T}$ are called* transactions.

While specific applications such as Bitcoin or Ethereum mandate that transactions follow a certain format and must include, for example, signatures, we will not impose such requirements on our protocol. As there are preconceptions about what constitutes a transaction, we feel the need to give some examples of transaction languages. The set of valid transactions could be the empty set, the set $\{0, 1\}$ of bits, the set of natural numbers, or the set of triplets of a message, a public key, and a digital signature that pass verification under a certain signature scheme. While the latter corresponds more closely to practical protocols such as Bitcoin, our treatment is quite general and has no requirements to remain within this strict format.

Once the set of valid transactions $\mathcal{T}$ has been defined by the application layer, it can now specify a *validity language* which specifies which *sequences* of transactions are valid. This captures what is deemed to be a valid transaction given a previous history of transactions in the system and allows the application layer to specify, for example, that double spending is not allowed.

Definition 14 (Validity Language)**.** *Given a transaction language $\mathcal{T}$, a predifined set of finite transaction sequences $\mathbb{V} \subseteq \mathcal{T}^*$ is called its* validity language.

The validity languages we will concern ourselves with have the property that they contain the empty transaction sequence ϵ. This is useful because it allows a node booting up anew to begin with an empty transaction sequence before it starts receiving and validating transactions. Our validity languages are also *extensible*: Given a valid transaction sequence $\overline{\mathsf{tx}} \in \mathbb{V}$ and a new candidate transaction $\mathsf{tx} \in \mathcal{T}$, it is possible to check whether $\overline{\mathsf{tx}} \,\|\, \mathsf{tx} \in \mathbb{V}$ by applying a predicate $\mathsf{extend}(\overline{\mathsf{tx}}, \mathsf{tx})$. This extend predicate ensures that the transaction only spends money that belongs to it and exists in the system. Furthermore, once a transaction which invalidates the sequence has been added to the sequence, the sequence remains invalid.

In addition to allowing transactions that spend existing money, it must be possible to also create new money. The macroeconomic rules for money creation are captured by another application-specific predicate $\mathsf{mints}(\overline{\mathsf{tx}}, \mathsf{tx})$ which checks whether a transaction tx is a valid minting transaction. The rules for this can include, for example, limiting the amount of money generated per block. In typical cryptocurrencies, there is one minting transaction allowed per block and the amount that can be generated by this minting transaction has an upper bound which is algorithmically determined [54]. We will leave this predicate undefined in our treatment.

Validity by extension is captured by the definition below:

Definition 15 (Validity by extension)**.** *Given an extension predicate $\mathsf{extends}$, and a transaction language $\mathcal{T}$, the validity language $\mathbb{V}^{\mathsf{extends},\mathsf{mints},\mathcal{T}}$ obtained* by extension *is the minimum set of transaction sequences which satisfies the following:*

1. ***Base.*** $\epsilon \in \mathbb{V}^{\mathsf{extends},\mathsf{mints},\mathcal{T}}$
2. ***Extension.*** *For all $\overline{\mathsf{tx}} \in \mathbb{V}^{\mathsf{extends},\mathsf{mints},\mathcal{T}}$, for all $\mathsf{tx} \in \mathcal{T}$, if $\mathsf{extends}(\overline{\mathsf{tx}}, \mathsf{tx})$ or $\mathsf{mints}(\overline{\mathsf{tx}}, \mathsf{tx})$ then $\overline{\mathsf{tx}} \,\|\, \mathsf{tx} \in \mathbb{V}^{\mathsf{extends},\mathsf{mints},\mathcal{T}}$.*

From the above definition, the following result follows immediately.

Lemma 4 (Validity Language Monotonicity)**.** *Consider a validity language $\mathbb{V}$ generated* by extension *of a transaction language $\mathcal{T}$. For all $w, w' \in \mathcal{T}^*$ we have $w \notin \mathbb{V} \Rightarrow w \,\|\, w' \notin \mathbb{V}$.*

Monotonicity mandates the natural property that if a sequence of transactions is invalid, it cannot become valid again by adding further transactions.

Furthermore, it is useful to ensure transactions are unique. This is captured in the following requirement for the validity languages of our interest.

Definition 16 (Validity Language Transaction Uniqueness)**.** *A validity language* $\mathbb{V}$ *pertaining to a transaction language* $\mathcal{T}$ *has* transaction uniqueness *if it never contain the same transaction twice: for any* $\mathsf{tx} \in \mathcal{T}$ *and any* $w_1, w_2, w_3 \in \mathcal{T}^*$ *we have*

$$w_1 \,\|\, \mathsf{tx} \,\|\, w_2 \,\|\, \mathsf{tx} \,\|\, w_3 \notin \mathbb{V} \ .$$

The natural "uniqueness" property of transactions holds in existing implementations, but is not necessary for our treatment, albeit allowing for some simplifications.

For illustrative purposes, and because we aim our protocols to be deployable to existing blockchain systems, in particular to Bitcoin-compatible and Ethereum-compatible chains, we now explore two particular approaches to the transaction and validity languages employed in the blockchain space: the UTXO model and the Account model. We note, however, that our consensus protocols which enable compression and interoperability are not limited to these two models, but are generic.

2.6.2 Keys and addresses

While the consensus layer does not require this from the application layer, all known application layer instantiations make use of public/private keys. These keys are used to identify money holders in the system. The public key is used to *receive* money, while the private key is used to sign instructions to *send* money. The public key is publicized, while the private key remains secret. As the public key can be assumed to be known to anyone, anyone can *send* money to everyone and this cannot be prevented. The receipient does not need to authorize a payment to receive it.

To somewhat increase anonymity, it is recommended that public keys used to receive money are not recycled. In particular, it is recommended that a new public key is issued every time money is to be received. The set of all the private keys that belong to a user are known as a *wallet*.

The lifecycle of money is as follows. If Alice wishes to pay Bob, first she contacts Bob to ask for his public key. Bob generates a new public/private key pair and send the public key to Alice. Alice then issues a payment instruction which instructs Bob to be paid and contains the amount payable as well as Bob's public key. When Bob wishes to spend the money he received from Alice, he uses the respective private key to sign a message sending out a payment to someone else.

Public keys used to send money are encoded into *addresses* using special encodings such as base58 or mixed-case [34]. These addresses can include checksums or other features which make them harder to miscommunicate or mistype.

2.6.3 The UTXO Model

The UTXO model was first introduced in Bitcoin. In the UTXO model, each transaction is a node in a *weighted directed acyclic graph.* Despite being represented as graph nodes, transactions do not correspond to accounts or account holders, but denote a payment. An edge incoming to a transaction designates *who is paying*, and an edge outgoing from a transaction denotes *who is being paid*. The edge is annotated with two pieces of information: Its *weight* that designates the *amount* paid; and the *address* of the payment receipient. The weights are integers denominated in the smallest denomination of the currency. In Bitcoin, this is the Satoshi, which equals 10^{-8} BTC. Denominating amounts in satoshis rather than bitcoins is helpful because it avoids rounding errors. The data of a transaction contains the collection of its input and output edges. The hash of that data gives the *transaction id*, or *txid*, which can be used to uniquely refer to a transaction. In the case of Bitcoin, the hash function used for this purpose is `SHA256`.

A directed edge connecting two transactions denotes the transfer of money from one transaction to the other. It indicates that money was paid to a beneficiary through the first transaction, and that beneficiary subsequently spent the money they had received through the second transaction. As such, a *coin is a chain of transactions*. Transactions can have *dangling outgoing edges*, which are edges that are outgoing from a transaction but have not yet been connected as incoming edges into another transaction. These edges have not yet been spent and are available for spending. They are known as Unspent Transaction Outputs (UTXOs). The collection of all UTXOs constitutes the available money in the system.

The transaction DAG is known to all network participants. To determine how much money Alice owns, she collects the UTXOs of the transaction DAG and filters it according to the addresses that she is the owner of; that is, addresses that correspond to a public key for which she owns the corresponding private key. The sum of the amounts in these UTXOs is the total amount of money she owns. She does not look at *spent* transaction outputs, because these have already been consumed and they cannot be spent again.

To send money to Bob, Alice finds a UTXO e_1 she owns. She then creates a new transaction `tx` with one incoming edge e_1 and one outgoing edge e_2 and connects the UTXO e_1 as the incoming edge of `tx`. As e_1 is no longer a dangling edge since it is now incoming to `tx`, it is longer in the UTXO set. The outgoing edge of `tx` is now dangling, and so is now part of the UTXO set. Therefore, with the transaction, the UTXO set is updated so that e_1 is removed from it and e_2 is added to it. Edge e_2 is weighted by the amount that Alice wishes to pay and annotated by the address of the beneficiary to be paid, which is deduced from the public key of the receipient.

To prove that she is the rightful owner of e_1, Alice produces a signature using the private key corresponding to the public key that corresponds to the address annotated in e_1. The contents signed by Alice's signature include the reference to e_1 so that it is clear *which* coin Alice is spending. The contents also include e_2 and in particular the amount and new beneficiary. This ensures Alice's payment cannot be forged to be made payable to a different beneficiary.

Alice then diffuses her transaction on the network, which is subsequently received by Bob.

A transaction can have multiple inputs. This is useful if Alice has received multiple payments and she wants to make a larger payment. She creates one transaction with multiple inputs and a single output. The transaction consumes all inputs and pays them to the given output. A transaction can also have multiple outputs. Because every transaction fully consumes its inputs, if Alice has received a large payment in a single UTXO, she can consume it via a transaction that contains multiple outputs so that she can make multiple payments with it simultaneously.

She can also consume a large UTXO by creating a transaction that contains two outputs: one output that pays out some amount to her desired beneficiary, and another that pays the rest of the money back to Alice. Outputs that pay back the original owner are known as change outputs. Change outputs are paid out to addresses created for this purpose that are known as *change addresses* and are owned by the original owner. A typical transaction on a UTXO-based network consumes one or more inputs all owned by the same owner and contains two outputs: One which indicates the payment beneficiary, and another which indicates the change.

An edge can be a dangling *outgoing* edge from a transaction, in that it is not yet an *incoming* edge to any other transaction. However, every edge must have a transaction from which it is outgoing. As such, every edge can be associated with a unique transaction that produced it. The outgoing edges of every transaction are ordered and indexed by the natural numbers. An edge can therefore be identified by the transaction that produced it together with its index among the outgoing edges of that transaction. This pair is known as an *outpoint*.

Definition 17 (UTXO transaction)**.** *A* UTXO transaction $\texttt{tx}$ *is a pair* (in, out) *of inputs and outputs such that in is a sequence of outpoints* $(in_1, in_2, \cdots)$ *and out is a sequence of edges* $(out_1, out_2, \cdots)$*. It must hold that every input* in_i *is a tuple* $(txid_i, j_i, \sigma_i)$ *where the pair* $(txid_i, j_i)$ *is an outpoint indicated by the transaction id* $txid_i$ *and an index* j_i *which marks the output index within the transaction identified by* $txid_i$*. The signature* σ_i *is a signature on* $\texttt{tx}$ *in which all* σ_k *have been replaced by* ϵ*. Additionally, it must hold that every output* out_i *is a pair* $(amount, pk)$ *indicating the amount payable and the beneficiary address* pk*. The set of all syntactically valid UTXO transactions is the UTXO transaction language* $\mathcal{T}_{\text{UTXO}}$*.*

A transaction's signatures must be valid. They must sign the plaintext $\texttt{tx}$ in which all σ_k have been replaced by ϵ. The replacement is made so that signatures do not have to sign themselves, which would be an impossible task. They must also pass the verification using the public key indicated by the respective outpoint. A transaction must follow the *conservation principle* which mandates that the sum of output amounts cannot exceed the sum of input amounts.

These rules define transaction validity inductively. When Alice receives a transaction from the network, whether it pertains to a payment to her or not, she needs to validate it. This require Alice to maintain a currently valid UTXO set against which she will compare the transaction and which she will use to update this UTXO set. Therefore, to verify a transaction, the procedure followed is thus. First, Alice already holds some valid UTXO set by the inductive hypothesis (she starts with the empty UTXO set). First, she checks that the inputs to the new transaction refer to outpoints that are in her existing UTXO set. She follows the outpoint pointers to check that the law of conservation holds for the new transaction. She also verifies all the signatures on the inputs of the new transaction using the public keys that appear in the outputs referenced by the outpoints. As long as everything is valid, she updates her UTXO to remove the outputs referenced by the outpoints and add the outputs of the new transaction.

We are now ready to define transaction validity formally in the UTXO model. While the UTXO model should already be straightforward, we go through the exercise of presenting it precisely to illustrate the expressiveness of validity language formulations.

First, we define outpoints.

Definition 18 (Outpoint)**.** *Consider the UTXO transactions language $\mathcal{T}_{\text{UTXO}}$ and the transaction id hash function H. Let $\overline{\mathsf{tx}} \in \mathcal{T}^*_{\text{UTXO}}$ be a transaction sequence containing unique transactions. Define the transaction lookup function $\textit{lookup-tx}_{\overline{\mathsf{tx}}}(txid)$ to equal tx if $H(\mathsf{tx}) = txid$ and $\mathsf{tx} \in \overline{\mathsf{tx}}$, otherwise set $\textit{lookup-tx}_{\overline{\mathsf{tx}}}(txid) = \perp$. Define the* outpoint lookup function *$\textit{outpoint}_{\overline{\mathsf{tx}}}(txid, j)$ to be the j^{th} item of out where $(in, out) = \textit{lookup-tx}(txid)$ if such a j exists. Otherwise set $\textit{outpoint}_{\overline{\mathsf{tx}}}(txid, j) = \perp$.*

We can now define what the UTXO set of a transaction sequence is.

Definition 19 (UTXO set)**.** *Consider the transaction language $\mathcal{T}_{\text{UTXO}}$ and the transaction id hash function H. Let $\overline{\mathsf{tx}} \in \mathcal{T}^*_{\text{UTXO}}$ be a transaction sequence containing unique transactions. The* UTXO set $\text{UTXO}(\overline{\mathsf{tx}})$ *of $\overline{\mathsf{tx}}$ is defined as the set which contains all outpoints $(txid, j)$ with the following properties:*

1. ***Unspent.*** *There is no $\mathsf{tx}' \in \overline{\mathsf{tx}}$ with $\mathsf{tx}' = (in', out')$ such that $(txid, j) \in in'$.*
2. ***Transaction output.*** *There is some $\mathsf{tx} \in \overline{\mathsf{tx}}$ with $\mathsf{tx} = (in, out)$ such that $H(\mathsf{tx}) = txid$ and $1 \leq j \leq |out|$.*

We are now ready to define the validity language for a UTXO system. We will define our validity language by extension according to Definition 15. The $\mathsf{extends}_{\text{UTXO}}(\overline{\mathsf{tx}}, \mathsf{tx})$ predicate checks that the transaction tx can be applied on top of the existing transaction sequence $\overline{\mathsf{tx}}$ and is defined as follows.

Definition 20 (UTXO validity)**.** *Let $\overline{\mathsf{tx}} \in \mathcal{T}^*_{\text{UTXO}}$ and $\mathsf{tx} \in \mathcal{T}_{\text{UTXO}}$ with $\mathsf{tx} = (in, out)$ and let $\textit{mint}$ be a* minting predicate. *Let $S = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ be a secure signature scheme.*

We define $\textit{extends}_{\text{UTXO}}(\overline{\mathsf{tx}}, \mathsf{tx})$ to be $\textit{true}$ if all of the following conditions hold:

1. ***Rightful.*** *For all $(txid, j, \sigma) \in in$ we have that $\mathsf{Ver}(pk, \mathsf{tx}', \sigma)$ where $(amount, pk) = \textit{outpoint}_{\overline{\mathsf{tx}}}(txid, j)$ and tx' denotes tx with all signatures replaced by ϵ.*
2. ***Unspent.*** *All $(txid, j, \sigma) \in in$ form unique outpoints $(txid, j)$ and for all $(txid, j, \sigma) \in in$ we have $(txid, j) \in \text{UTXO}(\overline{\mathsf{tx}})$*
3. ***Conserving.***

$$\sum_{in_i \in in} in_i.\textit{amount} \geq \sum_{out_i \in out} out_i.\textit{amount}\,.$$

The UTXO validity language $\mathbb{V}_{\text{UTXO}}$ *with macroeconomic policy $\textit{mints}$ is defined as*

$$\mathbb{V}^{\textit{extends}_{\text{UTXO}}, \textit{mints}}\,.$$

2.6.4 The Account Model

A different approach to transactions is followed in the Account Model, which was first put forth by Ethereum. Instead of maintaining UTXOs, the Account Model maintains account balances. Transactions are instructions to transfer an amount from one account to another. Accounts are represented by addresses. A transaction therefore contains the source account, the target account, the amount, as well as a signature authorizing the transfer. The conservation principle here mandates that the source account must have sufficient balance to cover the amount a transaction wishes to spend.

Contrary to the UTXO model where every UTXO is spent only once, here it is possible to have multiple transactions which spend from the same source account, pay into the same target account, and transmit the same amount. As such, these transactions require a nonce, or counter, which ensures they are unique. This counter is necessary. If such no such counter was present, the system needing to support a repeated transfer would accept the same transaction twice. But such admissibility of transaction duplication is problematic, as an adversary could replay an existing transaction, with the same signature, benefiting her account twice, even though no such intention was recorded by the sender. The counter is therefore required to signal the fact that the sender wishes to initiate yet another transfer.

Definition 21 (Account Transaction)**.** *An* account transaction tx *is a tuple* $(\mathit{from}, \mathit{to}, \mathit{amount}, ctr, \sigma)$ *such that* $\mathit{from} \neq \mathit{to}$*. The signature* σ *is a signature on* $(\mathit{from}, \mathit{to}, \mathit{amount}, ctr, \epsilon)$*. The set of all syntactically valid account transactions is the account transaction language* $\mathcal{T}_{\text{account}}$*.*

Balances can be obtained from a transaction sequence by summing the amounts transfered. We will then make use of balances to define whether an account transaction validly extends a transaction sequence.

Definition 22 (Account Balances)**.** *Let* $\overline{\mathsf{tx}} \subseteq \mathcal{T}^*_{\text{account}}$ *be an account transaction sequence. We define the* balance $\mathit{balance}_{\overline{\mathsf{tx}}}(\mathit{acc})$ *of an account* acc *at the end of the transaction sequence* $\overline{\mathsf{tx}}$ *as follows. If* $\overline{\mathsf{tx}} = \epsilon$*, then define* $\mathit{balance}_{\overline{\mathsf{tx}}}(\mathit{acc}) = 0$*. Otherwise,* $\overline{\mathsf{tx}}$ *is non-empty, so set* $\overline{\mathsf{tx}} = \overline{\mathsf{tx}}' \,\|\, \mathsf{tx}$ *and let* $(\mathit{from}, \mathit{to}, \mathit{amount}, \sigma) = \mathsf{tx}$*. Recursively let* $\mathit{balance}' = \mathit{balance}_{\overline{\mathsf{tx}}'}(\mathit{acc})$*.*

- *If* $\mathit{from} = \mathit{acc}$*, then define* $\mathit{balance}_{\overline{\mathsf{tx}}}(\mathit{acc}) = \mathit{balance}' - \mathit{amount}$*.*
- *If* $\mathit{to} = \mathit{acc}$*, then define* $\mathit{balance}_{\overline{\mathsf{tx}}}(\mathit{acc}) = \mathit{balance}' + \mathit{amount}$*.*
- *Otherwise define* $\mathit{balance}_{\overline{\mathsf{tx}}}(\mathit{acc}) = \mathit{balance}'$*.*

We are now ready to define the validity language for an account-based system. Again, we will define our validity language by extension according to Definition 15.

The $\mathsf{extends}_{\text{account}}(\overline{\mathsf{tx}}, \mathsf{tx})$ predicate checks that the transaction tx can be applied on top of the existing transaction sequence $\overline{\mathsf{tx}}$ and is defined as follows.

Definition 23 (Account validity)**.** *Let* $\overline{\mathsf{tx}} \in \mathcal{T}^*_{\text{account}}$ *and* $\mathsf{tx} \in \mathcal{T}_{\text{account}}$*. Let* $\mathsf{tx} = (\mathit{from}, \mathit{to}, \mathit{amount}, \mathit{ctr}, \sigma)$ *and let* mint *be a* minting predicate*. Let* $S = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ *be a secure signature scheme.*

We define $\mathit{extends}_{\text{account}}(\overline{\mathsf{tx}}, \mathsf{tx})$ *to be* ***true*** *if:*

1. ***Rightful.*** $\mathsf{Ver}(\mathit{from}, \mathsf{tx}', \sigma)$ *where and* $\mathsf{tx}' = (\mathit{from}, \mathit{to}, \mathit{amount}, \mathit{ctr}, \epsilon)$*.*
2. ***Conserving.*** *For all* acc *it holds that* $\mathit{balances}_{\overline{\mathsf{tx}} \,\|\, \mathsf{tx}}(\mathit{acc}) \geq 0$*.*

The account validity language $\mathbb{V}_{\text{account}}$ *with macroeconomic policy* mints *is defined as* $\mathbb{V}^{\mathit{extends}_{\text{account}}, \mathit{mints}}$*.*

2.7 Blockchains

2.7.1 The Consensus Problem

As the peer-to-peer parties in a cryptocurrency system exchange money, transactions are generated and disseminated on the network. Messages on the network naturally do not arrive at every party in the same order (as is formally captured by our network model of Section 2.5.2). The parties need to determine whether a newly arriving transaction is valid by comparing it against the currently adopted transaction sequence. The order in which transactions are processed is important because it can affect the validity of the transaction sequence. If every party organizes their transactions according to the order they receive them from the network, they will end up disagreeing about whether a transaction is valid. This gives rise to the *double spending problem.*

Consider the following situation. In the UTXO model, Alice sends 1 coin to Eve via a legitimate transaction tx which has only one output. This transaction is broadcast to the network and every party adopts it as valid. Subsequently, Eve, who is malicious, generates two transactions $\mathsf{tx}_1, \mathsf{tx}_2$ each of which consumes the outpoint of tx and generates a single new output of the same value, but to a different receipient each: tx_1 pays Bob, while tx_2 pays back Charlie. Both tx_1 and tx_2 are broadcast on the network at about the same time, and it so happens that Bob receives tx_1 before tx_2, while Charlie receives tx_2 before tx_1. If they adopt transactions in the order they receive them, Bob will consider tx_1 to be valid and will adopt it, while rejecting tx_2 as invalid. On the other hand, Charlie will do the opposite. Note that, without the existence of the other, each of these transactions is individually valid, as it is rightful and conserving. When the time comes for Bob to pay Charlie, Charlie will not agree with Bob that the money belongs to Bob. Without *consensus* on which transaction occurred first, the economic participants cannot agree who owns what, and the monetary system breaks down.

Before we describe the solution that proof-of-work offers, let us first observe that obvious solutions do not work. Consider the following protocol: Since only a malicious party would create a signature on both transactions tx_1 and tx_2, we could eliminate both transactions and mark them as invalid as soon as the double spend has been detected. This is not a good strategy. In this case, Eve can first pay Bob using tx_1. After she receives the services of Bob, at a much later time, she can publish tx_2, rendering Bob's money invalid. While Eve does not gain from this behavior, the trustworthiness of the monetary system is subverted. Simple adjustments to this strategy do not work either. Noting that tx_2 would have to be broadcast much later, consider, for example, a protocol which waits for some time Δ prior to establishing whether a double spend has occurred or not. If tx_1 appears at some time r_0 without any double spending transaction appearing prior to time $r_0 + \Delta$, the transaction tx_1 can be accepted. Any double spending transaction tx_2 appearing after time $r_0 + \Delta$ is rendered invalid. On the other hand, if tx_2 appears within time Δ, both tx_1 and tx_2 are rendered invalid. This protocol can be subverted by Eve as follows. She initially broadcasts tx_1 at round r_0. When time $r_0 + \Delta$ approaches, she broadcasts tx_2 and it so happens that Bob receives tx_2 prior to time $r_0 + \Delta$, while Charlie receives tx_2 after time $r_0 + \Delta$. This would cause Bob to reject both tx_1 and tx_2, while Charlie would only reject tx_2. No simple protocol can withstand attacks that cause such disagreements.

The problem then becomes an issue of coming to an agreement about the *order* in which transactions have occurred. This problem of agreement is solved by the *consensus layer*, the part of a decentralized system which attempts to organize *application layer* transactions into sequences that belong to the application's validity language by ordering them into transaction sequences.

As in later chapters we will speak about multiple decentralized systems interoperating, it is helpful to note which *ledger* $\mathbf{L}$ the party in question maintains. In the simplest case, each ledger is associated with a unique cryptocurrency, but we will later relax this. The $\mathbf{L}$ is the protocol which the honest parties execute to maintain their local views.

Definition 24 (Ledger)**.** *A ledger protocol* $\mathbf{L}$ *is an honest probabilistic polynomial-time algorithm for maintaining a transaction sequence belonging to a validity language* $\mathbb{V}$*.*

To make the problem precise, we will consider the honest parties' beliefs on *which transactions have occurred and what their order is.* The transaction sequence reported by an honest party at a particular round is known as the *ledger view* of the party. An honest party's protocol exposes two methods related to transaction processing: A method to *read* the current transaction sequence and a method to *write* a transaction to their ledger.

Definition 25 (Ledger view)**.** *The* ledger view $\mathbf{L}^P[r] \in \mathbb{V}$ *of an honest party* P *at round* r *pertaining to ledger* $\mathbf{L}$ *with validity language* $\mathbb{V}$ *is the transaction sequence reported by the honest party when it is given the instruction to* read *the current transaction sequence.*

Definition 26 (Confirmation)**.** *A transaction* tx *of ledger* $\mathbf{L}$ *is called* confirmed *by honest party* P *at round* r *if* $\mathsf{tx} \in \mathbf{L}^P[r]$*. We say that a transaction is* confirmed *at round* r *if it is confirmed by all honest parties.*

A good ledger protocol will exhibit two virtues: Persistence and Liveness. On one hand, persistence mandates that the parties eventually come to agreement. On the other hand, liveness mandates that transactions that occur eventually appear in everyone's view.

Definition 27 (Persistence)**.** *A ledger protocol* $\mathbf{L}$ *has* persistence *with parameter* λ *(the* persistence parameter*) if for any two honest parties* P_1, P_2 *and two rounds* $r_1 \leq r_2 + \lambda$*, it holds* $\mathbf{L}^{P_1}[r_1] \preceq \mathbf{L}^{P_2}[r_2]$*.*

Note that we do not require that the ledger views of the honest parties are equal, but that, after sufficient time, they are a prefix of each other. The reason for this is that, while both parties will eventually agree on their ledger view up to a point, each may include a transaction at a later time. Until an honest party can be certain that a transaction is confirmed and is ready to report it in a particular position within its ledger view as *stable*, it will keep the transaction as *unstable* and not report it in its view. The duration, in number of rounds, during which transactions can remain unstable is known as the *persistence parameter.*

Definition 28 (Liveness)**.** *A ledger protocol* $\mathbf{L}$ *has* liveness *with parameter* u *(the* liveness parameter*) if the following holds. If* all *honest parties in the system attempt to include a transaction* tx *(which validly extends their ledger views) for all rounds* $r, r+1, \cdots, r+u$ *then, at any round* $r' > r+u$*, any queried honest party* P *will report* $\mathsf{tx} \in \mathbf{L}^P[r']$*.*

Liveness ensures that transactions appear on the ledger views of honest parties. The liveness parameter u denotes the duration for which a party has to wait until a transaction is confirmed. When we develop our protocols and we require a transaction to appear in a ledger, this parameter will appear as waiting time to ensure the transaction has taken its position to all honest parties' ledgers.

We note that it is trivial to design a ledger protocol which has either liveness or persistence, but not both. To achieve persistence without liveness, the honest parties always return ϵ when their ledger views are read. This trivially satisfies persistence, as the empty sequence as a prefix of itself. Liveness is not satisfied because transactions are never confirmed. To achieve liveness without persistence, the honest parties include transactions in their ledgers in the order they see them on the network. This ensures transactions appear immediately, which achieves liveness, but the ledgers of different parties will disagree about their contents and order, and so persistence is lost. The challenge in creating a *secure ledger* will be to achieve both properties.

Definition 29 (Secure ledger)**.** *A ledger* $\mathbf{L}$ *is called* secure *with parameters* λ*,* u *if it has persistence with parameters* λ *and liveness with parameter* u*.*

To solve the consensus problem and create a secure ledger protocol, the honest parties must come to agreement and ensure that the adversary does not split their views while including all transactions. In order to do that, we will allow parties to *vote* on a transaction order. We will then utilize the majority of votes to arrive at a conclusion. This requires us to introduce an additional assumption: That the majority of parties are honest.

2.7.2 Proof-of-Work

In our decentralized systems in which the number of participants is unknown and their channels remain unauthenticated, we will assume that the *majority* of the population is honest [60]. This assumption is summarized in the following equation:

$$t \leq (1 - \delta)(n - t)$$

This mandates that the number of adversarial parties t is less than the honest parties $n-t$ by a fraction determined by the parameter δ, the *honest advantage.* We will refer to this model as the $\frac{1}{2}$-adversary. This assumption is generally necessary to solve the consensus problem in polynomial time [120], although it can be temporarily relaxed [9]. Throughout this work, we will assume that the honest majority holds during *all* rounds.

In some cases, most notably in our constructions of Chapter 5, our results will be limited to weaker adversaries that satisfy more stringent assumptions and are bounded by $\frac{1}{3}$ or $\frac{1}{4}$ as defined below:

Definition 30 (Bounded adversaries)**.** *We say that a population has a* ϕ*-bounded adversary (where* ϕ *typically takes the values of* $\frac{1}{2}$*,* $\frac{1}{3}$*, or* $\frac{1}{4}$*) for some* $0 < \phi \leq \frac{1}{2}$ *if*

$$\frac{t}{n} \leq (1 - \delta)\phi$$

Algorithm 15 The Proof-of-Work discovery algorithm

```
1: function $\mathsf{pow}_{H,T,q}(m)$
2:     for $ctr \leftarrow 1$ to $q$ do
3:         if $H(m \,\|\, ctr) \leq T$ then
4:             return $ctr$
5:         end if
6:     end for
7:     return $\bot$
8: end function
```

The honest majority assumption denotes a $\frac{1}{2}$-bounded adversary. If the number of parties changes from round to round, the respective assumption must be maintained throughout the whole execution.

While theoretically the honest majority assumption discusses population counts, in practice this is translated into a majority in computational power. To achieve this in our model, we bound the number of Random Oracle queries which are allowed per party per round by some constant $q > 0$, which is the same for all parties. Limited computational power is then captured by the limited number of queries to the Random Oracle. As such, each honest party has q available queries per round, for a total of $(n-t)q$ queries per round, while the adversary has tq available queries per round. We call this model the *q-bounded model.*

When working in the synchronous setting, we will set q to be some polynomial of the security parameter. As the queries are counted *per round*, the model captures the fact that up to q queries can be made prior to an honest party being able to communicate the result of their query to the rest of the honest parties. In the Δ-bounded delay setting, we will simplify by setting $q = 1$, as the number of queries allowed before a message reaches the rest of the network can be controlled by adjusting Δ. Concretely, we note that a Δ-bounded delay setting with $q > 1$ is captured by an equivalent model in which $\Delta' = \Delta + q$ and $q' = 1$.

The honest parties try to distinguish between messages diffused by the other honest parties and the adversary. This is a form of *voting.* The parties vote on a message m by solving the *proof-of-work equation* [52] which is defined as follows.

Definition 31 (Proof-of-Work)**.** *Consider a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\kappa$ and some $T \leq 2^\kappa$. A nonce $ctr \in \{0,1\}^*$ is called* proof-of-work *for message $m \in \{0,1\}^*$ against* target T *if the following inequality holds:*

$$H(m \,\|\, ctr) \leq T$$

If H is modelled as a Random Oracle, the best way to solve proof-of-work for a previously unseen message m is by brute force: iterate through all possible ctr values until a solution is found. The message m must have sufficient entropy to ensure no other party is looking for proof-of-work for the same message. The algorithm that looks for a proof-of-work solution, given q queries available in a round, is illustrated in Algorithm 15.

This makes the proof-of-work problem of finding a nonce a *moderately hard* problem. In the extreme case where $T = 1$, the problem is computationally hard and lies in EXP, as it takes an exponential number of random oracle queries to find the value required. On the other end, if $T = 2^{\kappa-1}$, the problem is computationally easy and lies in P, as only a single query is required to solve the problem in expectation. When T takes a moderate value, the problem requires a moderate number of queries until a solution is found. Crucially, the expected number of queries can be controlled by adjusting the target parameter T. Proof-of-work has the property that it is moderately hard to find, but once found it can be easily verified by checking that the equation holds.

Definition 32 (Successful Query)**.** *We call a query to the Random Oracle that satisfies the proof-of-work equation a* successful query*.*

Lemma 5 (Successful Query)**.** *The probability of a new query being successful* $p = \frac{T}{2^\kappa}$*.*

We now define the random variables X_r, Y_r that specify whether a round was *successful* and *uniquely successful*.

Definition 33. *If an honest party has made a successful query during a round* r*, then we call* r *a* successful round *and set* $X_r = 1$*; otherwise, we set* $X_r = 0$*. If among the* honest *queries during* r *only* one *was successful, we call* r *a* uniquely successful round *and we set* $Y_r = 1$*; otherwise, we set* $Y_r = 0$*.*

It is of course possible that, in addition to the honest parties, the adversary could have had successful queries during a successful or uniquely successful round. The adversary can also succeed in rounds during which the honest parties were unsuccessful. We let $Z_{rj} = 1$ if during round r the j^{th} adversarial query was successful, where j ranges from 1 to tq; otherwise we let $Z_{rj} = 0$. For a set of rounds S, we define $X(S) = \sum_{r \in S} X_r$ and $Y(S) = \sum_{r \in S} Y_r$ as well as $Z(S) = \sum_{r \in S} \sum_{j=1}^{tq} Z_{rj}$.

In reality, not every honest party has the same computational power, and multiple honest parties may combine their computational power into a so-called *mining pool*. These can be captured by treating a more powerful honest party as multiple honest parties each of which contributes q queries per round. This is made explicit for the adversary, as she does not incur any network overhead to achieve communication between the t corrupted parties. On the contrary, honest players discovering proof-of-work must diffuse it to the network at a given round and wait for it to be received and validated by the rest of the honest players at the beginning of the next round (or Δ rounds later in the Δ-bounded delay model).

These random variables X, Y, Z have the following expected values. For all rounds r:

- $\mathbb{E}[X_r] = 1 - (1-p)^{q(n-t)}$
- $\mathbb{E}[Y_r] = pq(n-t)(1-p)^{q(n-t-1)}$
- $\mathbb{E}[Z_r] = pqt$

The random variables X_r and Y_r are Bernoulli distributed, while the random variable Z_r is Binomially distributed. The expectation for the adversary assumes that she makes use of all her available Random Oracle queries. As we will only apply upper bounds to $\mathbb{E}[Z_r]$, this is without loss of generality.

The expectation $\mathbb{E}[X_r]$ is a value critical for our protocols and will be denoted f throughout this work. If f is too small, the proof-of-work problem is too hard and there is no progress, harming liveness. If f is large, the proof-of-work problem is too easy and the honest parties all solve proof-of-work simultaneously. In that case, there are no intermittent *periods of silence* that honest parties can utilize to reach agreement, and thus persistence is harmed. The parameter T must be carefully calibrated according to n and q so that f takes a balanced value.

As rounds are independent, the expectations for $\mathbb{E}[X(S)]$, $\mathbb{E}[Y(S)]$, $\mathbb{E}[Z(S)]$ are equal to $|S|\,\mathbb{E}[X_r]$, $|S|\,\mathbb{E}[Y_r]$, and $|S|\,\mathbb{E}[Z_r]$ respectively. If $|S|$ is sufficiently large, the actual values attained by $\mathbb{E}[X(S)]$, $\mathbb{E}[Y(S)]$, and $\mathbb{E}[Z(S)]$ will most likely be near their expectation, except with some small error ϵ. This intuition is formally captured by the notion of a *typical query distribution*:

Definition 34 (Typical Query Distribution)**.** *Let $\epsilon \in (0, 1)$ (the* Chernoff error*) and integer $\lambda \geq \frac{2}{f}$ (the* wait time*). Let S be a set of rounds with $|S| \geq \lambda$. An execution has* (ϵ, λ)-typical query distribution *if:*

- $(1 - \epsilon)\,\mathbb{E}[X(S)] < X(S) < (1 + \epsilon)\,\mathbb{E}[X(S)]$
- $(1 - \epsilon)\,\mathbb{E}[Y(S)] < Y(S)$
- $Z(S) < \mathbb{E}[Z(S)] + \epsilon\,\mathbb{E}[X(S)]$

We are now ready to state the full *honest majority assumption.* Our assumption requires that the honest majority gap δ is sufficient to account for both the Chernoff error ϵ as well as the the lack of silence due to a potentially large parameter f.

Definition 35 (Honest Majority Assumption)**.** *We say that a population has* honest majority *if*

$$t \leq (1 - \delta)(n - t)$$

where

$$\delta > 3f + 3\epsilon\,.$$

The honest majority assumption allows us to argue that the number of uniquely successful queries of the honest parties are generally larger than the number of successful queries of the adversary. This follows immediately if our query distribution is typical and is formally proven in the Bitcoin Backbone series of papers [60, 61].

Throughout this work, we will be interested in executions in which our distributions behave nicely. We term these executions *typical* [60]. The properties we care about pertain to the values attained by X, Y, Z as well as the collision-resistance and unpredictability of the Random Oracle.

Definition 36 (Typical Execution)**.** *An execution is* (ϵ, λ)-typical *if it has an* (ϵ, λ)-typical query distribution *and no* collisions *or* predictions *have occurred.*

The following theorem is essential for our development and is proven in the backbone series of papers [60, 61].

Theorem 6 (Typicality)**.** *An execution is (ϵ, λ)-typical with overwhelming probability in $\epsilon^2\lambda$.*

We introduce some additional definitions that are useful for Δ delays. In the Δ-bounded delay model, periods of silence that corresponded to an honest party succeeding in a uniquely successful round now correspond to a sequence of Δ consecutive rounds at the beginning of which only one honest party is successful. The intuitive reason is that the adversary can delay any other successful query diffusion message within that same Δ period and make the two messages coincide in the view of receiving honest parties, causing disagreement. As such, a useful concept will be to define an *isolated (uniquely) successful* round as a round that is (uniquely) successful and is followed by a silence of duration Δ. We also introduce the respective indicator random variables X'_i, Y'_i for Δ-isolated (uniquely) successful rounds.

Definition 37 (Δ-isolated (Uniquely) Successful Round)**.** *A round r is a* Δ-isolated successful round *if $X_r = 1$ and for all $r < r' < r + \Delta$ it holds that $X_{r'} = 0$. In that case we set $X'_r = 1$; otherwise, we set $X'_r = 0$.*

A round r is a Δ-isolated uniquely successful round *if $Y_r = 1$ and for all $r < r' < r + \Delta$ it holds that $X_{r'} = 0$. In that case we set $Y'_r = 1$; otherwise, we set $Y'_r = 0$.*

These random variables have the following expectations for every round r:

- $\mathbb{E}[X'_r] = f(1-f)^{\Delta-1} \geq f[1-(\Delta-1)f]$
- $\mathbb{E}[Y'_r] \geq f(1-f)^{2\Delta-1} \geq f[1-(2\Delta-1)f]$

In this case, we also have to strengthen the honest majority assumption to accommodate for the Δ delay:

Definition 38 (Δ Honest Majority)**.** *We say that a population has* honest majority *with* wait time $\lambda \in \mathbb{N}$ *under a Δ-bounded delay with wait if*

$$t \leq (1-\delta)(n-t)$$

where

$$\delta > 5(\epsilon + 2\Delta f + \frac{2\Delta}{f})\,.$$

Typicality of queries is strengthened as follows:

Definition 39 (Δ Typical Query Distribution)**.** *An execution has* $(\epsilon, \lambda, \Delta)$-typical query distribution *with parameters $\epsilon \in (0, 1)$ (the* Chernoff error*), integer $\lambda \geq \frac{2}{f}$ (the* wait time*) and integer Δ (the* delay*), if for any set of rounds S with $|S| \geq \lambda$ it holds that:*

- $(1-\epsilon)\,\mathbb{E}[X'(S)] < X'(S), X(S) < (1+\epsilon)\,\mathbb{E}[X(S)]$
- $(1-\epsilon)\,\mathbb{E}[Y'(S)] < Y'(s)$
- $Z(S) < \mathbb{E}[Z(S)] + \epsilon\,\mathbb{E}[X'(S)]$

Typical executions are then defined as before and the standard theorem holds:

Theorem 7 (Δ Typicality)**.** *An execution is* $(\epsilon, \lambda, \Delta)$*-typical with overwhelming probability in* $\epsilon^2 f^2 (1-f)^{4\Delta-2}\lambda$*.*

Before we get to ordering transactions, consider momentarily the classical problem of a one-bit agreement [97] known as the Byzantine Agreement problem. In the Byzantine Agreement problem, each party boots up with one, potentially different, *input bit* each, either 0 or 1, in its state. The parties wish to coordinate among each other in order to output a single *output bit*. The protocol has *agreement* if all honest parties always give output bits consistent with each other (i.e., if one honest party outputs b, all do). The protocol has *validity* if, when all honest parties are given the same input bit, the output bit matches the input bit. Agreement or validity alone are easy to achieve. An example of a protocol that has only agreement asks all parties to always output 0. An example of a protocol that has only validity asks all parties to simply output their input. The challenge is to achieve both validity and agreement. Contrary to the classical setting, our setting here is different, because parties are anonymous, unauthenticated, and their count is unknown. As such, any attempt by the honest parties to report their inputs and coordinate among each other can be subverted by an adversary who injects messages that report honest-looking input bits, perhaps differently to each party, splitting their view.

Instead, proof-of-work must be leveraged. One approach that doesn't work, but gives intuition about how proof-of-work can be leveraged is as follows. Initially, each party boots with an input bit b. The party repeatedly attempts to find proof-of-work solutions ctr_i for the message $b \parallel r_i$ where r_i is sampled uniformly at random from $\{0,1\}^\kappa$ to ensure high entropy. Once some proof-of-work is found, the solution $b \parallel r_i \parallel ctr_i$ is broadcast to the network and the process is repeated for as many r_i as possible. After λ rounds have passed, the parties look at the proof-of-work solutions that have been diffused on the network throughout the execution and count the votes for bit 0 and bit 1. If all honest parties are given the same input, then they will all produce votes in favour of that input. In *expectation*, the count of these votes will be more than whatever the adversary produces, thereby achieving validity. However, *agreement* is not achieved because, if half the honest parties are given input 0 and the other half are given input 1, the adversary can create proof-of-work only in favour of the bit 0 and hand over that work only to half the parties, causing disagreement. This lack of agreement hints towards the need to *chain* proof-of-work, incorporating each previous solution as part of the message for each future attempt, and continuously work on top of the most proof-of-work chain that exists on the network, giving rise to *blockchains*. We describe this process in the next section.

2.7.3 Chains and Ledgers

Blockchains are finite block sequences obeying the *blockchain property*: that in every block in the chain there exists a pointer to its previous block. We denote a blockchain (or simply *chain*) by C. A special block generated at the beginning of the protocol execution called the *genesis* block $\mathcal{G}$ is known by all parties. Every valid chain must begin with the genesis block. We call such a chain *anchored*.

Given chains $\mathsf{C}_1, \mathsf{C}_2$ and block B we concatenate them as $\mathsf{C}_1\mathsf{C}_2$ or $\mathsf{C}_1 B$. $\mathsf{C}_2[0]$ must point to $\mathsf{C}_1[-1]$ and B must point to $\mathsf{C}_1[-1]$. The *id* function on a block returns the hash of the block header data $\mathsf{id} = H(ctr, x, h')$.

In blockchain protocols, each honest party P maintains a currently adopted chain. We denote $\mathsf{C}^P[t]$ the chain adopted by party P at slot t.

A *ledger* (denoted in bold-face, e.g. $\mathbf{L}$) is a mechanism for maintaining a sequence of transactions, often stored in the form of a blockchain.

We call a *ledger state* a concrete sequence of transactions $\mathsf{tx}_1, \mathsf{tx}_2, \ldots$ stored in the *stable* part of a ledger $\mathbf{L}$, typically as viewed by a particular party. Hence, in every blockchain-based ledger $\mathbf{L}$, every fixed chain C defines a concrete ledger state by applying the interpretation rules given as a part of the description of $\mathbf{L}$ (for example, the ledger state is obtained from the blockchain by dropping the last k blocks and serializing the transactions in the remaining blocks). We maintain the typographic convention that a ledger state (e.g. L) always belongs to the bold-face ledger of the same name (e.g. $\mathbf{L}$). We denote by $\mathbf{L}^P[t]$ the ledger state of a ledger $\mathbf{L}$ as viewed by a party P at the beginning of a time slot t, and by $\check{\mathbf{L}}^P[t]$ the complete state of the ledger (at time t) including all pending transactions that are not stable yet.

For a ledger $\mathbf{L}$ that satisfies persistence at time t, we denote by $\mathbf{L}^\cup[t]$ (resp. $\mathbf{L}^\cap[t]$) the sequence of transactions that are seen as included in the ledger by *at least one* (resp., *all*) of the honest parties. Finally, $\mathsf{length}(\mathbf{L})$ denotes the length of the ledger $\mathbf{L}$, i.e., the number of transactions it contains.

2.8 Blockchain Protocols

2.8.1 Blockchain Backbone

The Bitcoin protocol has been analyzed in a series of works [60, 61, 122, 15]. Here, we give an overview.

The Bitcoin protocol for the synchronous static difficulty model is illustrated in Algorithm 16. Every honest party runs the protocol. Each party maintains a chain C which is initialized as the empty sequence [3]. Each party uses its q Random Oracle queries in each round in an attempt to mine a new block which extends its currently adopted chain. The block points to the tip of the currently adopted blockchain and contains any transactions that the party receives from its environment. We say that the environment attempts to *inject* into the honest party, which, in turn, attempts to place them on the newly mined block. The environment's choice of transactions here plays the role of the mempool.

When an honest party successfully finds a proof-of-work for its candidate block, it appends that block to its adopted blockchain and broadcasts the new blockchain to the network.

When it receives a blockchain from the network, it first checks to see if it is valid. Validation involves checking that each of its blocks has proof-of-work, that it has the blockchain property, and that its transaction sequences satisfies the validity language. It adopts it if it has more blocks than its currently adopted chain. This procedure, called $\mathsf{maxvalid}$, is illustrated in Algorithm 17, where the valid predicate ensures that a chain respects proof-of-work and its contents belong to the validity language.

Chains maintained by honest parties running the backbone protocol in an honest majority setting satisfy the following three properties.

[3]this is a formalism which in practice is replaced with a sequence starting with a well-known high entropy block, the Genesis block $\mathcal{G}$; the two formulations are theoretically equivalent

Algorithm 16 The backbone protocol

```
1: $\mathsf{C} \leftarrow \varepsilon$
2: $st \leftarrow \varepsilon$
3: $round \leftarrow 1$
4: function $\mathsf{Backbone}(1^\lambda)$
5:     $\tilde{\mathsf{C}} \leftarrow \mathsf{maxvalid}(\mathsf{C}, \text{any chain } \mathsf{C}' \text{ received from the network})$
6:     $\langle st, x \rangle \leftarrow I(st, \tilde{\mathsf{C}}, round, \text{Input}(), \text{Receive}())$
7:     $\mathsf{C}_{\mathsf{new}} \leftarrow \mathsf{pow}(x, \tilde{\mathsf{C}})$
8:     if $\mathsf{C} \neq \mathsf{C}_{\mathsf{new}}$ then
9:         $\mathsf{C} \leftarrow \mathsf{C}_{\mathsf{new}}$
10:        Diffuse($\mathsf{C}$)
11:    else
12:        Diffuse($\perp$)
13:    end if
14:    $round \leftarrow round + 1$
15: end function
```

Algorithm 17 The *maxvalid* algorithm which chooses the longest valid chain

```
1: function $\mathsf{maxvalid}(\tilde{C})$
2:     $\mathsf{C} \leftarrow \mathcal{C}[0]$
3:     for $\mathsf{C}' \in \tilde{C}$ do
4:         if $\mathsf{valid}(\mathsf{C}') \wedge |\mathsf{C}'| > |\mathsf{C}|$ then
5:             $\mathsf{C} \leftarrow \mathsf{C}'$
6:         end if
7:     end for
8:     return $\mathsf{C}$
9: end function
```

The *chain growth* property states that the chain of an honest party will keep growing at a certain rate α. Because honest parties always extend the longest chain, this property holds even in executions of dishonest majority.

Definition 40 (Chain Growth)**.** *An execution has* chain growth *with parameters $\alpha \in \mathbb{R}$ (the* chain velocity*) and $s \in \mathbb{N}$ (the* chunk size*) if for all honest parties P and for all rounds r the following holds. If P has adopted chain C at round r, then for every round $r' > r + s$ the chain C' adopted by P at round r' satisfies:*

$$|\mathsf{C}'| \geq |\mathsf{C}| + \tau s\,.$$

The *chain quality* property states that any large enough ($\geq s$) chunk of an honestly adopted chain will always contain some (ρ) honest blocks.

Definition 41 (Chain Quality)**.** *An execution has* chain quality *with parameters ρ (the* chain quality parameter*) and $\ell \in \mathbb{N}$ (the* chunk size*) if for all honest parties P and for all rounds r during which the party has adopted chain C the following holds. For any i, j such that $|\mathsf{C}[i:j]| = \ell$, the block set $\mathsf{C}' = \{B \in \mathsf{C}[i{:}j] : B \text{ was honestly generated}\}$ satisfies:*

$$\frac{|\mathsf{C}'|}{\ell} \geq \rho\,.$$

The *common prefix* property states that the chains of two honest parties cannot deviate much. In particular, they will share a large common prefix and can only differ by up to k blocks near their ends.

Definition 42 (Common Prefix)**.** *An execution has* common prefix *with parameter k (the* common prefix parameter*) if for all honest parties P_1, P_2 and rounds $r_1 \leq r_2$ the following holds. If P_1 has adopted C_1 during round r_1 and P_2 has adopted C_2 during round r_2, then:*

$$\mathsf{C}_1[:-k] \preceq \mathsf{C}_2\,.$$

The next three theorems establish that these three properties hold in typical executions and have been proven in the backbone series of papers. These theorems hold in both the synchronous and the Δ-bounded delay network models [122] as well as in the static [60] and variable difficulty settings [61]. We state them here without proof.

In the synchronous model it holds that [60]:

Theorem 8 (Chain Growth)**.** *A typical execution satisfies* Chain Growth *with velocity $\alpha = (1-\epsilon)f$ and chunk size $s \geq \lambda$.*

Theorem 9 (Chain Quality)**.** *A typical execution satisfies* Chain Quality *with quality $\rho = 1 - (1 + \frac{\delta}{2})\frac{t}{n-t} - \frac{\epsilon}{1-\epsilon}$ and chunk size $\ell \geq 2\lambda f$.*

Theorem 10 (Common Prefix)**.** *A typical execution satisfies* Common Prefix *with common prefix parameter $k \geq 2\lambda f$.*

With the addition of a Δ-bounded delay it holds that [58]:

Theorem 11 (Chain Growth)**.** *A typical execution satisfies* Chain Growth *with velocity $\alpha = (1-\epsilon)f(1-f)^{\Delta-1}$ and chunk size $s \geq \lambda$.*

Theorem 12 (Chain Quality)**.** *A typical execution satisfies* Chain Quality *with quality $\rho = 1 - \frac{1}{(1-\epsilon)(1-f)^{\Delta}}\frac{t}{n-t} - \frac{e}{1-\epsilon}(1 + \frac{\Delta}{\lambda})$ and chunk size $\ell \geq 2\lambda f + 2\Delta f$.*

Theorem 13 (Common Prefix)**.** *A typical execution satisfies* Common Prefix *with $k \geq 2\lambda f + 2\Delta$.*

In the variable difficulty synchronous case it holds that [61]:

Theorem 14 (Chain Growth)**.** *A typical execution satisfies* Chain Growth *with velocity $\alpha = (1-\epsilon)f$ and chunk size $s \geq \lambda$.*

Theorem 15 (Chain Quality)**.** *A typical execution satisfies* Chain Quality *with quality $\rho = \delta - 2\epsilon - \theta f \geq \frac{\delta}{2}$. and chunk size $\ell \geq \frac{\theta\gamma m}{8\tau}$.*

Theorem 16 (Common Prefix)**.** *A typical execution satisfies* Common Prefix *with* $k \geq \frac{\theta\gamma m}{4\tau}$.

Remark 4 (Impossibility of full semi-synchrony)**.** *Revisiting the Δ-bounded delay model, a folklore observation that has not appeared in the litature is that blockchain protocols are impossible to obtain in the fully semi-synchronous setting where* no *conditions are imposed on Δ, because of the anonymous nature of the model. This impossibility stems from the fact that n is unknown to the honest parties. To see this, consider an honest majority execution in which an adversary controls $t = (1-\delta)(n-t)$ parties for some $\delta > 0$. If the honest parties take a decision of transaction acceptance within some time Δ, then that Δ can be used as network delay in which the messages of a percentage larger than δ of the honest parties are delayed. That setting is then indistinguishable to the honest parties from a setting in which the adversary controls the majority of the parties $t > n - t$, as there is an honest percentage which is effectively eclipsed. This is the case regardless of which solution is used to approach the problem of consensus – whether it is through blockchains or other means. Standard dishonest majority attacks therefore become possible avenues to break the protocol. Hence, the Δ-bounded delay setting in which Δ is unknown but some conditions are imposed on it is the best possible model we can hope for, as further relaxation would not allow for a solution, as long as dishonest majority breaks the protocol. The model can only be improved by relaxing, but not altogether removing, the conditions.*

2.8.2 Ouroboros

The protocol operates (and was analyzed) in the synchronous model with semi-adaptive corruptions. In each slot, each of the parties can determine whether she qualifies as a so-called *slot leader* for this slot. The event of a particular party becoming a slot leader occurs with a probability proportional to the stake controlled by that party and is independent for two different slots. It is determined by a public, deterministic computation from the stake distribution and so-called *epoch randomness* (we will discuss shortly where this randomness comes from) in such a way that for each slot, exactly one leader is elected.

If a party is elected to act as a slot leader for the current slot, she is allowed to create, sign, and broadcast a block (containing transactions that move stake among stakeholders). Parties participating in the protocol are collecting such valid blocks and always update their current state to reflect the longest chain they have seen so far that did not fork from their previous state by too many blocks into the past.

Multiple slots are collected into *epochs*, each of which contains $R \in \mathbb{N}$ slots. The security arguments in [89] require $R \geq 10k$ for a security parameter k; we will consider $R = 12k$ as additional $2k$ slots in each epoch will be useful for our construction. Each epoch is indexed by an index $j \in \mathbb{N}$. During an epoch j, the stake distribution that is used for slot leader election corresponds to the distribution recorded in the ledger up to a particular slot of epoch $j-1$, chosen in a way that guarantees that by the end of epoch $j-1$, there is consensus on the chain up to this slot. (More concretely, this is the latest slot of epoch $j-1$ that appears in the first $4k$ out of its total R slots.) Additionally, the *epoch randomness* η_j for epoch j is derived during the epoch $j-1$ via a *guaranteed-output delivery coin tossing* protocol that is executed by the epoch slot leaders, and is available after $10k$ slots of epoch $j-1$ have passed.

In our treatment, we will refer to the relevant parts of the above-described protocol as follows:

$\mathsf{GetDistr}(j)$ returns the stake distribution SD_j to be used for epoch j, as recorded in the chain up to slot $4k$ of epoch $j-1$;

$\mathsf{GetRandomness}(j)$ returns the randomness η_j for epoch j as derived during epoch $j-1$;

$\mathsf{ValidateConsensusLevel}(\mathsf{C})$ checks the consensus-level validity of a given chain C: it verifies that all block hashes are correct, signatures are valid and belong to eligible slot leaders;

$\mathsf{PickWinningChain}(\mathsf{C}, \mathcal{C})$ applies the chain-selection rule: from a set of chains $\{\mathsf{C}\} \cup \mathcal{C}$ it chooses the longest one that does not fork from the current chain C more than k blocks in the past;

$\mathsf{SlotLeader}(U, j, sl, \mathsf{SD}_j, \eta_j)$ determines whether a party U is elected a slot leader for the slot sl of epoch j, given stake distribution SD_j and randomness η_j.

Moreover, the function $\mathsf{EpochIndex}$ (resp. $\mathsf{SlotIndex}$) always returns the index of the current epoch (resp. slot), and the event $\mathsf{NewEpoch}$ (resp. $\mathsf{NewSlot}$) denotes the start of a new epoch (resp. slot). Since we use these functions in a black-box manner, our construction can be readily adapted to PoS protocols with a similar structure that differ in the details of these procedures.

Ouroboros was shown in [89] to achieve both persistence and liveness under the following assumptions: (1) synchronous communication; (2) $2R$-semi-adaptive corruptions; (3) majority of stake in the stake distribution for each epoch is always controlled by honest parties during that epoch.

Chapter 3

Proofs of Proof-of-Work

In this chapter, we put forth a cryptographic security definition for Non-Interactive Proofs of Proof-of-Work protocols which describes what such a synchronization protocol must achieve. We then construct a protocol which solves the problem and requires sending only a logarithmic number of blocks from the chain. We construct a protocol which can synchronize recent blocks, a *suffix proof* protocol which we term the *charity* NIPoPow. We analyze the security and succinctness of our protocol. We show a simple addition to the suffix proofs protocol which allows synchronizing any part of the blockchain that the client may be interested in, the *infix proofs* protocol. We give formal proofs of security and succinctness, provide concrete parameters for the implementation of our scheme, present applications beyond superlight clients including cross-chain applications, propose a mechanism with which our scheme can be deployed to existing cryptocurrencies without a fork.

In this chapter, we treat superblock NIPoPoWs in the *synchronous* and *static difficulty* backbone setting. We relax both of these assumptions in Chapter 5.

3.1 Definitions

Extending the Backbone model. The entities on the blockchain network are of 3 kinds: (1) Miners, who try to mine new blocks on top of the longest known blockchain and broadcast them as soon as they are discovered. Miners commit new transactions they receive from clients. (2) Full nodes, who maintain the longest blockchain without mining and also act as the provers in the network. (3) Verifiers or stateless clients, who do not store the entire blockchain, but instead connect to provers and ask for proofs in regards to which blockchain is the largest. The verifiers attempt to determine the value of a predicate on these chains, for example whether a particular payment has been finalized.

Our main challenge is to design a protocol so that clients can sieve through the responses they receive from the network and reach a conclusion that should never disagree with the conclusion of a full node who is faced with the same objective and infers it from its local blockchain state.

Blockchain blocks are generated by including the following data in them: *ctr*, the nonce used to achieve the proof-of-work; x the Merkle tree [112] root of the transactions confirmed in this block; and *interlink* [83], a vector containing pointers to previous blocks, including the id of the previous block. The *interlink* data structure contains pointers to more blocks than just the previous block. We will explain this further in Section 3.2. Given two hash functions H and G modelled as random oracles, the id of a block is defined as $\mathsf{id} = H(ctr, G(x, \mathsf{interlink}))$. In bitcoin's case, both H and G would be SHA256.

The prover and verifier model. In our protocol, the nodes include a *proof* along with their responses to clients. We need to assume that clients are able to connect to at least one correctly functioning node (i.e., that they cannot be eclipsed from the network [67, 5]). Each client makes the same request to every node, and by verifying the proofs the client identifies the correct response. Henceforth we will call clients *verifiers* and nodes *provers*.

The prover-verifier interaction is parameterized by a predicate (e.g. "the transaction tx is committed in the blockchain"). The predicates of interest in our context are predicates on the active blockchain. Some of the predicates are more suitable for succinct proofs than others. We focus our attention in *stable* predicates having the property that all honest miners share their view of them in a way that is updated in a predictable manner, with a truth-value that persists as the blockchain grows (an example of an unstable predicate is e.g., the least significant bit of the hash of last block). Following the backbone work, we wait for k blocks to bury a block before we consider it *confirmed* and thereby the predicates depending on it stable (k is the common prefix parameter).

In our setting, for a given predicate Q, several provers (including adversarial ones) will generate proofs claiming potentially different truth values for Q based on their claimed local longest chains. The verifier receives these proofs and accepts one of the proofs, determining the truth value of the predicate. We denote a *blockchain proof protocol* for a predicate Q as a pair (P, V) where P is the *prover* and V is the *verifier*. P is a PPT algorithm that is spawned by a full node when they wish to produce a proof, accepts as input a full chain C and produces a proof π as its output. V is a PPT algorithm which is spawned at some round (having only Genesis), receives a pair of proofs (π_A, π_B) from both an honest party and the adversary and returns its decision $d \in \{T, F\}$ before the next round and terminates. The honest miners produce proofs for V using P, while the adversary produces proofs following some arbitrary strategy. Before we introduce the security properties for blockchain proof protocols we introduce some necessary notation for blockchains.

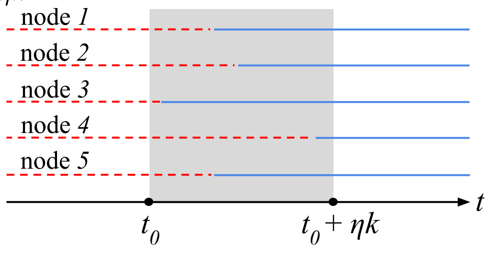
3.1.1 Provable chain predicates

Our aim is to prove statements about the blockchain, such as "The transaction tx is included in the current blockchain" without transmitting all block headers. We consider a general class of predicates that take on values *true* or *false*. Since a Bitcoin-like blockchain can experience delays and intermittent forks, not all honest parties will be in exact agreement about the entire chain. However, when all honest parties are in agreement about the truth value of the predicate, we require that the verifier also arrives at the same truth value.

To aid the construction of our proofs, we focus on predicates that are *monotonic*; they start with the value *false* and, as the blockchain grows, can change their value to *true* but not back.

Definition 43. *(Monotonicity) A chain predicate $Q(\mathsf{C})$ is* monotonic *if for all chains C and for all blocks B we have that $Q(\mathsf{C}) \Rightarrow Q(\mathsf{C}B)$.*

Figure 3.1: The truth value of a fixed predicate Q about the blockchain, as seen from the point of view of 5 honest nodes, drawn on the vertical axis, over time, drawn as the horizontal axis. The truth value evolves over time starting as *false* at the beginning, indicated by a dashed red line. At some point in time t_0, the predicate is ready to be evaluated as *true*, indicated by the solid blue line. The various honest nodes each realize this independently over a period of ηk duration, shaded in gray. The predicate remains *false* for everyone before t_0 and *true* for everyone after $t_0 + \eta k$.



Additionally, we require that our predicates only depend on the *stable* portion of the blockchain, blocks that are buried under k subsequent blocks. This ensures that the value of the predicate will not change due to a blockchain reorganization.

Definition 44. *(Stability) Parameterized by $k \in \mathbb{N}$, a chain predicate Q is k-*stable *if its value only depends on the prefix* $\mathsf{C}[: -k]$.

3.1.2 Desired properties

We now define two desired properties of a non-interactive blockchain proof protocol, *succinctness* and *security*.

Definition 45. *(Security) A* blockchain proof protocol (P, V) *about a predicate Q is* secure *if for all environments and for all PPT adversaries $\mathcal{A}$ and for all rounds $r \geq \eta k$, if V receives a set of proofs $\mathcal{P}$ at the beginning of round r, at least one of which has been generated by the honest prover P, then the output of V at the end of round r has the following constraints:*

- *If the output of V is* false*, then the evaluation of $Q(\mathsf{C})$ for* all *honest parties must be* false *at the end of round $r - \eta k$.*
- *If the output of V is* true*, then the evaluation of $Q(\mathsf{C})$ for* all *honest parties must be* true *at the end of round $r + \eta k$.*

Some explanation is needed for the rationale of the above definition. The parameter η is borrowed from the Backbone [60] work and indicates the rate at which new blocks are produced, i.e., the number of rounds needed on average to produce a block. If the scheme is secure, this means that the output of the verifier should match the output of a *potential honest full node*. However, in various executions, not all potential honest full node behaviors will be instantiated. Therefore, we require that, if the output of the proof verifier is *true* then, consistently with honest behavior, all other honest full nodes will converge to the value *true*. Conversely, if the output of the proof verifier is *false* then, consistently with honest behavior, all honest full nodes must have indicated *false* sufficiently long in the past. The period

ηk is the period needed for obtaining sufficient confirmations (k) in a blockchain system. A predicate's value has the potential of being *true* as seen by an honest party starting at time t_0. Before time t_0, all honest parties agree that the predicate is *false*. It takes ηk time for all parties to agree that the predicate is *true*, which is certain after time $t_0 + \eta k$. The adversary may be able to convince the verifier that the predicate has any value during the period from t_0 to $t_0 + \eta k$. However, our security definition mandates that before time t_0 the verifier will necessarily output *false* and after time $t_0 + \eta k$ the verifier will necessarily output *true*.

Definition 46. *(Succinctness) A* blockchain proof protocol (P, V) *about a predicate* Q *is* succinct *if for all PPT provers* $\mathcal{A}$*, any proof* π *produced by* $\mathcal{A}$ *at some round* r*, the verifier* V *only reads a* $O(polylog(r))$*-sized portion of* π*.*

It is easy to construct a *secure but not succinct* protocol for any computable predicate Q: The prover provides the entire chain C as a proof and the verifier simply selects the longest chain: by the *common-prefix property* of the backbone protocol (c.f. [60]), this is consistent with the view of every honest party (as long as Q depends only on a *prefix* of the chain, as we explain in more detail shortly). In fact this is how widely-used cryptocurrency clients (including SPV clients) operate today.

It is also easy to build *succinct but insecure* clients: The prover simply sends the predicate value directly. This is roughly what hosted wallets do [28].

The challenge we will solve is to provide a non-interactive protocol that at the same time achieves security and succinctness over a large class of useful predicates. We call this primitive a NIPoPoWs. Our particular instantiation for NIPoPoWs is a *superblock-based NIPoPoW construction.*

3.2 Consensus layer support

3.2.1 The interlink pointers data structure

In order to construct our protocol, we rely on the *interlink data structure* [83]. This is an additional hash-based data structure that is proposed to be included in the header of each block. The interlink data structure is a skip-list [127] that makes it efficient for a verifier to process a sparse subset of the blockchain, rather than only consecutive blocks.

Valid blocks satisfy the proof-of-work condition: $id \leq T$, where T is the mining target. In this chapter, we work in the static difficulty, and so make the simplifying assumption that T is constant. Some blocks will achieve a lower id. If $id \leq \frac{T}{2^\mu}$ we say that the block is of level μ. All blocks are level 0. Blocks with level μ are called *μ-superblocks.* μ-superblocks for $\mu > 0$ are also $(\mu - 1)$-superblocks. The level of a block is given:

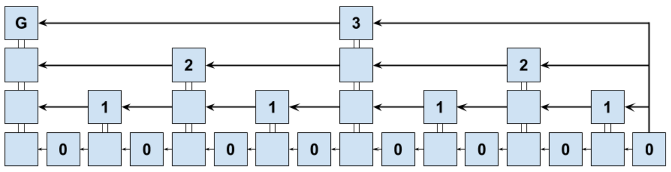
Definition 47 (Level)**.** *Let B be a valid block. Its* level μ *is defined as:*

$$\mu = \text{level}(B) = \lfloor \log(T) - \log(\mathsf{id}(\mathsf{B})) \rfloor \ .$$

By convention for Gen we set $\mu = \infty$.

Definition 48 (Superblock)**.** *A block of level μ is called a μ-*superblock*.*

Figure 3.2: The hierarchical blockchain. Higher levels have achieved a lower target (higher difficulty) during mining. All blocks are connected to the genesis block G.



Observe that in a blockchain protocol execution it is expected 1/2 of the blocks will be of level 1; 1/4 of the blocks will be of level 2; 1/8 will be of level 3; and $1/2^{\mu}$ blocks will be of level μ. In expectation, the number of superblock levels of a chain C will be $\Theta(\log(\mathsf{C}))$ [83]. Figure 3.2 illustrates the blockchain superblocks starting from level 0 and going up to level 3 in case these blocks are distributed exactly according to expectation. Here, each level contains half the blocks of the level below.

We wish to connect the blocks at each level with a *previous block* pointer pointing to the most recent block of the same level. These pointers must be included in the data of the block so that proof-of-work commits to them. As the level of a block cannot be prediced before its proof-of-work is calculated, we extend the *previous block id* structure of classical blockchains to be a vector, the *interlink vector*. The interlink vector points to the most recent preceding block of every level μ. Genesis is of infinite level and hence a pointer to it is included in every block. The number of pointers that need to be included per block is in expectation $\log(|\mathsf{C}|)$.

The algorithm for this construction is shown in Algorithm 18 and is borrowed from [83]. The interlink data structure turns the blockchain into a skiplist-like [127] data structure.

The updateInterlink algorithm accepts a block B', which already has an interlink data structure defined on it. The function evaluates the interlink data structure which needs to be included as part of the next block. It copies the existing interlink data structure and then modifies its entries from level 0 to $\mathsf{level}(B')$ to point to the block B'.

Algorithm 18 updateInterlink

```
1: function updateInterlink($B'$)
2:     interlink ← $B'$.interlink
3:     for $\mu = 0$ to $level(B')$ do
4:         interlink[$\mu$] ← id($B'$)
5:     end for
6:     return interlink
7: end function
```

We will only care about *whether* a block contains a pointer to a previous block, not the positions of these pointers within the block. An optimization we can readily perform on this algorithm is to treat the interlink vector as a *set* and remove duplicates, as illustrated in Algorithm 19. This optimization achieves a constant saving of about 50% in expectation [77].

Algorithm 19 The updateInterlinkSet algorithm which updates the interlink set

```
1: function updateInterlinkSet($B'$)
2:     interlinkSet ← $\{H(B')\}$
3:     for $H(B) \in B'$.interlink do
4:         if $level(B) > level(B')$ then
5:             interlinkSet ← interlinkSet $\cup \{H(B)\}$
6:         end if
7:     end for
8:     return interlinkSet
9: end function
```

Traversing the blockchain. As we have now extended blocks to contain multiple pointers to previous blocks, if certain blocks are omitted from the middle of a chain we will obtain a subchain, as long as the *blockchain property* is maintained (i.e., that each block must contain an interlink pointer to its previous block in the sequence).

Blockchains are sequences, so we will use the set notation defined in Chapter 1. We note that, when manipulating blockchains in this manner, it is important to remember when the blockchain property is maintained. In particular, $\mathsf{C}_1 \cup \mathsf{C}_2$ and $\mathsf{C}_1 \cap \mathsf{C}_2$ as well as chains filtered through set-builder notation, while sequences of blocks, may not always be chains since pointers may be missing. If $\mathsf{C}_1[0] = \mathsf{C}_2[0]$ and $\mathsf{C}_1[-1] = \mathsf{C}_2[-1]$, we say the chains $\mathsf{C}_1, \mathsf{C}_2$ *span* the same block range.

Definition 49 (Lowest Common Ancestor). *The* lowest common ancestor *of chains* C_1*,* C_2 *is defined as*

$$LCA(\mathsf{C}_1, \mathsf{C}_2) = (\mathsf{C}_1 \cap \mathsf{C}_2)[-1]\,.$$

It will soon become clear that it is useful to construct a chain containing only the superblocks of another chain.

Definition 50 (Upchain). *Given* C *and level* μ*, the* upchain $\mathsf{C}{\uparrow}^{\mu}$ *is defined as* $\{B \in \mathsf{C} : level(B) \geq \mu\}$*.*

A chain containing only μ-superblocks is called a *μ-superchain*. It is also useful, given a μ-superchain C' to go back to the regular chain C. Given chains $\mathsf{C}' \subseteq \mathsf{C}$, the *downchain* $\mathsf{C}'{\downarrow}_{\mathsf{C}}$ is defined as $\mathsf{C}\{\mathsf{C}'[0]{:}\mathsf{C}'[-1]\}$. C is the *underlying chain* of C'. The underlying chain is often implied by context, so we will simply write $\mathsf{C}'{\downarrow}$. By the above definition, the $\mathsf{C}{\uparrow}$ operator is idempotent: $(\mathsf{C}{\uparrow}^{\mu}){\uparrow}^{\mu}= \mathsf{C}{\uparrow}^{\mu}$. Given a set of consecutive rounds $S = \{r, r+1, \cdots, r+j\} \subseteq \mathbb{N}$, we define $\mathsf{C}^S = \{B \in \mathsf{C} :$ B was generated during $S\}$.

To aid readability, we define the chain filtering operators $\uparrow$, $[\cdot]$, and $\{\cdot\}$ to have a higher precedence than $\cup, \cap$.

3.3 Non-interactive blockchain *suffix* proofs

In this section, we introduce our non-interactive suffix proofs. With foresight, we caution the reader that the non-interactive construction we present in this section is *insecure*. A small patch will later allow us to modify our construction to achieve security.

We allow provers to prove general predicates Q about the chain C. Among the predicates which are stable, in this section, we will limit ourselves to *suffix sensitive* predicates. We extend the protocol to support more flexible predicates (such as transaction inclusion, as needed for our applications) which are not limited to the suffix in Section 3.5.

Definition 51 (Suffix sensitivity)**.** *A chain predicate Q is called k-*suffix sensitive *if its value can be efficienty computed given the last k blocks of the chain.*

Example. In general our applications will require predicates that are not suffix-sensitive. However, as an example, consider the predicate "an Ethereum contract at address C has been initialized with code h at least k blocks ago" where h does not invoke the `selfdestruct` opcode. This can be implemented in a suffix-sensitive way because, in Ethereum, each block includes a Merkle Trie over all of the contract codes [35, 151], which cannot be changed after initialization. This predicate is thus also monotonic and k-stable. Any predicate which is both *suffix-sensitive* and *k-stable* must solely depend on data at block $\mathsf{C}[-k]$.

3.3.1 Construction

We next present a generic form of the verifier first and the prover afterwards. The generic form of the verifier works with any practical suffix proof protocol. Therefore, we describe the generic verifier first before we talk about the specific instantiation of our protocol. The generic verifier is given access to call a protocol-specific proof comparison operator $\leq_m$ that we define. We begin the description of our protocol by first illustrating the generic verifier. Next, we describe the prover specific to our protocol. Finally, we show the instantiation of the $\leq_m$ operator, which plugs into the generic verifier to make a concrete verifier for our protocol.

The generic verifier. The Verify function of our NIPoPoW construction for suffix predicates is described in Algorithm 20. The verifier algorithm is parameterized by a chain predicate Q and security parameters k, m; k pertains to the amount of proof-of-work needed to bury a block so that it is believed to remain stable (e.g., $k = 6$); m is a security parameter pertaining to the prefix of the proof, which connects the genesis block to the k-sized suffix. The verifier receives several proofs by different provers in a collection of proofs $\mathcal{P}$ at least one of which will be honest. Iterating over these proofs, it extracts the best.

Each proof is a chain. For honest provers, these are subchains of the adopted chain. Proofs consist of two parts, π and χ; $\pi\chi$ must be a valid chain; χ is the proof suffix; π is the prefix. We require $|\chi| = k$. For honest provers, χ is the last k blocks of the adopted chain, while π consists of a selected subset of blocks from the rest of their chain preceding χ. The method of choice of this subset will become clear soon.

Algorithm 20 The Verify algorithm for the NIPoPoW protocol

```
1: function $\mathsf{Verify}^{Q}_{m,k}(\mathcal{P})$
2:     $\tilde{\pi} \leftarrow$ (Gen)                                ▷ Trivial anchored blockchain
3:     for $(\pi, \chi) \in \mathcal{P}$ do                          ▷ Examine each proof $(\pi, \chi)$ in $\mathcal{P}$
4:         if $\mathsf{validChain}(\pi\chi) \wedge |\chi| = k \wedge \pi \geq_m \tilde{\pi}$ then
5:             $\tilde{\pi} \leftarrow \pi$
6:             $\tilde{\chi} \leftarrow \chi$                         ▷ Update current best
7:         end if
8:     end for
9:     return $\tilde{Q}(\tilde{\chi})$
10: end function
```

The verifier compares the proof prefixes provided to it by calling the $\geq_m$ operator. We will get to the operator's definition shortly. Proofs are checked for validity before comparison by ensuring $|\chi| = k$ and calling $\mathsf{validChain}$ which checks if $\pi\chi$ is an anchored blockchain.

At each loop iteration, the verifier compares the next candidate proof prefix π against the currently best known proof prefix $\tilde{\pi}$ by calling $\pi \geq_m \tilde{\pi}$. If the candidate prefix is better than the currently best known proof prefix, then the currently known best prefix is updated by setting $\tilde{\pi} \leftarrow \pi$. When the best known prefix is updated, the suffix $\tilde{\chi}$ associated with the best known prefix is also updated to match the suffix χ of the candidate proof by setting $\tilde{\chi} \leftarrow \chi$. While $\tilde{\chi}$ is needed for the final predicate evaluation, it is not used as part of any comparison, as it has the same size k for all proofs. The best known proof prefix is initially set to (Gen), the trivial anchored chain containing only the genesis block. Any well-formed proof compares favourably against the trivial chain.

After the end of the `for` loop, the verifier will have determined the best proof $(\tilde{\pi}, \tilde{\chi})$. We will later prove that this proof will necessarily belong to an honest prover with overwhelming probability. Since the proof has been generated by an honest prover, it is associated with an underlying honestly adopted chain C. The verifier then extracts the value of the predicate Q on the underlying chain. Note that, because the full chain is not available to the verifier, the verifier here must evaluate the predicate on the suffix. Because the predicate is suffix-sensitive, it is possible to do so. As a technical detail, we denote $\tilde{Q}$ the predicate which accepts only a k-suffix of a blockchain and outputs the same value that Q would have output if it had been evaluated on a chain with that suffix.

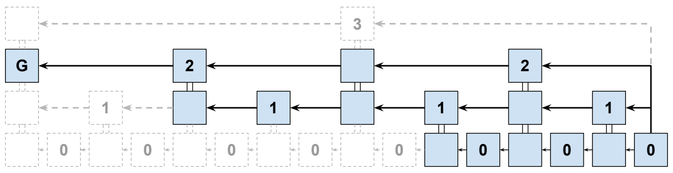
Algorithm 21 The Prove algorithm for the NIPoPoW protocol

```
1: function $\mathsf{Prove}_{m,k}(\mathsf{C})$
2:     $B \leftarrow \mathsf{C}[0]$                                    ▷ Genesis
3:     for $\mu = |\mathsf{C}[-k-1].\mathsf{interlink}|$ down to 0 do
4:         $\alpha \leftarrow \mathsf{C}[: -k]\{B :\}{\uparrow}^{\mu}$
5:         $\pi \leftarrow \pi \cup \alpha$
6:         if $m < |\alpha|$ then
7:             $B \leftarrow \alpha[-m]$
8:         end if
9:     end for
10:    $\chi \leftarrow \mathsf{C}[-k :]$
11:    return $\pi\chi$
12: end function
```

The concrete prover. The NIPoPoW prover construction is shown in Algorithm 21. The honest prover is supplied with an honestly adopted chain C and security parameters m, k and returns proof $\pi\chi$, which is a chain. The suffix χ is the last k blocks of C. The prefix π is constructed by selecting various blocks from $\mathsf{C}[: -k]$ and adding them to π, which consists of a number of blocks for every level μ from the highest level $|\mathsf{C}[-k].\mathsf{interlink}|$ down to 0. At the highest possible level at which at least m blocks exist, all these blocks are included. Then, inductively, for every superchain of level μ that is included in the proof, the suffix of length m is taken. Then the underlying superchain of level $\mu - 1$ spanning from this suffix until the end of the blockchain is also included. All the μ-superblocks which are within this range of m blocks will also be $(\mu - 1)$-superblocks and so we do not want to keep them in the proof twice (we use the union set notation to indicate this). Each underlying superchain will have $2m$ blocks in expectation and always at least m blocks. This is repeated until level $\mu = 0$ is reached. Note that no check is necessary to make sure the top-most level has at least m blocks, even though the verifier requires this. The reason is the following: Assume the blockchain has at least m blocks in total. Then, when a superchain of level μ has less than m blocks in total, these blocks will all be necessarily included into the proof by a lower-level superchain $\mu - i$ for some $i > 0$. Therefore, it does not hurt to add them to π earlier.

Figure 3.3 contains an example proof constructed for parameters $m = k = 3$. The top superchain level which contains at least m blocks is level $\mu = 2$. For the m-sized suffix of that level, 6 blocks of superblock level 1 are included to span the same range ($2m$ blocks at this level). For the last 3 blocks of the 1-superchain, blocks of level 0 spanning the same range are included (again $2m$ blocks at this level). Note that the superchain at a lower levels may reach closer to the end of the blockchain than a higher level. Level 3 was not used, as it does not yet have a sufficient number of blocks.

Figure 3.3: NIPoPoW *prefix* π for $m = 3$. It includes the Genesis block G, three 2-superblocks, six 1-superblocks, and six 0-blocks.



Algorithm 22 The algorithm implementation for the $\geq_m$ operator to compare two proofs in the NIPoPoW protocol parameterized with security parameter m. Returns *true* if the underlying chain of player A is deemed longer than the underlying chain of player B.

```
1: function best-arg$_m(\pi, b)$
2:     $M \leftarrow \{\mu : |\pi{\uparrow}^{\mu} \{b :\}| \geq m\} \cup \{0\}$          ▷ Valid levels
3:     return $\max_{\mu \in M}\{2^{\mu} \cdot |\pi{\uparrow}^{\mu} \{b :\}|\}$          ▷ Score for level
4: end function
5: operator $\pi_A \geq_m \pi_B$
6:     $b \leftarrow (\pi_{\mathcal{A}} \cap \pi_B)[-1]$          ▷ LCA
7:     return best-arg$_m(\pi_A, b) \geq$ best-arg$_m(\pi_B, b)$
8: end operator
```

The concrete verifier. The $\geq_m$ operator which performs the comparison of proofs is presented in Algorithm 22. It takes proofs π_A and π_B and returns *true* if the first proof is winning, or *false* if the second is winning. It first computes the LCA block b between the proofs. As parties A and B agree that the blockchain is the same up to block b, arguments will then be taken for the diverging chains after b. An *argument* is a subchain of a proof provided by a prover such that its blocks are after the LCA block b and they are all at the same level μ. The best possible argument from each player's proof is extracted by calling the best-arg$_m$ function. To find the best argument of a proof π given b, best-arg$_m$ collects all the indices μ which point to superblock levels that contain valid arguments after block b. Argument validity requires that there are at least m μ-superblocks following block b, which is captured by the comparison $|\pi{\uparrow}^{\mu} \{b :\}| \geq m$. 0 is always considered a valid level, regardless of how many blocks are present there. These level indices are collected into set M. For each of these levels, the score of their respective argument is evaluated by weighting the number of blocks by the level as $2^{\mu}|\pi{\uparrow}^{\mu} \{b :\}|$. The highest possible score across all levels is returned. Once the score of the best argument of both A and B is known, they are directly compared and the winner returned. An advantage is given to the first proof in case of a tie by making the $\geq_m$ operator favour the adversary $\mathcal{A}$.

Looking ahead, the core of the security argument will be that, given a block b, it will be difficult for a mining minority adversary to produce blocks descending from b faster than the honest party. This holds for blocks of any level.

3.4 Analysis

We now give a sketch indicating why our construction is secure. The fully formal security proof, together with a detail in the construction which ensures statistical *goodness* and is necessary for withstanding full 1/2 adversaries, appears in the later sections.

Theorem 17 (Security). *Assuming honest majority, the Non-interactive Proofs of Proof-of-Work construction for computable k-stable monotonic suffix-sensitive predicates is secure with overwhelming probability in κ.*

Sketch. Suppose an adversary produces a proof $\pi_{\mathcal{A}}$ and an honest party produces a proof π_B such that the two proofs cause the predicate Q to evaluate to different values, while at the same time all honest parties have agreed that the correct value is the one obtained by π_B. Because of Bitcoin's security, $\mathcal{A}$ will be unable to make these claims for an actual underlying 0-level chain.

We now argue that the operator $\leq_m$ will signal in favour of the honest parties. Suppose b is the LCA block between $\pi_{\mathcal{A}}$ and π_B. If the chain forks at b, there can be no more adversarial blocks after b than honest blocks after b, provided there are at least k honest blocks (due to the Common Prefix property). We will now argue that, further, there can be no more disjoint $\mu_{\mathcal{A}}$-level superblocks than honest μ_B-level superblocks after b.

To see this, let b be an honest block generated at some round r_1 and let the honest proof be generated at some round r_3. Then take the sequence of consecutive rounds $S = (r_1, \cdots, r_3)$. Because the verifier requires at least m blocks from each of the provers, the adversary must have m $\mu_{\mathcal{A}}$-superblocks in $\pi_{\mathcal{A}}\{b :\}$ which are not in $\pi_B\{b :\}$. Therefore, using a negative binomial tail bound argument, we see that $|S|$ must be long; intuitively, it takes a long time to produce a lot of blocks $|\pi_{\mathcal{A}}\{b :\}|$. Given that $|S|$ is long and that the honest parties have more mining power, they must have been able to produce a longer $\pi_B\{b :\}$ argument (of course, this comparison will have the superchain lengths weighted by $2^{\mu_{\mathcal{A}}}, 2^{\mu_B}$ respectively). To prove this, we use a binomial tail bound argument; intuitively, given a long time $|S|$, a lot of μ_B-superblocks $|\pi_B\{b :\}|$ will have been honestly produced.

We therefore have a fixed value for the length of the adversarial argument, a negative binomial random variable for the number of rounds, and a binomial random variable for the length of the honest argument. By taking the expectations of the above random variables and applying a Chernoff bound, we see that the actual values will be close to their means with overwhelming probability, completing the proof. □

We formalize the above proof sketch in the next sections.

Lastly, the following theorem illustrates that our proofs are succinct. Intuitively, the number of levels exchanged is logarithmic in the length of the chain, and the number of blocks in each level is constant. The formal proofs are included in the next section.

Theorem 18 (Optimistic succinctness). *In an optimistic execution, Non-Interactive Proofs of Proof-of-Work produced by honest provers are succinct with the number of blocks bounded by $4m \log(|\mathsf{C}|)$, with overwhelming probability in m.*

3.5 Non-interactive blockchain *infix* proofs

In the main body we have seen how to construct proofs for suffix predicates. As mentioned, the main purpose of that construction is to serve as a stepping stone for the construction of this section that presents a more useful class of proofs. This class of proofs allows proving more general predicates that can depend on multiple blocks even buried deep within the blockchain.

More specifically, the generalized prover for *infix proofs* allows proving any predicate $Q(\mathsf{C})$ that depends on a number of blocks that can appear anywhere within the chain (except the k suffix for stability). These blocks constitute a *subset* C' of blocks, the *witness*, which may not necessarily form a chain. This allows proving useful statements such as, for example, whether a transaction is confirmed. We next formally define the class of predicates that will be of interest.

Definition 52 (Infix sensitivity)**.** *A chain predicate $Q_{d,k}$ is* infix sensitive *if it can be written in the form*

$$Q_{d,k}(\mathsf{C}) = \begin{cases} \text{true}, \textit{ if } \exists \mathsf{C}' \subseteq \mathsf{C}[: -k] : |\mathsf{C}'| \leq d \wedge D(\mathsf{C}') \\ \text{false}, \textit{ otherwise} \end{cases}$$

where D is an arbitrary efficiently computable predicate such that, for any block sets $\mathsf{C}_1 \subseteq \mathsf{C}_2$ we have that $D(\mathsf{C}_1) \rightarrow D(\mathsf{C}_2)$.

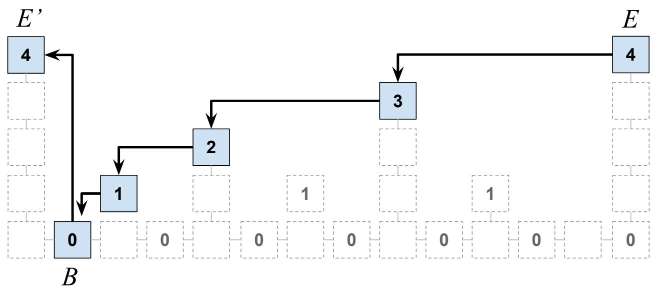
Note that C' is a blockset and may not necessarily be a blockchain. Furthermore, observe that for all blocksets $\mathsf{C}' \subseteq \mathsf{C}$ we have that $Q(\mathsf{C}') \rightarrow Q(\mathsf{C})$. This will allow us to later argue that adding more blocks to a blockchain cannot invalidate its witness.

Similarly to suffix-sensitive predicates, infix-sensitive predicates Q can be evaluated very efficiently. Intuitively this is possible because of their localized nature and dependency on the $D(\cdot)$ predicate which requires only a small number of blocks to conclude whether the predicate should be true.

Example. We next show how to express the predicate that asks whether a certain transaction with id *txid* has been confirmed as an infix sensitive predicate. We define the predicate D^{txid} that receives a single block and tests whether a transaction with id *txid* is included. The predicate $Q^{txid}_{1,k}$ is defined as in Definition 52 using the predicate D^{txid} and the parameter k which in this case determines the desired stability of the assertion that *txid* is included (e.g., $k = 6$). Q alone proves that a particular block is included in the blockchain. Some auxiliary data is supplied by the prover to aid the provability of transaction inclusion: the Merkle Tree proof-of-inclusion path to the transactions Merkle Tree root, similar to an SPV proof. This data is logarithmic in the number of transactions in the block and, hence, constant with respect to blockchain size. In case of a vendor awaiting transaction confirmation to ship a product, the proof that a certain transaction paid into a designated address for the particular order is sufficient. In this scheme it is impossible to determine whether the money has subsequently been spent in a future block, and so must only be used by the vendor holding the respective secret keys.

In the above example, note that if the verifier outputs *false*, this behavior will generally be inconclusive in the sense that the verifier could be outputting *false* either because the payment has not yet been confirmed or because the payment was never made.

Figure 3.4: An infix proof descend. Only blue blocks are included in the proof. Blue blocks of level 4 are part of π, while the blue blocks of level 1 through 3 are produced by followDown to get to the block of level 0 which is part of C'.



Algorithm 23 The Prove algorithm for infix proofs

```
function $\mathsf{ProveInfix}_{m,k}(\mathsf{C}, \mathsf{C}', \mathsf{height})$
    $aux \leftarrow \emptyset$
    $(\pi, \chi) \leftarrow \mathsf{Prove}_{m,k}(\mathsf{C})$                    ▷ Start with a suffix proof
    for $B \in \mathsf{C}'$ do
        for $E \in \pi$ do
            if $\mathsf{height}[E] \geq \mathsf{height}[B]$ then
                $aux \leftarrow aux \cup \mathsf{followDown}(E, B, \mathsf{height})$
                break
            end if
        end for
    end for
    return $(aux \cup \pi, \chi)$
end function
```

3.5.1 Construction

The construction of these proofs is shown in Algorithm 23. The infix prover accepts two parameters: The chain C which is the full blockchain and C' which is a sub-blockset of the blockchain and whose blocks are of interest for the predicate in question. The prover calls the previous suffix prover to produce a proof as usual. Then, having the prefix π and suffix χ of the suffix proof in hand, the infix prover adds a few auxiliary blocks to the prefix π. The prover ensures that these auxiliary blocks form a chain with the rest of the proof π. Such auxiliary blocks are collected as follows: For every block B of the subset C', the immediate previous (E') and next (E) blocks in π are found. Then, a chain of blocks R which connects E back to B is found by the algorithm $\mathsf{followDown}$. If E' is of level μ, there can be no other μ-superblock between B and E', otherwise it would have been included in π. Therefore, B already contains a pointer to E' in its interlink, completing the chain.

The way to connect a superblock to a previous lower-level block is implemented in Algorithm 24. Block B' cannot be of higher or equal level than E, otherwise it would be equal to E and the $\mathsf{followDown}$ algorithm would return. The algorithm proceeds as follows: Starting at block E, it tries to follow a pointer to as far as possible. If following the pointer surpasses B, then the procedure at this level is aborted and a lower level is tried, which will cause a smaller step within the skiplist. If a pointer was followed without surpassing B, the operation continues from the new block, until eventually B is reached, which concludes the algorithm.

Algorithm 24 The followDown function which produces the necessary blocks to connect a superblock E to a preceeding regular block B.

```
1: function followDown($E$, $B$, height)
2:     $aux \leftarrow \emptyset$; $\mu \leftarrow level(E)$
3:     while $E \neq B$ do
4:         $B' \leftarrow$ blockById[$E$.interlink[$\mu$]]
5:         if height[$B'$] $<$ height[$B$] then
6:             $\mu \leftarrow \mu - 1$
7:         else
8:             $aux \leftarrow aux \cup \{E\}$
9:             $E \leftarrow B'$
10:        end if
11:    end while
12:    return $aux$
13: end function
```

An example of the output of followDown is shown in Figure 3.4. This is a portion of the proof shown at the point where the superblock levels are at level 4. A descend to level 0 was necessary so that a regular block would be included in the chain. The level 0 block can jump immediately back up to level 4 because it has a high-level pointer.

The verification algorithm must then be modified as in Algorithm 25.

The algorithm works by calling the suffix verifier. It also maintains a blockDAG collecting blocks from all proofs (it is a DAG because *interlink* can be adversarially defined in adversarially mined blocks). This DAG is maintained in the blockById hashmap. Using it, ancestors uses simple graph search to extract the set of ancestor blocks of a block. In the final predicate evaluation, the set of ancestors of the best blockchain tip is passed to the predicate. The ancestors are included to avoid an adversary who presents an honest chain but skips the blocks of interest. In particular, such an adversary would work by including a complete suffix proof, but “forgetting” to include the blocks generated by followDown for the infix proof pertaining to blocks in C'.

Algorithm 25 The verify algorithm for the NIPoPoW infix protocol

```
function ancestors($B$, blockById)
    if $B = \text{Gen}$ then
        return $\{B\}$
    end if
    $\mathsf{C} \leftarrow \emptyset$
    for id $\in B$.interlink do
        if id $\in$ blockById then
            $B' \leftarrow$ blockById[id]
            $\mathsf{C} \leftarrow \mathsf{C} \cup$ ancestors($B'$, blockById)        ▷ Collect into DAG
        end if
    end for
    return $\mathsf{C} \cup \{B\}$
end function
function verify-infx$^{D}_{\ell,m,k}(\mathcal{P})$
    blockById $\leftarrow \emptyset$
    for $(\pi, \chi) \in \mathcal{P}$ do
        for $B \in \pi$ do
            blockById[id($B$)] $\leftarrow B$
        end for
    end for
    $\tilde{\pi} \leftarrow$ best $\pi \in \mathcal{P}$ according to suffix verifier
    return $D$(ancestors($\tilde{\pi}[-1]$, blockById))
end function
```

3.6 Implementation & Parameters

We now discuss the size of NIPoPoW proofs and evaluate concrete parameters. Organizing the interlink data structure as a Merkle tree of $\log(|\mathsf{C}|)$ items, a proof-of-inclusion is provided in $\log\log(|\mathsf{C}|)$ space in expectation; the proof need not include 0-level pointers, but must include the genesis block. Transaction inclusion[1] can be proved in the block header in $\log(|\overline{x}|)$ using the standard Merkle tree of transactions, where $\overline{x}$ denotes the vector of all transactions included in the particular block. This makes the proof size require $\log(|\overline{x}|) + \log\log(|\mathsf{C}|)$ hashes per block for a total of $(2m(\log|\mathsf{C}| - \log m) + m)(\log|\overline{x}| + \log\log|\mathsf{C}|)$ hashes. In addition, $m(\log(|\mathsf{C}|) - \log(m))$ headers and coinbase transactions are needed. As an example, given that currently in bitcoin $|\mathsf{C}| = 464{,}185$ and $|\overline{x}| = 2000$, we have $\log(|\mathsf{C}|) = 18, \log\log(|\mathsf{C}|) = 5, \log(|\overline{x}|) = 11$. For the k-suffix, only k headers are needed. We set $k = 6$ and see that headers are 80 bytes and hashes 32 bytes. For the k-suffix as well as the $2m$ 0-blocks in π, neither coinbase data nor prev ids are needed, limiting header size to 48 bytes. The root and leaves of the pointers tree can be omitted from coinbase when transmitting the proof. In fact, no block ids need to be transmitted. From these observations, we estimate our scheme's proof sizes for various parameterizations of m in Table 3.1.

Concrete parameterization. To determine concrete values for security parameter m, we focus on a particular adversarial strategy and analyze its probability of

[1]This additional data is needed if a *soft* or *hard fork* is to be avoided. For more information about gradual deployment, consult the relevant section on *Deployment Paths.*

Table 3.1: Size of NIPoPoWs applied to Bitcoin today (≈450k blocks) for various values of m, setting $k = 6$.

m	**NIPoPoW size**	**Blocks**	**Hashes**
6	70 kB	108	1440
15	146 kB	231	2925
30	270 kB	426	5400
50	412 kB	656	8250
100	750 kB	1206	15000
127	952 kB	1530	19050

success. The attack is an extension of the stochastic processes described in [116] and [133].

The experiment works as follows: m is fixed and some adversarial computational power percentage q of the total network computational power is chosen; k is chosen based on q according to Nakamoto [116]. The number of blocks y during which parallel mining will occur is also fixed. The experiment begins with the adversary and honest parties sharing a common blockchain which ends in block B. After B is mined, the adversary starts mining in secret and in parallel with the honest parties on her own private fork on top of B. She ignores the honest chain, so that the two chains remain disjoint after B. As soon as y blocks have been mined in total, the adversary attempts a double spend via a NIPoPoW by creating two conflicting transactions which are committed to an honest block and an adversarial block respectively on top of each current chain. Finally, the adversary mines k blocks on top of the double spending transaction within her private chain. After these k blocks have been mined, she publishes her private chain in an attempt to overcome the honest chain.

We measure the probability of success of this attack. We experiment with various values of m for $y = 100$, indicating 100 blocks of secret parallel mining. We make the assumption that honest party communication is perfect and immediate. We ran 1,000,000 Monte Carlo executions[2] of the experiment for each value of m from 1 to 30. We ran the simulation for values of computational power percentage $q = 0.1$, $q = 0.2$ and $q = 0.3$. The results are plotted in Figure 3.5. Based on these data, we conclude that $m = 5$ is sufficient to achieve a 0.001 probability of failure against an adversary with 10% mining power. To secure against an adversary with more than 30% mining power, a choice of $m = 15$ is needed.

3.7 Evaluation & Applications

In this section we evaluate the cost of NIPoPoWs when used in realistic blockchain applications. First we simulated the resources savings resulting from the use of a NIPoPoW-based client compared to ordinary SPV. We model the arrival of payments in each cryptocurrency as a Poisson process and assume that the market cap of a cryptocurrency is a proxy for usage. Currently, a total of 731 cryptocurrencies are listed on coin market directories[3]. We narrow our focus to the 80

[2]Our experiment can be reproduced by running our code available under an open source MIT license at `https://github.com/dionyziz/popow/tree/master/experiment`

[3]`https://coinmarketcap.com/`

Figure 3.5: Simulation results for a private mining attacker with k according to Nakamoto and parallel mining parameter $y = 100$. Probabilities in logarithmic scale. The horizontal line indicates the threshold probability of [116].

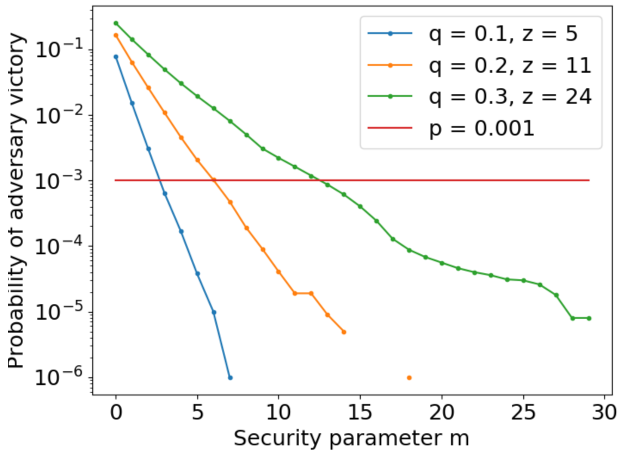


Table 3.2: Cost of header chains for all active PoW-based cryptocoins (collected from `coinwarz.com`)

Hash	**Coins**	**Size today**	**Growth rate**
Scrypt	44	4.3 GB	5.5 MB / day
SHA-256	15	1.4 GB	937.0 kB / day
X11	5	581.1 MB	556.3 kB / day
Quark	3	647.9 MB	518.4 kB / day
CryptoNight	2	199.0 MB	115.2 kB / day
EtHash	2	588.6 MB	921.6 kB / day
Groestl	2	300.3 MB	184.2 kB / day
NeoScrypt	2	310.6 MB	153.6 kB / day
Others	5	266.2 MB	311.1 kB / day
Total	80	8.5 GB	9.2 MB / day

cryptocurrencies that have their own PoW blockchains with a market cap of over USD $100,000.

In Table 3.2 we show aggregate statistics about these 80 cryptocurrencies, grouped according to the their PoW puzzle. While the entire chain in Bitcoin only amounts to 40 MB, taken together, the 80 cryptocurrencies comprise 10 GB of proofs-of-work, and generate 10 MB more each day. In Table 3.3 we show the resulting bandwidth costs from simulating a period of 60 days with $m = 24, k = 6$, with varying rates of payments received. For the naïve SPV client, the total bandwidth cost is dominated by fetching the entire chain of headers, which the NIPoPoW client avoids. The marginal cost for naïve SPV depends on the number of blocks generated per day, as well as the transaction inclusion proofs associated with each payment. The NIPoPoW-based client provides the most savings when the number of transactions per day is smallest, reducing the necessary bandwidth per day (excluding the initial sync up) by 90%.

Table 3.3: Simulated bandwidth of multi-blockchain clients after two months (Averaged over 10 trials each)

tx/	**Naive SPV**		**NIPoPoW**		
day	**Total**	**(Daily)**	**Total**	**(Daily)**	**Savings**
100	5.5 GB	(5.5 MB)	31.7 MB	(507 kB)	99% (91%)
500	5.5 GB	(5.7 MB)	68.2 MB	(1.1 MB)	99% (81%)
1000	5.5 GB	(6.0 MB)	99.1 MB	(1.6 MB)	98% (73%)
3000	5.6 GB	(7.0 MB)	192 MB	(3.1 MB)	97% (56%)

Multi-blockchain wallets. An application of our technique is an efficient multi-cryptocoin client. Consider a merchant who wishes to accept payments in any cryptocoin, not just the popular ones. The naïve approach would be to install an SPV client for each known coin. This approach would entail downloading the header chain for each coin, and periodically syncing up by fetching any newly generated block headers. An alternative would be to use an online service supporting multiple currencies, but this introduces reliance on a third party (e.g. Jaxx and Coinomi rely on third party networks).

A NIPoPoW-based client would not download the entire header chain, but would instead only receive NIPoPoW proofs each time a payment is received. When a peer informs the client about a payment, they include a block index ℓ and NIPoPoW proof of transaction inclusion. The peer must then query *all* of their connected peers, requesting any better proof for the same predicate. After waiting a short time period for a response, the client runs the `verify-infix` routine on all received proofs, and accepts the transaction if the output is *true*. Although initially such proofs must be relative to genesis, the client may store the most recently-known (k-stable) blockhash for each coin such that future payments can include NIPoPoW proofs relative to that. Thus for popular cryptocurrencies, the NIPoPoW-based client downloads nearly every block header, like an ordinary SPV client; but for coins used infrequently, the NIPoPoW-based client can skip over most blocks.

Cross-chain ICOs. As an example use-case of our construction, we present the case of an ICO in which tokens are distributed in one blockchain, but funds are raised in another. It works as follows: There are two designated blockchains, the *source* and the *destination blockchain*. The source is the blockchain where the fund-raising will take place, while the destination is the blockchain where the newly issued tokens will be distributed and subsequently exchanged. The destination blockchain must be smart-contract-enabled in order to allow for the distribution of ERC-20-style [150] tokens. In addition, the smart contracts on the destination blockchain must allow for programming the verification of a NIPoPoW proof by including, for example, the appropriate hash functions. The source blockchain must be NIPoPoW-enabled. This setup allows the creation of NIPoPoWs *about* the source blockchain which will be included in the destination blockchain. For example, a source blockchain can be Litecoin and a destination blockchain Ethereum.

In order to run the ICO, the fund-raising entity first creates a designated account in the source blockchain in which funds will be deposited. It then creates the ERC-20-style smart contract in the destination blockchain. When someone wishes to participate in the ICO, they transfer funds into the designated account on the source blockchain. Once they have made the transfer and it becomes confirmed, the payer generates a NIPoPoW about the transaction paying into the designated account. This NIPoPoW is then sent as a parameter to a method call on the ICO smart contract on the destination blockchain. The method call stores the proof and waits for a certain period of time for possible contestations, which can be accepted and compared using the $\leq_m$ mechanism previously described. If no contesting proof is presented within the contestation period, the prover receives their respective ICO tokens on the target blockchain. In order for only the rightful owner to be able to receive the tokens, they are required to sign the destination address on the destination blockchain using the private key corresponding to their source account used to make the payment within the source blockchain.

We implemented the NIPoPoW verifier algorithm as a Solidity smart contract[4]. The contract consists of two functions. The `submit_nipopow` function is used by the provers to provide their proof vectors. Instead of passing the block headers of the proof, the provers pass the hashes of the block headers and the hashes of the interlink vector. The reason is that the full data of the block header (especially the Merkle tree root) is only useful for the blocks of interest. Thus, we reduce the amount of data needed for the proof by a factor of 2. The rest of the parameters are used in the inclusion proof of the block. After confirming the validity of the proof, the `compare_proofs` function is called between the current and the best proof. If the current proof is better then it is assigned to the best proof in the contract's storage. The gas costs are summarized in Table 3.4. The $USD column represents the current price of this much gas on Ethereum.

Table 3.4: Verifier contract functions

Function	Data	Gas cost	$USD
`compare_proofs`	∼8Kb	∼5M	$4
`submit_nipopow`	∼65Kb	∼40M	$32

3.8 Superchain Quality Distributions

In order to argue formally about the security properties of blockchains that are equipped with the interlink data structure we introduce the new concept of *superchain quality*, which generalizes the chain quality property from the backbone model [60].

We first define a notion of "goodness" that bounds the deviation of superblocks of a certain level from their expected mean. Using this we then define superchain quality.

Intuitively, these definitions tell us that μ-superblocks occur approximately once every 2^μ blocks. Below, we make this notion more formal.

Definition 53 (Locally good superchain). *A superchain* C' *of level* μ *with underlying chain* C *is said to be* μ-locally-good *with respect to security parameter* δ*, written* $\textsf{local-good}_\delta(\mathsf{C}', \mathsf{C}, \mu)$*, if* $|\mathsf{C}'| > (1-\delta)2^{-\mu}|\mathsf{C}|$.

Definition 54 (Superchain quality). *The* (δ, m) *superquality property* Q^μ_{scq} *of a chain* C *pertaining to level* μ *with security parameters* $\delta \in \mathbb{R}$ *and* $m \in \mathbb{N}$ *states that*

[4]The source code of the smart contract is available under an open source MIT license at `https://github.com/dionyziz/popow/blob/master/experiment/contractNipopow.sol`

for all $m' \geq m$, it holds that $\textsf{local-good}_\delta(C{\uparrow^\mu}[-m':], C{\uparrow^\mu}[-m':]{\downarrow}, \mu)$*. That is, all sufficiently large suffixes are locally good.*

Definition 55 (Multilevel quality)**.** *A μ-superchain C' is said to have* multilevel quality*, written* multi-good$_{\delta,k_1}(\mathsf{C}, \mathsf{C}', \mu)$ *with respect to an underlying chain* $\mathsf{C} = \mathsf{C}'{\downarrow}$ *with security parameters k_1, δ if for all $\mu' < \mu$ it holds that for any $\mathsf{C}^* \subseteq \mathsf{C}$, if $|\mathsf{C}^*{\uparrow^{\mu'}}| \geq k_1$, then $|\mathsf{C}^*{\uparrow^{\mu}}| \geq (1-\delta)2^{\mu'-\mu}|\mathsf{C}^*{\uparrow^{\mu'}}|$.*

Putting the above together we conclude with the notion of a *good* superchain.

Definition 56 (Good superchain)**.** *A μ-superchain C' is said to be* good*, written* good$_{\delta,k_1}(\mathsf{C}, \mathsf{C}', \mu)$*, with respect to an underlying chain $\mathsf{C} = \mathsf{C}'{\downarrow}$ if it has both superquality and multilevel quality with parameters (δ, m).*

It is not hard to see that the above good statistical properties are attained with overwhelming probability by all chains that are generated in optimistic environments, i.e. if no adversary tries to violate them. We formalize this in the following theorems.

Lemma 19 (Local goodness)**.** *Assume C contains only honestly-generated blocks in an optimistic execution. For all levels μ, for all constant $\delta > 0$, all continuous subchains $\mathsf{C}' = \mathsf{C}[i:j]$ with $|\mathsf{C}'| \geq m$ are locally good,* $\textsf{local-good}_\delta(\mathsf{C}', \mathsf{C}, \mu)$*, with overwhelming probability in m.*

Proof. Observing that for each honestly generated block the probability of being a μ-superblock for any level μ follows an independent Bernoulli distribution, we can apply a Chernoff bound to show that the number of superblocks within a chain will be close to its expectation, which is what is required for local goodness. □

Lemma 20 (Multilevel quality)**.** *For all $\mu, 0 < \delta \leq 0.5$, chain C containing only honestly-generated blocks in an optimistic execution has (δ, k_1) multilevel quality at level μ with overwhelming probability in k_1.*

Proof. Identical. □

Lemma 21 (Superquality)**.** *For all $\mu, \delta > 0$, a chain C adopted in an optimistic execution has (δ, m)-superquality at level μ with overwhelming probability in m.*

Proof. Let $\mathsf{C}' = \mathsf{C}{\uparrow^\mu}$ and let $\mathsf{C}^* = \mathsf{C}'[-m':]$ for some $m' \geq m$. Then let $B \in \mathsf{C}^*{\downarrow}$ and let X_B be the random variable equal to 1 if *level*$(B) \geq \mu$ and 0 otherwise. $\{X_B : B \in \mathsf{C}^*\}$ are mutually independent Bernoulli random variables with expectation $E(X_B) = 2^{-\mu}|\mathsf{C}^*{\downarrow}|$. Let $X = \sum_{B \in \mathsf{C}^*{\downarrow}} X_B$. Then X follows a Binomial distribution with parameters $(m', 2^{-\mu})$ and note that $|\mathsf{C}^*| = X$. Then $\mathbb{E}(|\mathsf{C}^*{\downarrow}|) = 2^{-\mu}|\mathsf{C}^*|$. Applying a Chernoff bound on $|\mathsf{C}^*{\downarrow}|$ we obtain $\Pr[|\mathsf{C}^*{\downarrow}| \leq (1-\delta)2^{-\mu}|\mathsf{C}^*{\downarrow}] \leq \exp(-\delta^2 2^{-\mu-1}|\mathsf{C}^*|)$. □

Lemma 22 (Optimistic superchain distribution)**.** *For any level μ, and any $0 < \delta < 0.5$, a chain C containing only honestly-generated blocks adopted by an honest party in an execution with random network scheduling is (δ, m)-good at level μ with overwhelming probability in m.*

Proof. This is a direct consequence of Lemma 21 and Lemma 20. □

3.9 Proof of attack on PoPoW

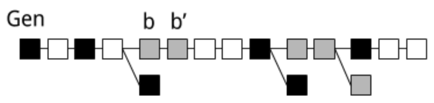
We now show that, if the statistical properties of blockchains are not respected in some execution, our construction presented previously is insecure by illustrating an explicit attack against our scheme. During the exposition of this attack, a simple patch for our construction, which will also lead to a correct generic security proof, will become clear.

We proceed in two steps. We first show that a powerful attacker can break chain superquality with non-negligible probability. Then we construct a concrete double spending attack based on this observation assuming an attacker of sufficiently high hashing power (but still below 50%).

3.9.1 Attacking chain superquality

We construct an adversary $\mathcal{A}$ that breaks the superchain quality at level μ. $\mathcal{A}$ works as follows. Assume she wants to attack the honest party B in order to have him adopt chain C_B which has a bad distribution of superblocks, i.e. such that local goodness is violated in some sufficiently long subchain. She continuously determines the current chain C_B adopted by B. The adversary starts playing after $|\mathsf{C}_B| \geq 2$. If $\mathit{level}(\mathsf{C}_B[-1]) < \mu$, then $\mathcal{A}$ remains idle. However, if $\mathit{level}(\mathsf{C}_B[-1]) \geq \mu$, then $\mathcal{A}$ attempts to mine an adversarial block b on top of $\mathsf{C}_B[-2]$. If successful, she attempts to mine another block b' on top of b. If successful again, she broadcasts b and b'. The adversarial mining continues until B adopts a new chain, which can be due to two reasons: Either the adversary successfully mined b, b' on top of $\mathsf{C}_B[-2]$ and B adopts them; or one of the honest parties mined a block which was adopted by B. In either case, the adversary restarts the strategy by inspecting $\mathsf{C}[-1]$ and acting accordingly. An execution of this attack is illustrated in Figure 3.6.

Figure 3.6: Superquality attack on prior work (PoPoW) [83]. The adversary performs a selfish-mining [53] attack (gray blocks) whenever any honest parties have recently mined a rare μ-superblock (black). The attack reduces the honest chain's superquality, while the attacker's private chain is unaffected.



Assume now that an honestly-generated μ-superblock was adopted by B at position $\mathsf{C}_B[i]$ at round r. We now examine the probability that $\mathsf{C}_B[i]$ will remain a μ-superblock in the long run. Suppose $r' > r$ is the first round after r during which a block is generated. $\mathcal{A}$ will succeed in this attack with non-negligible probability and cause B to abandon the μ-superblock from their adopted chain. Therefore, there exists δ such that the adversary will be able to cause δ-variance with non-negligible probability in m. This suffices to show that superquality is violated.

As seen in the illustration, while the honest parties have generated several μ-superblocks, some of them are in blockchain forks which have been abandoned, causing a superquality harm.

3.9.2 A double-spending attack

Extending the above attack, we modify the superquality attacker into an attacker that causes a double spending attack in the PoPoW construction. We first give a sketch of the attack.

As before, $\mathcal{A}$ targets the proofs generated by honest party B by violating μ-superquality in B's adopted chain. $\mathcal{A}$ begins by remaining idle until a certain chosen block b. After block b is produced, $\mathcal{A}$ starts mining a secret chain which forks off from b akin to a selfish mining attacker [53]. The adversary performs a normal spending transaction on the honestly adopted blockchain and has it confirmed in the block immediately following block b. She also produces a double spending transaction which she secretly confirms in her secret chain in the block immediately following b.

$\mathcal{A}$ keeps extending her own secret chain as usual. However, whenever a μ-superblock is adopted by B, she temporarily pauses mining in her secret chain and devotes her mining power to harm the μ-superquality of B's adopted chain. Intuitively, for large enough μ, the time spent trying to harm superquality will be limited, because the probability of a μ-superblock occurring will be small. Therefore, the adversary's superchain quality will be larger than the honest parties' superchain quality (which will be harmed by the adversary) and therefore, even though the adversary's 0-chain will be shorter than the honest parties' 0-chain, the adversary's μ-superchain will be longer than the honest parties' μ-superchain and thus will be favored by the verifier. We just remark here that for appropriate choice of system parameters, the attack can be made to succeed with overwhelming probability.

We now calculate the exact probability of success of the attack. The attack is parameterized by parameters r, μ which are picked by the adversary. μ is the superblock level at which the adversary will produce a proof longer than the honest proof. The modified attack works as follows: Without loss of generality, fix block b to be Genesis. The adversary always mines on the secret chain which forks off from genesis, unless a *superblock generation event* occurs. If a superblock generation event occurs, then the adversary pauses mining on the secret chain and attempts a *block suppression attack* on the honest chain. The adversary devotes exactly r rounds to this suppression attack; then resumes mining on the secret chain. We show that, despite this simplification (of fixing r) which is harmful to the adversary, the probability of a successful attack is non-negligible for certain values of the protocol parameters [5].

The adversary monitors the network for superblocks. Whenever an honest party diffuses an honestly-generated μ-superblock, at the end of a given round r_1, the adversary starts devoting their mining power to block suppression starting from the next round.

The block suppression attack works as follows. Let b be the honestly generated μ-superblock which was diffused at the end of the previous round. If the round was not uniquely successful, let b be any of the diffused honestly-generated μ-superblocks. Let b be the tip of an honest chain C_B. The adversary first mines on top of $\mathsf{C}_B[-2]$. If she is successful in mining a block b', she continues extending the chain ending at b' (to mine b'' and so on). The value r is fixed, so the adversary devotes exactly r rounds to this whole process; the adversary will keep mining on top of $\mathsf{C}_B[-2]$ (or one of the adversarially-generated extensions of it) for exactly r rounds, regardless

[5]The attack could be further optimized, but we simplify it for exposition.

of whether b' or b'' have been found. At the same time, the honest parties will be mining on top of b (or a competing block in the case of a non-uniquely successful round). Again, further successful block diffusion by the honest parties shall not affect that the adversary is going to spend exactly r rounds for suppression. This attack will succeed with overwhelming probability for the right choice of protocol values.

Theorem 23 (Double-spending attack)**.** *There exist parameters p, n, t, q, μ, δ, with $t \leq (1-\delta)(n-t)$, and a double spending attack against the constructions of Section 3.3 and Section 3.5 that succeeds with overwhelming probability.*

Proof. Recall that in the backbone notation n denotes the total number of parties, t denotes the number of adversarial parties, q denotes the number of the random oracle queries allowed per party per round and p is the probability that one random oracle query will be successful and remember that $p = T/2^\kappa$ where T is the mining target and κ is the security parameter (or hash function bit count). Then f denotes the probability that a given round is successful and we have that $f = 1 - (1-p)^{q(n-t)}$. Recall further that a requirement of the backbone protocol is that the honest majority assumption is satisfied, that is that $t \leq (1-\delta)(n-t)$ were $\delta \geq 2f + 3\epsilon$, where $\epsilon \in (0,1)$ is an arbitrary small constant describing the quality of the concentration of the random variables.

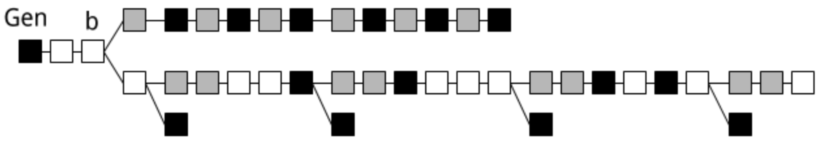
Denote $\alpha_\mathcal{A}$ the secret chain generated by the adversary and α_B the honest chain belonging to any honest party. We will show that for certain protocol values we have that $\Pr[|\alpha_\mathcal{A}{\uparrow}^\mu| \geq |\alpha_B{\uparrow}^\mu|]$ is overwhelming.

Assume that, to the adversary's harm and to simplify the analysis, the adversary plays at beginning of every round and does not perform adversarial scheduling. At the beginning of the round when it is the adversary's turn to play, she has access to the blocks diffused during the previous round by the honest parties.

First, observe that at the beginning of each round, the adversary finds herself in one of two different situations: Either she has been forced into an r-round-long period of suppression, or she is not in that period. If she is within that period, she blindly performs the suppression attack without regard for the state of the world. If she is not within that period, then she must initially observe the blocks diffused at the end of the previous round by the honest parties. Call these rounds during which the diffused data must be examined by the adversary *decision rounds*. Let there be ω decision rounds in total. In each such decision round, it is possible that the adversary discovers a diffused μ-superblock and therefore decides that a suppression attack must be performed starting with the current round. Call these rounds during which this discovery is made by the adversary *migration rounds*. Let there be y migration rounds in total. The adversary devotes the migration round to performing the suppression attack as well as $r-1$ non-migration rounds after the migration round. Call these rounds, including the migration round, *suppression rounds*. In the rest of the decision rounds, the adversary will not find any μ-superblocks diffused. Call these *secret chain rounds*; these are rounds where the adversary devotes her queries to mining on the secret chain. Let there be x secret chain rounds. If the adversary devotes ω decision rounds to the attack in total, then clearly we have that $\omega = x + y$. If the total number of rounds during which the attack is running is s then we also have that $s = x + ry$, because for each migration round there are $r-1$ non-decision rounds that follow.

We will analyze the honest and adversarial superchain lengths with respect to ω, which roughly corresponds to time (because note that $\omega \geq s/r$, and so ω is

Figure 3.7: The double spending attack. The top chain fork is wholly adversarially mined, while the bottom is honestly adopted. Gray blocks are adversarially mined 0-blocks. Black blocks are μ-superblocks.



proportional to the number of rounds). Let us calculate the probability p_{SB} ("superblock probability") that a decision round ends up being a migration round. Ignoring the negligible event that there will be random oracle collisions, we have that $p_{SB} = (n-t)qp2^{-\mu}$.

Given this, note that the decision taken at the beginning of each decision round follows independent Bernoulli distributions with probability p_{SB}. Denote z_i the indicator random variable indicating whether the decision round was a migration round. Therefore we can readily calculate the expectations for the random variables x and y, as $x = \omega - y$, $y = \sum_{i=1}^{\omega} z_i$. We have $E[x] = (1-p_{SB})\omega$ and $E[y] = p_{SB}\omega$. Applying a Chernoff bound to the random variables x and y, we observe that they will attain values close to their mean for large ω and in particular $\Pr[y \geq (1+\delta)E[y]] \leq \exp(-\frac{\delta^2}{3}E[y])$ and similarly $\Pr[x \leq (1-\delta)E[x]] \leq \exp(-\frac{\delta^2}{2}E[x])$, which are negligible in ω.

Given that there will be x secret chain rounds, we observe that the random variable indicating the length of the secret adversarial superchain follows the binomial distribution with xtq repetitions and probability $p2^{-\mu}$. We can now calculate the expected secret chain length as $E[|\alpha_{\mathcal{A}}\uparrow^{\mu}|] = xtqp2^{-\mu}$. Observe that in this probability we have given the adversary the intelligence to continue using her random oracle queries during a round even after a block has been found during a round and not to wait for the next round. Applying a Chernoff bound, we obtain that $\Pr[|\alpha_{\mathcal{A}}\uparrow^{\mu}| \leq (1-\delta)E[|\alpha_{\mathcal{A}}\uparrow^{\mu}|]] \leq exp(-\frac{\delta^2}{2}E[|\alpha_{\mathcal{A}}\uparrow^{\mu}|])$, which is negligible in ω (because we know that with overwhelming probability $x > (1-\delta)(1-p_{SB})\omega$).

It remains to calculate the behavior of the honest superchain. Suppose that a migration round occurs during which at least one superblock B is diffused. We will now calculate the probability p_{sup} that the adversary is able to suppress that block after r rounds by performing the suppression attack and cause all honest parties to adopt a chain not containing B.

One way for this to occur is if the adversary has generated exactly 2 shallow blocks (blocks which are not μ-superblocks) after exactly r rounds and the honest parties having generated exactly 0 blocks after exactly r rounds. This provides a lower bound for the probability, which is sufficient for our purposes. Call ADV-WIN the event where the adversary has generated exactly 2 shallow blocks after exactly r rounds since the diffusion of B and call HON-LOSE the event where the honest parties have generated exactly 0 blocks after exactly r rounds since the diffusion of B.

The number of blocks generated by the adversary after the diffusion of B follows the binomial distribution with r repetitions and probability p_{LB}, where p_{LB} denotes the probability that the adversary is able to produce a shallow block ("low block probability") during a single round. We have that $p_{LB} = tqp(1 - 2^{-\mu})$. To evaluate Pr[ADV-WIN], we evaluate the binomial distribution for 2 successes to obtain Pr[ADV-WIN] $= \frac{r(r-1)}{2} p_{LB}^2 (1 - p_{LB})^{r-2}$. The number of blocks generated by the honest parties after the diffusion of B follows the binomial distribution with r repetitions and probability f. To evaluate Pr[HON-LOSE], we evaluate the binomial distribution for 0 successes to obtain Pr[HON-LOSE] $= (1 - f)^r$. Note that this is an upper bound in the probability, in particular because there can be multiple blocks during a non-uniquely successful round during which a μ-superblock was generated.

Then observe that the two events ADV-WIN and HON-LOSE are independent and therefore $p_{sup} = \Pr[\text{ADV-WIN}] \Pr[\text{HON-LOSE}] = \frac{r(r-1)}{2} p_{LB}^2 (1 - p_{LB})^{r-2} (1 - f)^r$.

Now that we have evaluated p_{sup}, we will calculate the honest chain length in two chunks: The superblocks generated and adopted by the honest parties during secret chain rounds, C_1, and the superblocks generated and adopted by the honest parties during suppression rounds, C_2 (and note that these sets of blocks are not blockchains on their own).

$|\mathsf{C}_1|$ is a random variable following the binomial distribution with $s(n - t)q$ repetitions and probability $p2^{-\mu}(1 - p_{sup})$. In the evaluation of this distribution, we give the honest parties the liberty to belong to a mining pool and share mining information within a round, an assumption which only makes matters for the adversary worse. We can now calculate the expected length of C_1 to find $E[|\mathsf{C}_1|] = s(n - t)qp2^{-\mu}(1 - p_{sup})$. Applying a Chernoff bound, we find that $\Pr[|\mathsf{C}_1| \geq (1 + \delta)E[|\mathsf{C}_1|]] \leq exp(-\frac{\delta^2}{3} E[|\mathsf{C}_1|])$, which is negligible in s.

Finally, some additional μ-superblocks could have been generated by the honest parties while the adversary is spending r rounds attempting to suppress a previous μ-superblock. These μ-superblocks will be adopted in the case the adversary fails to suppress the previous μ-superblock. As the adversary does not devote any decision rounds to these new μ-superblocks, they will never be suppressed if the previous μ-superblock is not suppressed. We collect these in the set C_2. To calculate $|\mathsf{C}_2|$, observe that the number of unsuppressed μ-superblocks which caused an adversarial suppression period is $|\mathsf{C}_1|$. For each of these blocks, the honest parties spend r rounds attempting to form further μ-superblocks on top. The total number of such attemps is $r|\mathsf{C}_1|$. Therefore, the number of further honestly generated μ-superblocks attained during the $|\mathsf{C}_1|$ different r-round periods follows a binomial distribution with $|\mathsf{C}_1|rq(n - t)$ repetitions and probability $p2^{-\mu}$. Here we allow the honest parties to use repeated queries within a round even after a shallow success and to work in a pool to obtain an upper bound for the expectation. Therefore $E[|\mathsf{C}_2|] = |\mathsf{C}_1|rq(n-t)p2^{-\mu}$ and applying a Chernoff bound we obtain that $\Pr[|\mathsf{C}_2| \geq (1+\delta)E[|\mathsf{C}_2|]] \leq exp(-\frac{\delta}{3}E[|\mathsf{C}_2|])$, which is negligible in s and has a quadratic error term. We deduce that $|\mathsf{C}_2|$ will have a very small length compared to the rest of the honest chain, as it is a vanishing term in μ.

Concluding the calculation of the adversarial superchain, we get $E[|\alpha_B{\uparrow}^{\mu}|] = E[|\mathsf{C}_1|] + E[|\mathsf{C}_2|]$.

Finally, it remains to show that there exist values $p, n, t, q, r, \mu, \delta$ such that a $E[|\alpha_{\mathcal{A}}{\uparrow}^{\mu}|] \geq (1 + \delta)E[|\alpha_B{\uparrow}^{\mu}|]$. Using the values $p = 10^{-5}, q = 1, n = 1000, t = 489, \mu = 25, r = 200$, we observe that the honest majority assumption is preserved. Replacing these values into the expectations formulae above, we obtain $E[|\alpha_{\mathcal{A}}{\uparrow}^{\mu}|] \approx 1.457 * 10^{-10} * \omega$ and $E[|\alpha_B{\uparrow}^{\mu}|] \approx 1.424 * 10^{-10} * \omega$, which result to a constant gap δ. Because the adversarial chain grows linearly in ω, the adversary only has to wait sufficient rounds for obtaining m blocks to create a valid proof. Therefore, for these values, the adversary will be able to generate a convincing PoPoW at some level μ which is longer than the honest parties' proof, even when the adversary does not have a longer underlying blockchain. □

3.9.3 Interactive Proofs of Proof-of-Work

Our attack also applies against the protocol described in [83].

In [83], the main algorithm of the verifier aims at distinguishing between two candidate proofs (π_A, χ_A) and (π_B, χ_B). The honest prover, having adopted C_B during mining, initially produces the proof (π_B, χ_B) as follows. First, the last k blocks are sent as $\chi_B = \mathsf{C}_B[-k:]$. Then for the first part of the chain, $\mathsf{C}_B[:-k]$, the prover sets π_B to be the μ-superchain spanning C_B for the largest μ such that $|\pi_B| = m$, where m is the protocol's security parameter. The verifier ensures that $|\pi_A| \geq m, |\pi_B| \geq m$ so that the proofs are not shorter than m and then checks whether $\pi_A = \pi_B$; if so, the decision is drawn immediately based on χ_A, χ_B without interaction. Otherwise, the verifier queries the provers for their claimed anchored superchains $\mathsf{C}_A{\uparrow}^\mu$, $\mathsf{C}_B{\uparrow}^\mu$ at some level μ. The verifier starts querying at the highest possible level μ and descends until level μ is sufficiently low such that $b = LCA(\pi_A{\uparrow}^\mu, \pi_B{\uparrow}^\mu)$ is m blocks from the tip of the chain for one of the proofs. That is, the querying stops at such μ when $max(|\pi_A{\uparrow}^\mu\ \{b:\}|, |\pi_B{\uparrow}^\mu\ \{b:\}|) \geq m$. The winner is designated as the prover with the most blocks after b at that level; i.e., A, if $|\pi_A{\uparrow}^\mu\ \{b:\}| \geq |\pi_B{\uparrow}^\mu\ \{b:\}|$, and B otherwise. The communication overhead is reduced by only requesting blocks after the purported LCA. The security parameter m is chosen to ensure that the probability of the attacker producing a long superchain is negligible.

It is worth isolating the mistake in their security proof. Suppose player B is honest and player $\mathcal{A}$ is adversarial and suppose b, the LCA block, was honestly generated and suppose that the superchain comparison happens at level μ. Their security proof then correctly argues that there will have been more honestly- than adversarially-generated μ-superblocks after block b. Nevertheless, we observe that the mere fact that there have been more honestly- than adversarially-generated μ-superblocks after b does not imply that $|\overline{\pi}_\mathcal{A}{\uparrow}^\mu\ \{b:\}| \leq |\overline{\pi}_B{\uparrow}^\mu\ \{b:\}|$. The reason is that some of these superblocks could belong to blocktree forks that have been abandoned by B. Thus, the security conclusion does not follow. Regardless, their basic argument and construction is what we will use as a basis for constructing a system that is both provably secure and succinct under the same assumptions, albeit requiring a more complicated construction structure to obtain security.

3.10 Formal security treatment

Based on the attack explored above, it is now easy to see that our construction can be patched in a straightforward manner to achieve security. In particular, since the manner in which the adversary was able to subvert the prover was by the violation of *goodness*, we can mandate that the prover only tries to use succinct proofs to prove claims about chains that are *good at every level*. In case goodness is violated, the prover simply falls back to providing the whole chain. This allows us to argue that the construction is secure by distinguishing two cases. In case goodness is violated, the honest prover will fall back to providing the whole chain, in which case security will be reduced to the security of the standard blockchain protocol choosing the longest 0-chain. In case goodness is not violated, we will argue that the adversary is unable to win in these comparisons.

The previous construction was designed to prevent Bahack-style attacks [16], where the adversary constructs "lucky" high-difficulty superblocks without filling in the underlying proof-of-work in the lower levels. We now patch our protocol which, while retaining this high level approach, adds a defence against the double-spending attack of Section 3.9. The attack is neutralized since our verifier is more permissive, allowing the prover to construct a proof that takes superquality "goodness" into account when comparing forks. The modified construction is shown in Algorithm 26. The algorithm has been modified to check the current portion of the subchain α for *goodness* prior to moving to the lower superchain level. If goodness is indeed maintained at the current level μ, the prover only tries to cover the span of the last m blocks of level μ at level $\mu - 1$, as seen in Line 7. Otherwise, if goodness is violated at the part of the subchain α at level μ, then the prover completely ignores level μ and tries to use the lower level $\mu - 1$ to cover the whole span of α.

Algorithm 26 The *goodness aware* Prove algorithm for the NIPoPoW protocol

```
1:  function $\mathsf{Prove}^{\mathsf{good}}_{m,k,\delta}(\mathsf{C})$
2:      $B \leftarrow \mathsf{C}[0]$                                        ▷ Genesis
3:      for $\mu = |\mathsf{C}[-k].\mathsf{interlink}|$ down to 0 do
4:          $\alpha \leftarrow \mathsf{C}[: -k]\{B :\}{\uparrow^\mu}$
5:          $\pi \leftarrow \pi \cup \alpha$
6:          if $\text{good}_{\delta,m}(\mathsf{C}, \alpha, \mu)$ then
7:              $B \leftarrow \alpha[-m]$
8:          end if
9:      end for
10:     $\chi \leftarrow \mathsf{C}[-k :]$
11:     return $\pi\chi$
12: end function
```

Only the concrete prover needs to be modified. The verifier and $\leq_m$ operator remain as defined previously.

To aid intuition, we give a sketch of the proof before giving the full technical proof.

Theorem 17 (Security)**.** *Assuming honest majority, the Non-interactive Proofs of Proof-of-Work construction for computable k-stable monotonic suffix-sensitive predicates is secure with overwhelming probability in κ.*

Intuition. Suppose an adversary produces a proof $\pi_{\mathcal{A}}$ and an honest party produces a proof π_B such that the two proofs cause the predicate Q to evaluate to different values, while at the same time all honest parties have agreed that the correct value is the one obtained by π_B. Because of Bitcoin's security, $\mathcal{A}$ will be unable to make these claims for an actual underlying 0-level chain.

We now argue that the operator $\leq_m$ will signal in favour of the honest parties. Suppose b is the LCA block between $\pi_{\mathcal{A}}$ and π_B. If the chain forks at b, there can be no more adversarial blocks after b than honest blocks after b, provided there are at least k honest blocks (due to the Common Prefix property). We will now argue that, further, there can be no more disjoint $\mu_{\mathcal{A}}$-level superblocks than honest μ_B-level superblocks after b.

To see this, let b be an honest block generated at some round r_1 and let the honest proof be generated at some round r_3. Then take the sequence of consecutive rounds $S = (r_1, \cdots, r_3)$. Because the verifier requires at least m blocks from each of the provers, the adversary must have m $\mu_{\mathcal{A}}$-superblocks in $\pi_{\mathcal{A}}\{b :\}$ which are not in $\pi_B\{b :\}$. Therefore, using a negative binomial tail bound argument, we see that $|S|$ must be long; intuitively, it takes a long time to produce a lot of blocks $|\pi_{\mathcal{A}}\{b :\}|$. Given that $|S|$ is long and that the honest parties have more mining power, they must have been able to produce a longer $\pi_B\{b :\}$ argument (of course, this comparison will have the superchain lengths weighted by $2^{\mu_{\mathcal{A}}}, 2^{\mu_B}$ respectively). To prove this, we use a binomial tail bound argument; intuitively, given a long time $|S|$, a lot of μ_B-superblocks $|\pi_B\{b :\}|$ will have been honestly produced.

We therefore have a fixed value for the length of the adversarial argument, a negative binomial random variable for the number of rounds, and a binomial random variable for the length of the honest argument. By taking the expectations of the above random variables and applying a Chernoff bound, we see that the actual values will be close to their means with overwhelming probability, completing the proof. □

We now give a formal treatment of the above security proof.

Assume t adversarial and n total parties, each with q PoW random oracle queries per round. We will call a query to the RO *μ-successful* if the RO returns a value h such that $h \leq 2^{-\mu}T$.

We define boolean random variables X_r^μ, Y_r^μ and Z_r^μ. Fix some round r, query index j and adversarial party index k (out of t). If at round i an honest party obtains a PoW with $id < 2^{-\mu}T$, set $X_r^\mu = 1$, otherwise $X_r^\mu = 0$. If at round r exactly one honest party obtains a PoW with $id < 2^{-\mu}T$, set $Y_r^\mu = 1$, otherwise $Y_r^\mu = 0$. If at round r the j-th query of the k-th corrupted party is μ-successful, set $Z_{ijk}^\mu = 1$, otherwise $Z_{ijk}^\mu = 0$. Let $Z_r^\mu = \sum_{k=1}^{t}\sum_{j=1}^{q} Z_{ijk}^\mu$. For a set of rounds S, let $X^\mu(S) = \sum_{r\in S} X_r$ and similarly define $Y^\mu(S), Z^\mu(S)$.

Definition 57 (Typical execution). *An execution of the protocol is* (ϵ, η)-typical *if:*

Block counts don't deviate. *For all $\mu \geq 0$ and any set S of consecutive rounds with $|S| \geq 2^\mu \eta\kappa$, we have:*

- $(1-\epsilon)E[X^\mu(S)] < X^\mu(S) < (1+\epsilon)E[X^\mu(S)]$ *and* $(1-\epsilon)E[Y^\mu(S)] < Y^\mu(s)$.
- $Z^\mu(S) < (1+\epsilon)E[Z^\mu(S)]$.

Round count doesn't deviate. *Let S be a set of consecutive rounds such that $Z^\mu(S) \geq k$ for some security parameter k. Then $|S| \geq (1-\epsilon)2^\mu \frac{Z^\mu(S)}{pqt}$ with overwhelming probability in k.*

Chain regularity. *No insertions, no copies, and no predictions [60] have occurred.*

Theorem 24 (Typicality). *Executions are (ϵ, η)-typical with overwhelming probability in κ.*

Proof. **Block counts and regularity.** For the blocks count and regularity, we refer the reader to [60] for the full proof.

Round count. First, observe that $Z^{\mu}_{ijk} \sim \mathsf{Bern}(2^{-\mu}p)$ and these are jointly independent. Therefore $Z^{\mu}_{S} \sim \mathsf{Bin}(tq|S|, 2^{-\mu}p)$ and $|S| \sim \mathsf{NB}(Z_S, 2^{-\mu}p)$. So $\mathbb{E}(|S|) = 2^{\mu}\frac{Z_S}{pqt}$. Applying a tail bound to the negative binomial distribution, we obtain that $\Pr[|S| < (1-\epsilon)\mathbb{E}(|S|)] \in \Omega(\epsilon^2 m)$. □

The following lemma is at the heart of the security proof that will follow.

Lemma 25. *Suppose S is a set of consecutive rounds $r_1 \dots r_2$ and C_B is a chain adopted by an honest party at round r_2 of a typical execution. Let $\mathsf{C}^S_B = \{b \in \mathsf{C}_B :$ b was generated during $S\}$. Let $\mu_{\mathcal{A}}, \mu_B \in \mathbb{N}$. Suppose $\mathsf{C}^S_B{\uparrow}^{\mu_B}$ is good. Suppose $\mathsf{C}'_{\mathcal{A}}$ is a $\mu_{\mathcal{A}}$-superchain containing only adversarially generated blocks generated during S and suppose that $|\mathsf{C}'_{\mathcal{A}}| \geq k$. Then $2^{\mu_{\mathcal{A}}}|\mathsf{C}'_{\mathcal{A}}| < \frac{1}{3}2^{\mu_B}|\mathsf{C}^S_B{\uparrow}^{\mu_B}|$.*

Proof. From $|\mathsf{C}'_{\mathcal{A}}| \geq k$ we know that $Z^{\mu}(S) \geq k$. From the definition of typicality, we have $|S| \geq (1-\epsilon')2^{\mu_{\mathcal{A}}}\frac{1}{pqt}|\mathsf{C}'_{\mathcal{A}}|$. Applying the chain growth theorem [60] we obtain that $|\mathsf{C}^S_B| \geq (1-\epsilon)f|S|$. But from the goodness of $\mathsf{C}^S_B{\uparrow}^{\mu_B}$, we know that $|\mathsf{C}^S_B{\uparrow}^{\mu_B}| \geq (1-\delta)2^{-\mu_B}|\mathsf{C}^S_B|$. Therefore $|\mathsf{C}^S_B{\uparrow}^{\mu_B}| \geq 2^{-\mu_B}(1-\delta)(1-\epsilon)f(1-\epsilon')2^{\mu_{\mathcal{A}}}\frac{1}{pqt}|\mathsf{C}'_{\mathcal{A}}|$ and so $2^{\mu_{\mathcal{A}}}|\mathsf{C}'_{\mathcal{A}}| < \frac{pqt}{(1-\delta)(1-\epsilon')(1-\epsilon)f}2^{\mu_B}|\mathsf{C}^S_B{\uparrow}^{\mu_B}|$. □

Definition 58 (Adequate level of honest proof)**.** *Let π be an honestly generated proof constructed upon some adopted chain C and let $b \in \pi$.*

Then μ' is defined as $\mu' = \max\{\mu : |\pi\{b:\}{\uparrow}^{\mu}| \geq \max(m+1, (1-\delta)2^{-\mu}|\pi\{b:\}{\uparrow}^{\mu}{\downarrow}|)\}$. We call μ' the adequate *level of proof π with respect to block b with security parameters δ and m.*

Note that the adequate level of a proof is a function of both the proof π and the chosen block b.

Lemma 26. *Let π be some honest proof generated with security parameters δ, m. Let C be the underlying chain, $b \in \mathsf{C}$ be any block and μ' be the adequate level of the proof with respect to b and the same security parameters.*

Then $\mathsf{C}\{b:\}{\uparrow}^{\mu'} = \pi\{b:\}{\uparrow}^{\mu'}$.

Proof. $\pi\{b:\}{\uparrow}^{\mu'} \subseteq \mathsf{C}\{b:\}{\uparrow}^{\mu'}$ is trivial. For the converse, we show that for all $\mu^* > \mu'$, we have that in the iteration of the Prove `for` loop with $\mu = \mu^*$, the block stored in variable B precedes b in C.

Suppose $\mu = \mu^*$ is the first `for` iteration during which the property is violated. This cannot be the first iteration, as there $B = \mathsf{C}[0]$ and Genesis precedes all blocks. By the induction hypothesis we see that during the iteration $\mu = \mu^* + 1$, B preceded b. From the definition of μ' we know that μ' is the highest level for which $|\pi\{b:\}{\uparrow}^{\mu'}[1:]| \geq \max(m, (1-\delta)2^{-\mu'}|\pi\{b:\}{\uparrow}^{\mu'}[1:]{\downarrow}|)$.

Hence, this property cannot hold for $\mu^* > \mu'$ and therefore $|\pi_B\{b:\}{\uparrow}^{\mu^*}[1:]| < m$ or $\neg\mathsf{local\text{-}good}_{\delta}(\pi\{b:\}{\uparrow}^{\mu^*}[1:], \mathsf{C}, \mu^*)$.

In case $\mathsf{local\text{-}good}$ is violated, variable B remains unmodified and the induction step holds. If $\mathsf{local\text{-}good}$ is not violated, then $|\pi\{b:\}{\uparrow}^{\mu^*}[1:]| < m$ and so $\pi{\uparrow}^{\mu^*}[-m]$ precedes b. □

Remark 5 (Goodness adequacy)**.** *If the* goodness *of the chain can be assumed, then the adequate level of an honest proof is nothing else than the highest level having a sufficient (m) number of blocks after the fork point b. In that case, the proof for the above lemma is easy and follows from the prover construction. It always covers the last m blocks of level μ with the respective blocks in level $\mu - 1$.*

Lemma 27. *Suppose the verifier evaluates $\pi_{\mathcal{A}} \geq \pi_B$ in a protocol interaction where B is honest and assume during the comparison that the compared level of the honest party is μ_B. Let $b = \mathsf{LCA}(\pi_{\mathcal{A}}, \pi_B)$ and let μ'_B be the adequate level of π_B with respect to b. Then $\mu'_B \geq \mu_B$.*

Proof. Because μ_B is the compared level of the honest party we have $2^{\mu_B}|\mathsf{C}\{b :\}{\uparrow}^{\mu_B}| > 2^{\mu_B}|\mathsf{C}\{b :\}|$. The proof is by contradiction. Suppose $\mu'_B < \mu_B$. By definition, μ'_B is the maximum level such that $|\pi_B\{b :\}{\uparrow}^{\mu}[1 :]| \geq \max(m, (1-\delta)2^{-\mu}|\pi_B\{b :\}{\uparrow}^{\mu}[1 :]{\downarrow}|)$, therefore μ_B does not satisfy this condition. But we know that $|\pi_B\{b :\}{\uparrow}^{\mu_B}[1 :]| \geq m$ because μ_B was selected by the Verifier. Therefore $2^{\mu_B}|\mathsf{C}_B\{b :\}{\uparrow}^{\mu_B}| < (1-\delta)|\mathsf{C}\{b :\}|$. But μ'_B satisfies goodness, so $2^{\mu'_B}|\mathsf{C}_B\{b :\}{\uparrow}^{\mu'_B}| > (1-\delta)|\mathsf{C}\{b :\}|$. From the last two equations, we obtain $(1-\delta)|\mathsf{C}\{b :\}| > 2^{\mu'_B}|\mathsf{C}\{b :\}{\uparrow}^{\mu'_B}|$, which contradicts the previous equation. ☐

Theorem 17 (Security)**.** *Assuming honest majority, the Non-interactive Proofs of Proof-of-Work construction for computable k-stable monotonic suffix-sensitive predicates is secure with overwhelming probability in κ.*

Proof. By contradiction. Let $m = k_1 + k_2 + k_3$ and let k_1, k_2, k_3 be polynomial functions of κ. Let Q be a k-stable monotonic suffix-sensitive chain predicate. Assume NIPoPoWs on Q is insecure. Then, during an execution at some round r_3, $Q(\mathsf{C})$ is defined and the verifier V disagrees with *some* honest participant. Assume the execution is typical. V communicates with adversary $\mathcal{A}$ and honest prover B. The verifier receives proofs $\pi_{\mathcal{A}}, \pi_B$. Because B is honest, π_B is a proof constructed based on underlying blockchain C_B (with $\pi_B \subseteq \mathsf{C}_B$) which B has adopted during round r_3 at which π_B was generated. Furthermore, $\pi_{\mathcal{A}}$ was generated at round $r'_3 \leq r_3$.

The verifier outputs $\neg Q(\mathsf{C}_B)$, and so $\mathsf{Verify}^Q_{m,k} = \neg Q(\mathsf{C}_B)$. Thus it is necessary that $\pi_{\mathcal{A}} \geq_m \pi_B$, otherwise, because Q is suffix sensitive, Verify^Q would have returned $Q(\mathsf{C}_B)$. We now show that $\pi_{\mathcal{A}} \geq_m \pi_B$ is a negligible event.

Let $b = \mathsf{LCA}(\pi_{\mathcal{A}}, \pi_B)$ and let b^* be the most recent honestly generated block in C_B preceding b (and note that b^* necessarily exists because Genesis is honestly generated). Let the levels of comparison decided by the verifier be $\mu_{\mathcal{A}}$ and μ_B respectively. Let μ'_B be the adequate level of proof π_B with respect to block b. Call $\alpha_{\mathcal{A}} = \pi_{\mathcal{A}}{\uparrow}^{\mu_{\mathcal{A}}}\{b :\}$, $\alpha'_B = \pi_B{\uparrow}^{\mu'_B}\{b :\}$.

We now show three successive claims: First, $\alpha_{\mathcal{A}}$ and $\alpha'_B{\downarrow}$ are mostly disjoint. Second, $\alpha_{\mathcal{A}}$ contains mostly adversarially-generated blocks. And third, the adversary is able to produce this $\alpha_{\mathcal{A}}$ with negligible probability.

Claim 1a: If $\mu'_B \leq \mu_{\mathcal{A}}$ then $\alpha_{\mathcal{A}}[1 :]$ and $\alpha_B[1 :]{\downarrow}$ are completely disjoint.

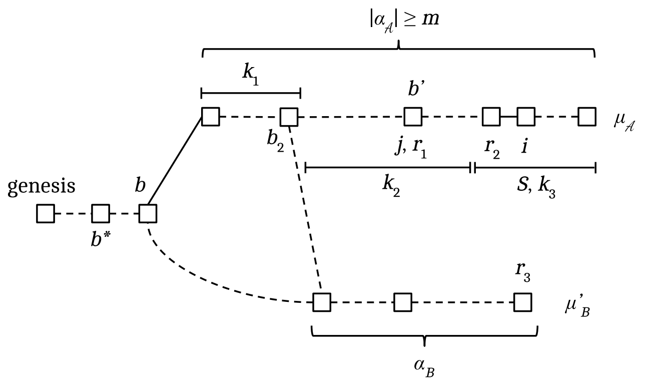
Applying Lemma 26 to $\mathsf{C}_B\{b :\}{\uparrow}^{\mu'_B}$ we see that $\mathsf{C}_B\{b :\}{\uparrow}^{\mu'_B} = \pi_B{\uparrow}^{\mu'_B}\{b :\}$ and so $\pi_B{\uparrow}^{\mu'_B}\{b :\}[1 :] \cap \pi_{\mathcal{A}}{\uparrow}^{\mu_{\mathcal{A}}}\{b :\}[1 :] = \emptyset$.

Claim 1b: If $\mu_{\mathcal{A}} < \mu'_B$ then $|\alpha_{\mathcal{A}}[1 :] \cap \alpha_B{\downarrow}[1 :]| \leq 2^{\mu'_B - \mu_{\mathcal{A}}}k_1$.

First, observe that, because the adversary is winning, therefore $|\alpha_{\mathcal{A}}| > 2^{\mu'_B - \mu_{\mathcal{A}}}m$. Suppose for contradiction that $|\alpha_{\mathcal{A}}[1 :] \cap \alpha_B{\downarrow}[1 :]| > 2^{\mu'_B - \mu_{\mathcal{A}}}k_1$. This means there are indices $1 \leq i < j$ such that $|\mathsf{C}_B{\uparrow}^{\mu_{\mathcal{A}}}[i : j]| > 2^{\mu'_B - \mu_{\mathcal{A}}}k_1$ but $|\mathsf{C}_B{\uparrow}^{\mu_{\mathcal{A}}}[i : j]{\downarrow}{\uparrow}^{\mu'_B}| = 0$. But this contradicts the goodness of $\mathsf{C}_B{\uparrow}^{\mu'_B}$. Therefore there are more than $2^{\mu'_B - \mu_{\mathcal{A}}}(k_2 + k_3)$ blocks in $\alpha_{\mathcal{A}}$ that are not in α_B, and clearly also more than $k_2 + k_3$ blocks.

From Claim 1a and Claim 1b, we conclude that there are at least $k_2 + k_3$ blocks after block b in $\alpha_{\mathcal{A}}$ which do not exist in α_B. We now set $b_2 = \mathsf{LCA}(\mathsf{C}_B, \alpha_{\mathcal{A}})$.

Figure 3.8: Two competing proofs at different levels.



Claim 2: At least k_3 superblocks of $\alpha_{\mathcal{A}}$ are adversarially generated.

We show this by showing that $\alpha_{\mathcal{A}}[k_2 + 1 :]$ contains no honestly mined blocks. By contradiction, assume that the block $\alpha_{\mathcal{A}}[i]$ for some $i \geq k_1 + k_2 + 1$ was honestly generated. This means that an honest party had adopted the chain $\alpha_{\mathcal{A}}[i-1]$ at some round $r_2 \leq r_3$. Because of the way the honest parties adopt chains, the superchain $\alpha_{\mathcal{A}}[: i-1]$ has an underlying properly constructed 0-level anchored chain $\mathsf{C}_{\mathcal{A}}$ such that $\mathsf{C}_{\mathcal{A}} \subseteq \alpha_{\mathcal{A}}[: i-1]$. Let j be the index of block b_2 within $\mathsf{C}_{\mathcal{A}}$. As $\alpha_{\mathcal{A}} \subseteq \mathsf{C}_{\mathcal{A}}$, observe that $|\mathsf{C}_{\mathcal{A}}[j+1 :]| > i - 1 \geq k_2 + k_1$. Therefore $\mathsf{C}_{\mathcal{A}}[: -(k_2 + k_1)] \not\preceq \mathsf{C}_B$. But $\mathsf{C}_{\mathcal{A}}$ was adopted by an honest party at round r_2 which is prior to round r_3 during which C_B was adopted by an honest party. This contradicts the Common Prefix [60] property with parameter k_2. It follows that with overwhelming probability in k_2, the $k_3 = m - k_2 - k_1$ last blocks of the adversarial proof have been adversarially mined.

Claim 3: $\mathcal{A}$ is able to produce a $\alpha_{\mathcal{A}}$ that wins against α_B with negligible probability.

Let b' be the latest honestly generated block in $\alpha_{\mathcal{A}}$, or b^* if no such block exists in $\alpha_{\mathcal{A}}$. Let r_1 be the round when b' was generated. Let j be the index of b'. Consider the set S of consecutive rounds $r_1 \dots r_3$. Every block in $\alpha_{\mathcal{A}}[-k_3 :]$ has been adversarially generated during S and $|\alpha_{\mathcal{A}}[-k_3 :]| = k_3$. C_B is a chain adopted by an honest party at round r_3 and filtering the blocks by the rounds during which they were generated to obtain C^S_B, we see that $\mathsf{C}^S_B = \mathsf{C}_B\{b^* :\}$. But chain $\mathsf{C}^S_B{\uparrow}^{\mu'_B}$ is good with respect to C^S_B. Applying Lemma 25, we obtain that with overwhelming probability $2^{\mu_{\mathcal{A}}}|\alpha_{\mathcal{A}}\{b' :\}| < \frac{1}{3}2^{\mu'_B}|\mathsf{C}^S_B{\uparrow}^{\mu'_B}|$.

But $|\alpha_B| \geq |\mathsf{C}^S_B{\uparrow}^{\mu'_B}|$ and $|\alpha_{\mathcal{A}}\{b' :\}| \geq |\alpha_{\mathcal{A}}| - k_2$, therefore $2^{\mu_{\mathcal{A}}}|\alpha_{\mathcal{A}}| - k_2 < \frac{1}{3}2^{\mu'_B}|\alpha_B|$. But $|\alpha_{\mathcal{A}}| - k_2 \geq k_3$, therefore $\frac{1}{3}2^{\mu'_B}|\alpha_B| > k_3$ and so $2^{\mu'_B}|\alpha_B| > 3k_3$ Taking $k_2 = k_3$, we obtain $2^{\mu_{\mathcal{A}}}|\alpha_{\mathcal{A}}| < \frac{1}{3}3k_3 + k_3 = 2k_3 < 2^{\mu'_B}|\alpha_B|$. But this contradicts the fact that $\pi_{\mathcal{A}} \geq \pi_A$, and so the claim is proven.

Therefore we have proven that $2^{\mu'_B}|\pi_B{\uparrow}^{\mu'_B}| > 2^{\mu_{\mathcal{A}}}|\pi^{\mu_{\mathcal{A}}}_{\mathcal{A}}|$. From the definition of μ_B, we know that $2^{\mu_B}|\pi_B{\uparrow}^{\mu_B}| \geq 2^{\mu'_B}|\pi_B{\uparrow}^{\mu'_B}|$, and therefore we conclude that $2^{\mu_B}|\pi_B{\uparrow}^{\mu_B}| > 2^{\mu_\mathcal{A}}|\pi_\mathcal{A}{\uparrow}^{\mu_\mathcal{A}}|$. □

Remark 6 (Variance attacks)**.** *The critical issue addressed by this security proof is to avoid Bahack-style attack [16] where the adversary constructs "lucky" high-difficulty superblocks without filling in the underlying proof-of-work in the lower levels. Observe that, while setting $m = 1$ "preserves" the proof-of-work in the sense that expectations remain the same, the probability of an adversarial attack becomes approximately proportional to the adversary power if the adversary follows a suitable strategy (for a description of such a strategy, see the parameterization section). With higher values of m, the probability of an adversarial attack drops exponentially in m, even though they maintain constant computational power, and hence satisfy a strong notion of security.*

Remark. Intuitively, the attack of Section 3.9 is neutralized, because our prover takes "goodness" of blockchains into account and the verifier does not compare proofs strictly at the same level.

Remark. We have explored security in the *synchronous* model. We remark that the same construction can work in a *partially synchronous* model by setting $k' = 2k$, where k' is the security parameter of the partially synchronous model and k is the security parameter in the synchronous model. We leave the full treatment of this for future work.

3.10.1 Infix security

We observe that now that we have proven the modified suffix construction secure, the security of infix proofs follows without any modifications in the infix construction. We formally state this in the following corollary.

Under honest majority, the infix NIPoPoW protocol (P, V) is secure for all computable infix-sensitive k-stable monotonic predicates Q, except with negligible probability in κ.

Proof. Assume a typical execution. It suffices to show that the verifier will output the same value $Q(\mathsf{C})$ as some honest prover. Assume honest prover B has adopted a chain C with $Q(\mathsf{C}) = v$ and has provided proof π_B. By Theorem 17 and because the evaluation of $\tilde{\pi}$ is identical in the suffix-sensitive and in the infix-sensitive case, we deduce that $b = \tilde{\pi}[-1]$ will be an honestly adopted block. Furthermore, due to the Common Prefix property [60], b will belong to all honest parties' chains and in the same position, as it is buried under $|\tilde{\chi}| = k$ blocks.

Because Q is infix-sensitive, it will be defined using a witness predicate D. Because Q is stable, we will have $\exists \mathsf{C}' \subseteq \mathsf{C}[:-k] : D(\mathsf{C}')$. But $\mathsf{C}' \subseteq \pi_B$. Let $S = \mathsf{ancestors}(b)$ be the ancestors evaluated by the verifier. As $\mathsf{C}' \subseteq S$, therefore $Q(\mathsf{C}') = Q(S) = v$. □

3.11 Succinctness

3.11.1 Optimistic succinctness

We analyse the patched scheme we saw in Algorithm 26. We will illustrate why our construction is succinct in the honest setting. We then discuss techniques to make the construction succinct in broader adversarial settings.

We first observe that full succinctness in the standard honest majority model is impossible to prove for our construction. To see why, recall that an adversary with sufficiently large mining power can significantly harm superquality as described in Section 3.9.1. By reducing superquality for a sufficiently low level μ, the adversary can cause the honest prover to digress in their proofs from high-level superchains down to low-level superchains, causing a linear proof to be produced.

For instance, if superquality is harmed at $\mu = 3$, the prover will need to digress down to level $\mu = 2$ and include the whole 2-superchain, which, in expectation, will be of size $|\mathsf{C}|/2$.

Having established security in the general case of the standard honest majority model, we now concentrate our succinctness claims to the particular "optimistic" case where the adversary does not use their (minority) computational power or network power.

Definition 59 (Optimistic execution)**.** *We will call an execution* optimistic *if the adversary has $q = 0$ random oracle queries and the messages diffused by honest parties are delivered in random (and not adversarial) order.*

In this setting, the superquality of the chain must be the same as a fully honestly-generated chain generated with no network adversary. Last, for now, we will not allow the adversary to produce any proofs; that is, all proofs consumed by the verifier are honestly-generated.

Theorem 28 (Number of levels)**.** *In any execution, let S denote the set of all blocks produced honestly or adversarially. The number of superblock levels which have at least m blocks are at most* $\log(|S|)$*, with overwhelming probability in m.*

Proof. Each block id in S is generated by the random oracle, so $\Pr[\text{id} \leq T2^{-\mu}] = 2^{-\mu}$. These are independent Bernoulli trials. For each $B \in S$, let $X^{\mu}_{B} \in \{0,1\}$ be the random variable indicating whether the block belongs to level μ and let $D_{\mu} = \sum_{B\in S} X^{\mu}_{B}$ indicate their sum, which is a Binomial distribution with parameters $(|S|, 2^{-\mu})$ and expectation $E[D_{\mu}] = |S|2^{-\mu}$.

All of the X^{μ} are independent. We apply a Binomial Chernoff bound to the sum. We have $\Pr[D_{\mu} \geq (1+\delta)E[D_{\mu}]] \leq \exp(-\frac{\delta^2}{3}E[D_{\mu}])$. Letting $\mu = \log(|S|)$ we have that $E[D_{\mu}] = 1$. Therefore $\Pr[D_{\mu} \geq 1+\delta] \leq \exp(-\frac{\delta^2}{3})$. Requiring $1 + \delta = m$, we get $\Pr[D_{\mu} \geq m] \leq \exp(-\frac{(m-1)^2}{3})$, which is negligible in m. □

The above theorem establishes that the number of superchains is small. What remains to be shown is that few blocks will be included at each superchain level.

Theorem 29 (Large upchain expansion)**.** *Consider an optimistic execution and let* C *be an honestly adopted chain and let* $\mathsf{C}' = \mathsf{C}{\uparrow}^{\mu-1}[i : i + 4m]$ *for any i. Then* $|\mathsf{C}'{\uparrow}^{\mu}| \geq m$ *with overwhelming probability in m.*

Proof. Because each block of level $\mu - 1$ was generated as a query to the random oracle, it constitutes an independent Bernoulli trial and the number of blocks in level μ, namely $\pi{\uparrow}^{\mu}$, is a Binomial distribution with parameters $(4m, 1/2)$. Observing that $E[\mathsf{C}'{\uparrow}^{\mu}] = 2m$ and applying a Chernoff bound, we get $\Pr[|\mathsf{C}'{\uparrow}^{\mu}| \leq m] = \Pr[|\mathsf{C}'{\uparrow}^{\mu}| \leq (1-\frac{1}{2})2m] \leq \exp(-\frac{(1/2)^2}{2}2m)$ which is negligible in m.

□

This probability bounds the probability of fewer than m blocks occurring in the μ level restriction of $(\mu - 1)$-level superchains of more than $4m$ blocks.

Lemma 30 (Small downchain support)**.** *Consider an optimistic execution and let* C *be an honestly adopted chain and* $\mathsf{C}' = \mathsf{C}{\uparrow}^{\mu}\ [i : i + m]$*. Then* $|\mathsf{C}'{\downarrow\!\uparrow}^{\mu-1}\ | \leq 4m$ *with overwhelming probability in* m*.*

Proof. Assume the $(\mu - 1)$-level superchain had at least $4m$ blocks. Then by Theorem 29 it follows that more than m blocks exist in level μ with overwhelming probability in m, which is a contradiction. □

This last lemma establishes the fact that the support of blocks needed under the m-sized chain suffix to move from one level to the one below is small. Based on this, we can establish our theorem on succinctness:

Theorem 31 (Optimistic succinctness)**.** *In an optimistic execution, Non-Interactive Proofs of Proof-of-Work produced by honest provers are succinct with the number of blocks bounded by* $4m\log(|\mathsf{C}|)$*, with overwhelming probability in* m*.*

Proof. Assume C is an honest party's chain. From Theorem 28, the number of levels in the NIPoPoW is at most $\log(|\mathsf{C}|)$ with overwhelming probability in m (note that $|\mathsf{C}| \sim \Theta(|S|)$). First, observe that the count of blocks in the highest level will be less than $4m$ from Theorem 29; otherwise a higher superblock level would exist. From Lemma 22, we know that at all levels μ the chain will be good. Therefore, for each μ superchain C the supporting $(\mu - 1)$-superchain will only need to span the m-long suffix of the μ-superchain above. For the m-long suffix of each superchain of level μ, the supporting superchain of level $\mu - 1$ will have at most $4m$ blocks from Lemma 30. Therefore the size of the proof is $4m\log(|\mathsf{C}|)$. □

In the above theorem, note the linear dependency between the round r during which a proof is generated and the length $|\mathsf{C}|$ of the chain of each honest prover.

3.11.2 Succinctness of adversarial proofs

In the stronger adversarial setting, however, it is possible for the adversary to produce large dummy (incorrect) proofs that expand the verification time; security will not be hurt but it would take more time to complete verification. One may dismiss this as a trivial denial of service attack and have a resource bounded verifier simply stop if it is confronted with such a processing task. However, simply dismissing superpolylogarithmic proofs is an incorrect strategy, as honest provers can produce such longer proofs in case an adversarial miner harms the goodness of the blockchain.

It would therefore be useful for honest provers to have the ability to signal to the verifier that such time expansion is indeed necessary because of an attack on superchain quality, rather than because a malicious prover is simply sending long proofs that will eventually be rejected. With such signaling mechanism, a resource bounded verifier can distinguish between a denial of service attack that may be directed solely to it from a denial of service attack that is launched by an attacker that has the ability to interfere globally with superchain quality.

To facilitate the above signaling, we offer a simple generalization of our construction that achieves this. Our extended construction allows the verifier to stop processing input early, in a streaming fashion, thereby only requiring logarithmic communication complexity per proof received. To achieve this, observe that honest proofs need to be large only if there is a violation of *goodness*. However, goodness is not harmed when the chain is not under attack by the adversarial computational power or network. Therefore, we require the prover to produce a *certificate of badness* in case there is a violation of *goodness* in the blockchain. This certificate will always be logarithmic in size and must be sent prior to the rest of the proof by the prover to the verifier. Because the certificate will be logarithmic in size even in the case of an adversarial attack on the chain, the honest verifier can stop processing the certificate after a logarithmic time bound. If the certificate is claimed to be longer, the honest verifier can reject early by deciding that the prover is adversarial. Looking at the certificate, the honest verifier determines whether there is a possibility for a lack of goodness in the underlying chain. If there's no adversarial computational power in use, the certificate is impossible to produce.

The certificates of badness are produced easily as follows. First, the honest verifier finds the maximum level max-μ at which there are at least m max-μ-superblocks and includes it in the certificate. Then, because there is a violation of goodness there must exist two levels $\mu < \mu'$ such that $2^{\mu}|\mathsf{C}{\uparrow}^{\mu}| > (1+\delta)2^{\mu'}|\mathsf{C}{\uparrow}^{\mu'}|$ in some part C of the honestly adopted chain. But $\mu' - \mu \leq$ max-μ. Therefore, there must exist two adjacent levels $\mu_1 < \mu_2$ which break goodness but with error parameter $(1+\delta)^{1/\text{max-}\mu}$. In particular, it will hold that $2^{\mu_1}|\mathsf{C}{\uparrow}^{\mu_1}| > (1+\delta)^{1/\text{max-}\mu}2^{\mu_2}|\mathsf{C}{\uparrow}^{\mu_2}|$. This condition is direct for the prover to find and trivial for the verifier to check and completes the construction. Note that it is possible that a certificate of badness is produceable where two adjacent levels have more than $(1+\delta)^{1/\text{max-}\mu}$ error even if there is no harm to global goodness; however, these certificates cannot be produced when no adversarial power is in use. The algorithm to do this is shown in Algorithm 27.

Algorithm 27 The $\mathsf{badness}$ prover which generates a succinct certificate of badness

```
function $\mathsf{badness}_{m,\delta}(\mathsf{C})$
    $M \leftarrow \{\mu : |\mathsf{C}{\uparrow}^{\mu}| \geq m\} \setminus \{0\}$
    $\rho \leftarrow 1/\max(M)$
    for $\mu \in M$ do
        for $B \in \mathsf{C}{\uparrow}^{\mu}$ do
            $\mathsf{C}' \leftarrow \mathsf{C}{\uparrow}^{\mu}\{B:\}[:m]$
            if $|\mathsf{C}'| = m$ then
                ▷ Sliding m-sized window
                $\mathsf{C}^* \leftarrow \mathsf{C}'{\downarrow}{\uparrow}^{\mu-1}$
                if $2|\mathsf{C}'| < (1-\delta)^{\rho}|\mathsf{C}^*|$ then
                    return $\mathsf{C}^*$                    ▷ Chain is bad
                end if
            end if
        end for
    end for
    return $\perp$                                          ▷ Chain is good
end function
```

Therefore, we augment the NIPoPoW construction as follows. The honest prover sends a tuple of two items. The first item is empty if the second item is polylogarithmic in the size of the chain; otherwise it is a certificate of badness. The second item is the NIPoPoW proof as in the previous construction. The verifier processes only the first polylogarithmic number of bytes from the incoming proof. If within that portion a certificate of badness is found, it is checked for validity. If it is found to be valid, the whole proof is checked, regardless of size. If it is found to be invalid or no certificate has been provided, then the proof is rejected as invalid. We call the augmented construction *certified NIPoPoWs*.

Lemma 32 (Certified NIPoPoWs succinctness)**.** *If all miners are honest and the network scheduling is random, certified non-interactive proofs-of-proof-of-work produced by the adversary are processed in polylogarithmic time in the size of the chain by honest verifiers, except with negligible probability in m.*

Proof. Because all miners are honest and the network scheduling is random, therefore certificates of badness exist with negligible probability in m. Conditioning on the event that certificates of badness do not exist, the honest verifier will reject the proof in polylogarithmic time. □

We also establish that the modified construction does not harm security below. Security is established in the general case where the adversary has minority mining power.

Theorem 33 (Certified NIPoPoWs security)**.** *Assuming honest majority, certified non-interactive proofs-of-proof-of-work are secure, except with negligible probability in κ.*

Proof. We distinguish two cases: Either goodness has been violated; or it has not been violated. Suppose that goodness has been violated. In that case, an honest prover will include a certificate of badness in their proof and their proof will be processed by an honest verifier.

In the case where goodness is not violated, all honest proofs will be logarithmic in size as established by Lemma 32. Therefore, all honest proofs will be processed by an honest verifier.

Under the condition that all honest proofs will be processed, the rest of the security argument follows immediately from Theorem 17. □

3.11.3 Infix succinctness

Having established the succinctness of the modified suffix construction, the succinctness of the infix construction follows in the next corollary.

The infix NIPoPoW protocol (P, V) is succinct for all computable infix-sensitive k-stable predicates Q in which the witness predicate D depends on a *polylogarithmic* number of blocks $d(|\mathsf{C}|)$.

Proof. As long as the number of blocks on which the predicate depends is polylogarithmic ($< d$) with respect to the chain length, our proofs remain succinct. Specifically, the proof size for the suffix has exactly the same size. Then the part of the proof that is of interest is the output of the followDown algorithm. However, notice that this algorithm will on average produce as many blocks as the difference of levels between B' and E, which is at most logarithmic in the chain size. Hence the proof sizes will be in expectation $(m + |\mathsf{C}'|)\log(|\mathsf{C}|)$, which remains succinct if $|\mathsf{C}'| \in O(polylog(|\mathsf{C}|))$. □

3.12 Gradual Deployment Paths

Our construction requires an upgrade to the consensus layer. We envision that new cryptocurrencies will adopt our construction in order to support efficient light clients. However, existing cryptocurrencies could also benefit by adopting our construction as an upgrade. In this section we outline several possible upgrade paths. We also contribute a novel upgrade approach, a "velvet fork," which allows for gradual deployment without harming unupgraded miners.

3.12.1 Hard Forks and Soft Forks

The obvious way to upgrade a cryptocoin to support our protocol is a hard fork: the block header is modified to include the interlink structure, and the validation rules modified to require that new blocks (after a "flag day") contain a correctly-formed interlink hash.

The safety of a hard fork is debated [33], as they are not "forward compatible". NIPoPoWs can also be implemented by a soft fork. A soft fork construction requires including the interlink not in the block header, but in the coinbase transaction. It is enough to only store a hash of the interlink structure. The only requirement for the NIPoPoWs to work is that the PoW commits to all the pointers within the interlink so that the adversary cannot cause a chain reorganization. If we take that route, then each NIPoPoW will be required to present not only the block header, but also a proof-of-inclusion path within the Merkle tree of transactions proving that the coinbase transaction is indeed part of the block. Once that is established, the coinbase data can be presented, and the verifier will thereby know that the hash of the interlink data structure is correct. Given that in the Bitcoin implementation there is a block size limit, observe that including such proofs-of-inclusion will only increase the NIPoPoW sizes by a constant factor per block, allowing for the communication complexity to remain polylogarithmic.

3.12.2 Velvet Forks

We now describe a novel upgrade path that avoids the need for a fork at all. The key idea is that clients can make use of our scheme, even if only some blocks in the blockchain include the interlink structure. Given that intuitively the changes we will propose require no rule modifications to the consensus layer, we call this technique a *velvet fork* [6].

We require upgraded miners to include the interlink data structure in the form of a new Merkle tree root hash in their coinbase data, similar to a soft fork. An unupgraded miner will ignore this data as comments. We further require the upgraded miners to accept all previously accepted blocks, regardless of whether they have included the interlink data structure or not. Even if the interlink data structure is included and contains invalid data, we require the upgraded miners to accept their containing blocks. Malformed interlink data could be simply of the wrong format, or the pointers could be pointing to superblocks of incorrect levels. Furthermore, the pointers could be pointing to superblocks of the correct level, but not to the most recent block. By requiring upgraded miners to accept all such blocks, we do

[6]After the first manuscript of our related paper was published on the *ePrint* archive, velvet forks were subsequently explored in detail in the follow-up work by Zamyatin et. al. [158]

not modify the set of accepted blocks. Therefore, the upgrade is simply a "recommendation" for miners and not an actual change in the consensus rules. Hence, while a hard fork makes new upgraded blocks invalid to unupgraded clients and a soft fork makes new unupgraded blocks invalid to upgraded clients, the velvet fork has the effect that blocks produced by either upgraded or unupgraded clients are valid for either. In reality, the blockchain is never forked. Only the codebase is upgraded, and the data on the blockchain is interpreted differently.

The reason this can work is because provers and verifiers of our protocol can check the validity of the claims of miners who make false interlink chain claims. An upgraded prover can check whether a block contains correct interlink data and use it. If a block does not contain correct interlink data, the prover can opt not to use those pointers in their proofs. The Verifier verifies all claims of the prover, so adversarial miners cannot cause harm by including invalid data. The one thing the Verifier cannot verify in terms of interlink claims is whether the claimed superblock of a given level is the most recent previous superblock of that level. However, an adversarial prover cannot make use of that to construct winning proofs, as they are only able to present shorter chains in that case. The honest prover can simply ignore such pointers as if they were not included at all.

The velvet prover works as usual, but additionally maintains a *realLink* data structure, which stores the *correct* interlink for each block. Whenever a new winning chain is received from the network, the prover checks it for blocks that it hasn't seen before. For those blocks, it maintains its own realLink data structure which it updates accordingly to make sure it is correct regardless of what the interlink data structure of the received block claims.

Algorithm 28 Supplying the necessary data to calculate a connected $\mathsf{C}\uparrow^\mu$ during a velvet fork.

```
1: function find $\mathsf{C}\uparrow^\mu$($b$, realLink, blockById)
2:     $B \leftarrow \mathsf{C}[-1]$
3:     $\mathsf{aux} \leftarrow \{B\}$
4:     $\pi \leftarrow [\,]$
5:     if $level(B) \geq \mu$ then
6:         $\pi \leftarrow \pi B$
7:     end if
8:     while $B \neq b$ do
9:         $(B, \mathsf{aux'}) \leftarrow \mathsf{followUp}(B, \mu, \mathsf{realLink}, \mathsf{blockById})$
10:        $\mathsf{aux} \leftarrow \mathsf{aux} \cup \mathsf{aux'}$
11:        $\pi \leftarrow \pi B$
12:    end while
13:    return $\pi$, aux
14: end function
```

Algorithm 29 followUp produces the blocks to connect two superblocks in velvet forks.

```
function followUp($B$, $\mu$, realLink, blockById)
    aux $\leftarrow \{B\}$
    while $B \neq$ Gen do
        if $B$.interlink[$\mu$] = realLink[id($B$)][$\mu$] then
            $id \leftarrow B$.interlink[$\mu$]
        else                                                  ▷ Invalid interlink
            $id \leftarrow B$.previd
        end if
        $B \leftarrow$ blockById[$id$]
        aux $\leftarrow$ aux $\cup \{B\}$
        if $level(B) = \mu$ then
            return $B$, aux
        end if
    end while
    return $B$, aux
end function
```

The velvet $C\uparrow$ operator shown in Algorithm 28 is implemented identically as before, except that instead of following the interlink pointer blindly it now calls the helper function *followUp*, shown in Algorithm 29. It accepts block B and level μ and creates a connection from B back to the most recent preceding μ-superblock, by following the interlink pointer if it is correct. Otherwise, it follows the previd link which is available in all blocks, and tries to follow the interlink pointer again from there. Finally, the velvet prover shown in Algorithm 30 simply applies the velvet $C\uparrow$ operator and includes the auxiliary connecting nodes within the final proof. No changes in the verifier are needed; note that in the case of infix proofs the index of the block is used by the verifier; if this information is not provided by the underlying blockchain headers, the index should be included in the interlink structure.

Algorithm 30 The Prove algorithm for the NIPoPoW protocol, modified for a velvet fork.

```
function Prove'$_{m,k}$(C, realLink, blockById)
    $max\mu \leftarrow$ |realLink[id(C[$-k-1$])]|
    $b \leftarrow$ C[0]                                           ▷ Genesis block
    $\tilde{\Pi} \leftarrow \emptyset$
    for $\mu = max\mu$ down to 0 do
        $\pi, aux \leftarrow$ find C[: $-k$]$\uparrow^\mu$ using realLink, blockById
        if $|\pi| \geq m$ then
            $b \leftarrow \pi[-m]$
        end if
        $\tilde{\Pi} \leftarrow \tilde{\Pi} \cup aux$
    end for
    $\chi \leftarrow$ C[$-k$ :]
    return $\tilde{\Pi}\chi$
end function
```

Velvet NIPoPoWs can preserve security if appropriately implemented [88]. Additionally, if a constant minority of miners has upgraded their nodes, then succinctness is also preserved as there is only a constant factor penalty as proven in the following theorem.

Theorem 34. *Velvet non-interactive proofs-of-proof-of-work on honest chains by honest provers remain succinct as long as a constant percentage g of miners has upgraded, with overwhelming probability.*

Proof. From Theorem 31 we know that the proofs π contain only a $O(polylog(m))$ amount of blocks. For each of these blocks, the velvet client needs to include a followUp tail of blocks. Assume a percentage $0 < g \leq 1$ of miners have upgraded with NIPoPoW support. Then the question of whether each block in the honest chain is upgraded follows a Bernoulli distribution. If the velvet proof were to be larger than Δ times the soft fork proof in the number of blocks included, then this would require at least one of the followUp tails to include at least Δ sequential unupgraded blocks. But since the upgrade status of each block is independent, the probability of this occurring is $(1 - g)^{\Delta}$, which is negligible in Δ. □

We would not have been able to pull off this upgrade without modifications to the consensus layer in the sense that the interlink data structure could not have been maintained somewhere independently of the blockchain: It is critical that the proof-of-work commits to the interlink data structure. Interestingly, the interlink data structure does not need to be part of coinbase and can be produced and included in regular transactions by users (such as OP_RETURN transactions). Thus, the miners can be completely oblivious to it, while users and provers make use of the feature, making it a *user-activated velvet fork*. Interested users regularly create transactions containing the most recent interlink pointers so that they are included in the next block. If the transaction makes it to the next block, it can be used by the prover who keeps track of these. Otherwise, if it becomes part of a subsequent block, in which case some of the pointers it contains are invalid, it can be ignored or only partially used.

The necessary changes needed in the various construction algorithms in order to support a velvet fork are shown in Algorithm 28, Algorithm 29, and Algorithm 30.

Remark 7 (Supporting clients with different beliefs)**.** *The interlink format does not depend on parameters m, k. Therefore, it is not necessary to agree on a particular value of these parameters. Instead, the choice of m and k can be a user-configurable parameter to clients. Clients would send a particular m and k as part of their requirement to the prover.*

Chapter 4

Superlight Clients

Will blockchain systems handle the whole world's economic data for the centuries to come? While such lofty visions are often ubiquitous in the cryptocurrency space, it is a practical reality that today's blockchain technology simply does not scale [8]. One aspect of scalability difficulty stems from the data required to be stored and sent over the network when blockchain nodes synchronize with each other or bootstrap from the network for the first time.

As explain in Chapter 2, these data contains two pieces of information: First, the *application data*. This includes transactions, account balances, and smart contract state evolution, and everything else that is included in the block data itself. Secondly, the *consensus data*. This includes consensus-critical information such as proof-of-work (or proof-of-stake) and nonces required to discover the longest chain among a sea of shorter forks — everything that is part of the *block header*. Nodes also need to reach consensus on the application data and ensure it follows the protocol rules for validity, but the application data is not what makes consensus happen. While application data can grow (or shrink) depending on the implementation, consensus data grows unboundedly at a constant linear rate in time. For example, in Bitcoin, while items can be added or removed from the UTXO [28], the number of block headers that need to be stored and communicated to newly bootstrapping nodes grows at a constant rate of 1 block header per 10 minutes in expectation [2]. Similarly, in Ethereum, while smart contracts can be added or destroyed [72], and smart contract state variables added or removed, block headers still grow at a constant rate of 1 block header per 12.5 seconds in expectation.

In this chapter, we leverage the NIPoPoWs developed in Chapter 3 to build superlight clients. We modify our previous construction, *charity* superblock NIPoPoWs of Chapter 3, and introduce the *distill* construction, which we prove both secure and succinct against all (1/3) adversaries (recall that the construction of Chapter 3 was secure against all adversaries, but only optimistically succinct). One critical difference of the construction is that the verifier compares competing proofs against the *same* level μ, and so the weighting with the factors 2^{μ_A} and 2^{μ_B} becomes unnecessary. However, proving these constructions secure and succinct requires some additional theoretical developments.

Our mechanism permanently *prunes* the consensus data in a way that maintains the blockchain's security. Our protocol compresses the amount of consensus data that needs to be stored and exchanged by nodes from linear to polylogarithmic —

an exponential improvement. These reductions affect full nodes and miners alike, and, to our knowledge, are the first of their kind. Our protocol is the first to suggest that nodes need not hold onto chains at all; instead, full nodes and miners collectively only hold a small *sample* of blocks. The rest of the blocks are lost for ever, unless maintained by archival nodes, and are not necessary for achieving consensus or bootstrapping new nodes. We note here that our proposed scheme is not a sharding-based solution. *All* the miners of our protocol will store the *same* data. Sharding solutions can be *composed* with our solution in a per-shard basis to achieve even better scalability.

To achieve these reductions securely, we develop a mathematical framework for the analysis of blockchain systems under *suppression attacks* in which an adversary attempts to silence the generation of selected blocks. For our system to work correctly, it is imperative that the adversary faces difficulty in suppressing *superblocks* (c.f. the suppression attacks and the necessity of certificates of badness in Chapter 3). We prove that, while a 1/2 can perform suppression attacks as explored previously, these blocks cannot be silenced by a 1/3 mining adversary. We begin by giving a construction that allows for logarithmic space mining against 1/3 adversaries, and later improve it to withstand 1/2 adveraries using the technique of *blinded mining*.

In summary, this chapter develops the following notions:

1. We put forth a mechanism which provides exponential improvements in the *consensus* data stored and exchanged between full nodes and miners in proof-of-work settings. Our protocol requires the storage and exchange of only *polylogarithmic* data, even when a new miner is bootstrapping from genesis.

2. We develop a mathematical framework for the analysis of *suppression* attacks, and analyze the security of our protocol therein. Our protocol is secure under honest majority assumptions (a 1/3 adversary for the simple construction, and a 1/2 adversary for the blinded construction) in the random oracle model.

We present this chapter's construction in stages. First, we discuss how an existing miner can compress their full state. Next, we discuss how a newly booting miner can bootstrap from genesis using only the compressed state. Subsequently, we show how a miner with only the compressed state can mine new blocks, giving rise to both *light* and *full* miners. Finally, we assemble our complete protocol, in which *all* miners are *light* miners. These constructions are accompanied by high-level security arguments and building an intuitive understanding of why the protocol works. After the full construction has been presented, the formal security analysis in the random oracle and backbone model follows. This analysis part is also where our mathematical framework for the treatment of suppression attacks is put forth. We conclude by discussing the limitations and shortcomings of our protocol.

4.1 Consensus and Application Data

To pinpoint exactly our contributions to optimizations in consensus data, in this section we review the difference between *application data* and *consensus data* which was discussed in Chapter 2.

Blockchain systems maintain certain *application state*. This state can be used to, for example, determine who owns how much money. There are two primary ways of representing ownership in today's blockchains: A *UTXO*-based system, in which the application state is comprised of the *unspent transaction outputs* that remain available for spending; and an *accounts*-based system, in which the application state is comprised of *accounts and their balances.* The first one is used primarily by Bitcoin, while the second one is used by Ethereum.

The application state evolves over time when *transactions* are applied to it. A transaction is a state evolution operator applied on the application state. Given a previous application state and a transaction, a new application state can be computed. Each block in the chain contains multiple transactions in a particular order. As such, a block is itself a state evolution operator which applies multiple transactions in order. By applying a block to a previous application state, a new application state can be computed.

There are two schools of thought regarding what should be stored in a block. In the first school of thought, only transactions (deltas) are stored. The application state at the end of the blockchain can be computed by starting at the *genesis* application state (an *empty* application state) and *traversing* the blockchain, applying the state evolution described by each block, in order, and arriving at the final application state. This is what Bitcoin does. The other school of thought stores both transactions and the state *after* these transactions have been applied, a so-called *snapshot.* In such systems, if one holds the longest chain, the application state at the end of the chain does not need to be computed by applying any deltas. Instead, a block near the end of the chain can simply be inspected and the application state within it extracted.

It is possible to apply either school of thought to either application state model. Bitcoin only keeps only deltas for a UTXO-based application state. However, nothing prevents Bitcoin from committing to the newly computed UTXO in every block [37, 108, 51], and in fact some Bitcoin forks have already done so. On the other hand, Ethereum keeps both deltas and snapshots in blocks. While the snapshots are not necessary, they are helpful. For the rest of this chapter, *we assume a proof-of-work blockchain in which each block commits to an application state snapshot.* The exact application state format (UTXO, accounts, or something else) is irrelevant for our purposes.

In both schools of thought, it is imperative that the validity of the application data (deltas or snapshots) is verified before a block can be accepted as valid. For example, in a snapshotted system, miners must check that the snapshot committed to a block was obtained by applying the transactions to the previous snapshot.

Let us now discuss how a bootstrapping node can synchronize with the rest of the network. A bootstrapping node is a node holding only the *genesis* block and booting for the first time. A wallet node is interested in the *current* application state that concerns it. For example, it is interested to learn which UTXOs it owns, or how much money is in its own accounts. The *custodial history* of how these assets came to belong to it is irrelevant [51], beyond archival purposes, as long as it can be sure that the assets it holds correspond to the correct application state based on the history that took place. Inspecting or having access to this history itself is not important for consensus purposes. As such, this node can synchronize with the rest of the network using the *SPV* method [116]: It downloads only the block *headers* to determine which chain is the longest one. It then inspects a block near the end of the chain and extracts the balance from the Merkle tree leaf for its own accounts, or for its UTXOs. This is sufficient to know the assets that it owns.

In case some nodes are interested in the history of the blockchain, this history can be maintained by special archival nodes or block explorers, but are not necessary for the maintenance of the security of the network.

A miner bootstrapping their node can function in a similar manner: Download only the block headers to determine the longest chain, then inspect a block near the end of the chain to obtain the application state snapshot. Contrary to a wallet node, the miner must obtain the whole application state so that it can validate new pending transactions as they arrive. As such, the miner downloads the *headers* for the whole chain, and the *full* blocks only for blocks near the end of the chain.

To be more precise, after the longest chain has been determined by comparing block header chain lengths, the k^{th} block from the end is inspected, its application state snapshot is extracted, and the deltas in next k blocks are applied. This is necessary because an adversary can place incorrect snapshots in the most recent k blocks of a blockchain (folklore wisdom suggests $k = 6$ for Bitcoin). While that blockchain will look valid and long to someone verifying only headers, it will have snapshots corresponding to an incorrect application of deltas. However, the adversary cannot modify blocks prior to that, due to Common Prefix.

Note here that the miner does not need to verify the veracity of all historical transactions: If we assume that the majority of the computational power was honest for the duration of history, this ensures that, at all times during the execution, the longest chain represented the correct history of the world (with the exception of up to k blocks towards the end). Under the honest majority assumption, this scheme is as secure as full mining (but see the Discussion section at the end of this chapter for a more nuanced take on this argument under temporary dishonest majority). This is contrary to schemes such as SPV mining in which no snapshots are available.

Application data can grow or shrink. UTXOs can be created or deleted, accounts and smart contracts can be created, updated and destroyed. State variables within smart contracts can also be constructed or destructed. *How* the application data grows is application-dependent. Typically, the application data will increase as the execution continues. There are several attempts to optimize the size of these data [38, 126, 86, 11, 10, 20, 142]. We remind the reader that, in this thesis, we do not focus on these.

Instead, we focus on the size of the *consensus data*, that of block headers $H(ctr \parallel x \parallel s)$. Contrary to the application data, these data increase at a constant linear rate, as block headers are added to the chain. No matter if channels or rollups are used, block headers must keep getting added to the chain. A system designed to survive for the centuries to come must provision for the scalability of this ever-growing part. Even the solutions above that only download block headers do not tackle that problem. As we will see, it is possible to reduce the consensus data and neither store nor communicate all block headers.

A visualization of the comparison between application and consensus data is shown in Figure 4.1. The consensus data (horizontal) grows at an expected constant rate in time. The application data (vertical) may grow (or shrink) depending on the application, and optimizations or pruning methods can be applied on top of them.

A comparison of our pruned mining construction against other constructions (including the construction of Chapter 3) is illustrated in Table 4.1. In all of these protocols, we have a node (the *prover*) that maintains all the necessary state to help a newly booting node (the *verifier* or *client*) synchronize with the rest of

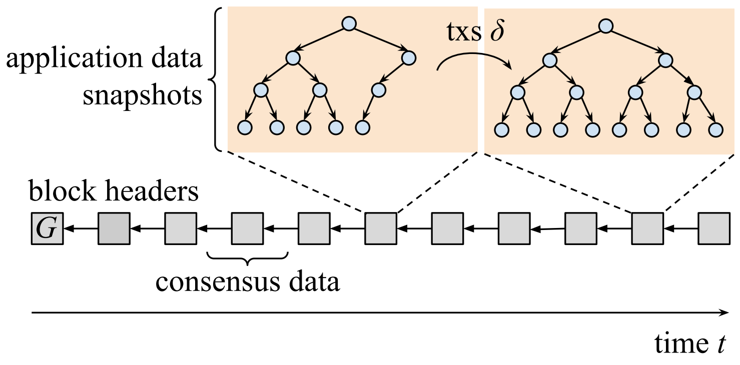


Figure 4.1: A comparison of consensus data (growing horizontally with time) and application data (growing or shrinking vertically depending on the application).

the network. We compare the storage requirements for the prover, as well as the communication complexity during bootstrapping. We are also interested in whether, after synchronizing with the rest of the network, the verifier can function as a fully-fledged miner on its own.

In this table, n denotes the number of blocks in the chain, δ is the size of the transactions in a single block (which may vary with time), a is the size of the snapshot or application state (which may also very with time), c is the size of a block header, and k is the common prefix parameter, the number of blocks required for stability (c.f., [76]). *BTC Full* indicates the full bitcoin miner that synchronizes by downloading all block headers and transactions $n(c + \delta)$. *BTC SPV* is a wallet-only client that downloads only block headers and a single transaction, but requires the prover (the node that serves it this data) to store the full history, as there are no snapshots available. *Ethereum* is a blockchain which uses block headers to synchronize, but makes use of snapshots. Here, the prover can prune block contents, but not block headers (the nc term remains). For the last k blocks, the transaction data of total size $k\delta$ are also needed to verify the veracity of the tip of the chain; for the k^{th} block from the end, only a snapshot of size a is needed. The client can start mining on top of these snapshots (after the $k\delta$ transaction data have been applied to the snapshot of size a). Note that $a \leq n\delta$ and $k \leq n$, and so (asymptotically) $n(c + \delta) \geq nc + k\delta + a$. *Superblock* and *FlyClient* Charity NIPoPoWs (of Chapter 3) allow a full node to function as a prover, only sending consensus data polylogarithmic in n, provided snapshots are available, but the receiving verifie cannot function as a miner or a prover for others. In this chapter, we present a protocol in which the verifier and prover both run on the same node. The prover is only required to store polylogarithmic consensus data, and communication complexity is also polylogarithmic. This is indicated by the term $\mathit{poly}\log(n)c$. The term ka, the application data, remains unaffected.

Proposal	Storage	Communication	Can mine?
BTC Full	$n(c+\delta)$	$n(c+\delta)$	yes
BTC SPV	nc	nc	no
Ethereum	$nc + k\delta + a$	$nc + k\delta + a$	yes
Chapter 3	$nc + k\delta + a$	$\mathit{poly}\log(n)c + k\delta + a$	no
FlyClient	$nc + k\delta + a$	$\mathit{poly}\log(n)c + k\delta + a$	no
This chapter	$\mathit{poly}\log(n)c + k\delta + a$	$\mathit{poly}\log(n)c + k\delta + a$	yes

Table 4.1: A comparison of our results and other constructions. n: the number of blocks in the chain; δ: size of transactions in a block; c: block header size; a: size of snapshot; k: common prefix parameter

4.2 State Compression

How can a newly booting miner synchronize with the rest of the network if block headers have been pruned? It seems impossible to do so securely. The constructions explored in the previous chapters give a glimpse on how this can be achieved.

Among all the block headers that would be maintained by a traditional blockchain protocol, we only keep a small sample of superblocks. *Most* of the block headers headers will be pruned. The small sample of block headers that remains will be polylogarithmic in size and used as evidence that *work* took place throughout history. These sample block headers will be stored by our miners, and will also be sent to new bootstrapping miners when they boot. No other block headers will be stored or communicated beyond these carefully chosen samples. The samples will be chosen to be the same for all miners. As such, some block headers will survive throughout the network, while others will be gone for ever. Once we describe which block headers to keep and which ones to throw away, the construction of our prover will be complete. The rest of the chapter will be to construct a verifier that can distinguish between honest and adversarial application state claims by examining these samples and, of course, proving that this operation is secure.

The key idea is that, once NIPoPoWs have been developed, no blockchains need to be maintained. Miners can only store NIPoPoWs. When mining, they can extend their existing NIPoPoWs into new NIPoPoWs. The data broadcast on the network can consist of NIPoPoWs only.

Let us begin our discussion by pinpointing which samples among all block headers will be maintained. We will slightly change the sampling process as compared to the previous chapters. We first present our *compression algorithm*: The code that can take in a full chain and perform the sampling. These block header samples will be the only ones that survive in our final protocol design. The compression algorithm takes in a full chain and produces the desired samples, but will not form part of our final protocol. In the final protocol, no full chain is to be found. However, the compression algorithm will prove educational in understanding the final protocol (and can also be used, once, to transition a full miner into a light miner). We will also reuse our compression algorithm in the final light miner construction, despite no full chains ever appearing.

Similar to the charity construction of Chapter 3, our sampling will be performed by *only keeping sufficiently high-level superblocks* and throwing away blocks of low levels. We will keep very high levels (so, very few blocks) near genesis and far back in history. As we get closer to the present, we will start including more and more samples, and so the threshold in our superblock level will decrease. Near the tip (the most recent block) of the blockchain, we will eventually get down to level 0 and keep all blocks.

The samples that we keep will *evolve* as the blockchain grows. A sample that was once selected for inclusion may be thrown away later. However, any sample that is thrown away at some point will never again be needed in the future. This property, of ensuring that the sampling is safe and that no samples discarded will be needed again in the future, is the *online* property of our protocol. It will eventually allow us to build a protocol where no full chain is needed, anywhere.

Given a chain C that we wish to compress, first, we keep the most recent k blocks aside, and let us call them χ. These are *unstable* and will need to always be stored. Besides, any miner that wishes to synchronize with us will need to look at them to arrive at a valid snapshot. For the next part, we only consider the *stable* part of the chain. For our sampling process, we begin by *the highest* level ℓ that has *at least* $2m$ blocks in it. We will include this level in earnest: All ℓ-superblocks will be included in our sampling. For every level below ℓ, we will include at least the $2m$ most recently generated blocks of that level, but occassionally more. To consider whether to include more blocks than $2m$ blocks in a level μ, we look at the m^{th} most recent block b in the level $\mu + 1$ *immediately above*. We include all μ-superblocks that are *more recent* than block b. Let us make this description more precise by writing it out in pseudocode.

Our chain compression algorithm $\mathsf{Compress}_{m,k}(\mathsf{C})$ is illustrated in Algorithm 31. It uses the helper function $\mathsf{Dissolve}_{m,k}(\mathsf{C})$ to obtain the highest level ℓ, the unstable suffix χ and a set $\mathcal{D}[\mu]$ of blocks sampled from the stable part of the chain at each level $\mu \leq \ell$. All of these levels are combined into a big chain π, which is sparse at the beginning and dense towards the end. The final compressed state consists of π, the stable part, and χ, the unstable part. Together, these form a chain. Let us now examine the inner workings of $\mathsf{Dissolve}_{m,k}(\mathsf{C})$. This function separates the stable part C^* of the chain and the unstable part χ. In the trivial case that our stable chain has no more than $2m$ blocks, all of them are included. Otherwise, the highest level ℓ with at least $2m$ blocks is extracted and included in earnest. Then, the levels are traversed downwards. For every level μ, the last $2m$ blocks are always included. This is captured by the term $\mathsf{C}^*{\uparrow}^{\mu}\,[-2m{:}]$. Additionally, we look at the m^{th} most recent block b from the end at level $\mu + 1$, that is $\mathsf{C}^*{\uparrow}^{\mu+1}\,[-m]$. For level μ, we also include all the blocks succeeding b, that is $\mathsf{C}^*{\uparrow}^{\mu}\,\{b{:}\}$.

Algorithm 31 Chain compression algorithm for transitioning a full miner to a logspace miner. Given a full chain, it compresses it into logspace state.

```
function $\mathsf{Dissolve}_{m,k}(\mathsf{C})$
    $\mathsf{C}^* \leftarrow \mathsf{C}[: -k]$
    $\mathcal{D} \leftarrow \emptyset$
    if $|\mathsf{C}^*| \geq 2m$ then
        $\ell \leftarrow \max\{\mu : |\mathsf{C}^*{\uparrow}^{\mu}| \geq 2m\}$
        $\mathcal{D}[\ell] \leftarrow \mathsf{C}^*{\uparrow}^{\ell}$
        for $\mu \leftarrow \ell - 1$ down to 0 do
            $b \leftarrow \mathsf{C}^*{\uparrow}^{\mu+1}[-m]$
            $\mathcal{D}[\mu] \leftarrow \mathsf{C}^*{\uparrow}^{\mu}[-2m:] \cup \mathsf{C}^*{\uparrow}^{\mu}\{b:\}$
        end for
    else
        $\mathcal{D}[0] \leftarrow \mathsf{C}^*$
    end if
    $\chi \leftarrow \mathsf{C}[-k:]$
    return $(\mathcal{D}, \ell, \chi)$
end function
function $\mathsf{Compress}_{m,k}(\mathsf{C})$
    $(\mathcal{D}, \ell, \chi) \leftarrow \mathsf{Dissolve}_{m,k}(\mathsf{C})$
    $\pi \leftarrow \bigcup_{\mu=0}^{\ell} \mathcal{D}[\mu]$
    return $\pi\chi$
end function
```

It may not yet be clear why this selection of block headers will lead to a secure protocol, but let us argue that this sampling is polylogarithmic in $|\mathsf{C}|$, considering that m and k are constants that do not grow as the execution progresses.

Theorem 35 (Succinctness)**.** *The construction of Algorithm 31 samples a polylogarithmic number of blocks with respect to the length of the chain* C*.*

Sketch. Firstly, the number ℓ of levels of interest is $\Theta(\log|\mathsf{C}|)$. Next, each level μ has either $2m$ blocks or more. $2m$ is a constant, so this is irrelevant. But the *more* blocks cannot be many more either: We are counting the μ-superblocks following the m^{th} block b at the level $\mu + 1$ above. How many can these be? They are indeed about $2m$. For, suppose for contradiction that they were many more than $2m$. But every block of level μ has a $\frac{1}{2}$ probability of also being a $\mu + 1$ level block. If there were, say, $4m$ instead of $2m$ superblocks of level μ following block b, then b would not be the m^{th} block from the end, but the $2m^{\text{th}}$ one! With high probability (with foresight, utilizing a Chernoff bound), $4m$ can be taken as an upper bound. As such, there will be $2m\log(|\mathsf{C}|) + k$ blocks sampled in expectation, and, with high probability, not many more. $\square$

We make this bound and argument more precise in the Analysis section.

4.3 Fast Synchronization

We have seen how a full miner can compress their state into a polylogarithmic *sample* $\pi\chi$ of blocks. But what is the use of this? We will now build the other side of the protocol: A node, and future miner, booting to the network for the first time, but holding only genesis $\mathcal{G}$. The node is also parametrized by the security parameters m and k. This node wishes to learn where to mine.

For now, let us assume that the rest of the network consists of full miners, and only one node is a light node. The first step of the neophyte is to determine *what the current tip and snapshot* are. The light miner can then start mining on top of that tip, extending its application data snapshot. It does not need to know the blocks preceding the tip! Of course, this node will not be helpful towards bootstrapping *yet more* nodes, but no matter — it can still mine as if it were a full miner, and just as securely, as long as the tip can be correctly discerned.

The protocol works as follows. Initially, the newly booting node (a NIPoPoW verifier), connects to multiple full nodes (the NIPoPoW prover) Each of these full nodes *compresses* their state using Algorithm 31 and sends the compressed state, or *proof* $\Pi = \pi\chi$, to the verifier. More concretely, the full node sends the block headers corresponding to the blocks in π (of size $c \cdot \mathit{poly}\log(n)$). For the blocks in χ, the full node sends the whole application snapshot (of size a) stored in $\chi[-k]$ and the transactions (of size $k\delta$) stored in χ. Naturally, the adversary can send any string as a claimed proof. The verifier checks that Π forms a chain, i.e., that all blocks are connected with interlinks and so they have been produced in the chronological order presented, and also that the first block in Π is the genesis block $\mathcal{G}$ that it knows. It then extracts the last k blocks as χ and the rest as π. It inspects the application data snapshot from $\chi[-k]$ and ensures that the transactions in χ can be cleanly applied. This allows it to obtain the application state at the end of $\pi\chi$, which, in honest cases, is the same as the application snapshot at the end of the underlying blockchain. If any of these checks fail, the particular connection is considered compromised and closed.

The verifier receives and verifies a series of such proofs, each consisting of a stable part π and an unstable part χ, with $|\chi| = k$. Given multiple such proofs $\Pi_1, \Pi_2, \cdots, \Pi_v$, the prover begins inspecting the proofs and comparing one against the other in a pairwise fashion. First, Π_1 is compared against Π_2, and one of them is deemed to be the best (using a mechanism we will soon study). The process continues until only one of them remains. As long as at least one proof was honestly generated, our protocol will arrive at a suffix χ that is *admissible*. This means that our light node will arrive at a snapshot which a *full node miner* booting for the first time from genesis *could* also have arrived at. Upon taking this decision, the light miner stores $\pi\chi$ in its state.

The light miner can then start mining on top of $\chi[-1]$ to produce further blocks and to fully verify the validity of incoming network transactions in its mempool. After all, it is holding onto an application snapshot. These blocks can be broadcast to the network and will be accepted by the rest of the miners, despite our light miner not holding the full chain leading from genesis up to the newly mined block. The light miner can also understand and verify newly mined blocks of others. It can also deal with chain reorganizations: In case a reorganization of up to k blocks occurs, the light miner holds the whole of χ and can verify the state transitions completely. As for reorganizations of more than k blocks long, these will never occur (except with negligible probability) due to the Common Prefix property.

As this miner is not interested in helping bootstrap others, it can even throw away π once it has booted up. Furthermore, every time a new block is mined (either by itself or by someone else), it can append it to χ and then truncate χ to only keep the k most recent blocks. However, in the full protocol, described in the next section, the miner will need to hold on to (and update) π to allow others to bootstrap.

Let us now study the security-critical portion of our protocol, namely how the verifier compares two different proofs Π and Π'. Given two proofs Π and Π', the algorithm must decide which one is *best* or captures the most proof of work. In other words, it must conceptually correspond to the *longest underlying chain*, or the underlying chain with the most work. The comparison algorithm is illustrated in Algorithm 32. The comparison is performed as follows. Initially, the two proofs Π and Π' are verified for syntactic validity: That Π begins with $\mathcal{G}$, it is a chain, and that χ contains valid transactions extending the application data snapshot contained in $\chi[0]$. The comparison continues by invoking the $\mathsf{Dissolve}_{m,k}(\Pi)$ function of Algorithm 31 on each of Π and Π'. As before, this function extracts the maximum level ℓ containing at least $2m$ blocks. Then it picks the required blocks from each level, with at least $2m$ blocks per level, but also a sufficient number of blocks per level to *span* the last m blocks in the level above. Contrary to the invocation in Algorithm 31, we are not passing the full chain to the function; instead, we are passing a chain which has already undergone compression. As such, if the compressed state was honestly generated, the triplet $(\chi, \ell, \mathcal{D})$ on the verifier end will be the same as the triplet on the prover end, because $\mathsf{compress}_{m,k}(\mathsf{C}) = \mathsf{compress}_{m,k}(\mathsf{compress}_{m,k}(\mathsf{C}))$ (but may be something else in case of adversarial proofs).

Algorithm 32 The state comparison algorithm.

```
1:  function $\mathsf{maxvalid}_{m,k}(\Pi, \Pi')$
2:      if $\Pi$ is not valid then
3:          return $\Pi'$
4:      end if
5:      if $\Pi'$ is not valid then
6:          return $\Pi$
7:      end if
8:      $(\chi, \ell, \mathcal{D}) \leftarrow \mathsf{Dissolve}_{m,k}(\Pi)$
9:      $(\chi', \ell', \mathcal{D}') \leftarrow \mathsf{Dissolve}_{m,k}(\Pi')$
10:     $M \leftarrow \{\mu \in \mathbb{N} : \mathcal{D}[\mu] \cap \mathcal{D}'[\mu] \neq \emptyset\}$
11:     if $M = \emptyset$ then
12:         if $\ell' > \ell$ then
13:             return $\Pi'$
14:         end if
15:         return $\Pi$
16:     end if
17:     $\mu \leftarrow \min M$
18:     $b \leftarrow (\mathcal{D}[\mu] \cap \mathcal{D}'[\mu])[-1]$
19:     if $|\mathcal{D}'[\mu]\{b{:}\}| > |\mathcal{D}[\mu]\{b{:}\}|$ then
20:         return $\Pi'$
21:     end if
22:     return $\Pi$
23: end function
```

Only once the two proofs are stratified into levels $\mathcal{D}$, the comparison algorithm attempts to choose a level μ at which the comparison will be performed. This level is the *minimum* level μ for which both provers have provided blocks (note that it is not sufficient that both provers have provided the same block at the same level; it must also have been selected in the same index of $\mathcal{D}$). In the edge case that *no* such level can be found, the prover with the higher ℓ wins (if no such level is found *and* they share the same level, it is irrelevant which prover will win). In the normal case that a level *is* found, then the comparison takes place by taking account *only* blocks of that level. The comparison begins by finding the most recent block shared by the two parties at that level, $(\mathcal{D}[\mu] \cap \mathcal{D}'[\mu])[-1]$. We call this the *lowest common ancestor* b. The blocks of the selected level *following* block b (which must necessarily be disjoint by the definition of b) are then counted, and the party with the most blocks wins.

Note the difference between the construction of this chapter (the *distill construction*) and the construction of Chapter 3 (the *charity construction*): The comparison here is performed in a unified level μ, and no weighting is applied.

Let us give a high-level intuition of why this protocol chooses the longest chain. The key idea is that, in addition to the Common Prefix property holding for regular blocks, this property also holds for μ-superblocks at any level. More precisely, if there is a forking point b, the adversary could not have produced more than m superblocks of level μ faster than the honest parties can produce m superblocks of level μ. This property stands at the heart of the following theorem.

Theorem 36 (Security)**.** *When the honest verifier of Algorithm 32 receives a proof* Π *constructed by an honest party using Algorithm 31 and a proof* Π' *constructed by the adversary, it will decide in favour of the honest proof, unless the adversary is playing honestly and* Π' *was generated according to protocol.*

Sketch. First, consider the case that $M \neq \emptyset$. If the comparison is performed at level $\mu = 0$, this is akin to comparing traditional chains and the theorem holds due to the Common Prefix property.

If the comparison is performed at a level $\mu > 0$, then we apply the extended Common Prefix property at level μ. By the minimality of μ, there will be at least m blocks of the appropriate level following b and so the honest parties will win.

Lastly, if $M = \emptyset$, then we can apply the extended Common Prefix property at the highest level ℓ achieved by the honest party. By construction, the honest party holds at least $2m$ blocks at this level. Because the adversary must have achieved a better $\ell' > \ell$ to win, she must also have at least $2m$ blocks of a higher level, but these are also of level ℓ. But this contradicts the extended Common Prefix property, giving us the desired result. □

While this gives some intuition about why the protocol is designed the way it is, the core security argument pertains to arguing why the extended Common Prefix property holds. We formally prove this statement in the Analysis section for 1/3 adversaries, where we also make the security theorem more precise.

4.4 Mining New Blocks

So far, we have used full nodes to help bootstrap newly booting miners. Can light miners be used to bootstrap newly booting miners instead? If we can achieve this, then we might as well get rid of full nodes altogether.

Our light miner already holds a valid proof $\Pi = \pi\chi$ corresponding to an underlying honest full node chain C at the time it is bootstrapped by others. Before further blocks are mined on the network (either by itself, or by others), it can send this Π to newly booting miners, and they, too, will be convinced of the current application data snapshot. The question is how to update this Π when a new block is mined. Suppose a new block b is mined on top of C, either by our light miner or by someone else. The underlying honest chain then becomes $\mathsf{C}' = \mathsf{C}b$. Can we produce a proof Π' corresponding to C' by only utilizing Π? More specifically, given $\Pi = \mathsf{Compress}_{m,k}(\mathsf{C})$ and b, but not given C, can we produce $\Pi' = \mathsf{Compress}_{m,k}(\mathsf{C}b)$? Indeed we can. In fact, it is as simple as evaluating $\mathsf{C}' = \mathsf{Compress}_{m,k}(\Pi b)$.

Theorem 37 (Online)**.** *Consider $\Pi = \mathsf{Compress}_{m,k}(\mathsf{C})$ generated about an underlying honest chain C, and a block b mined on top of C. Then $\mathsf{Compress}_{m,k}(\mathsf{C}b) = \mathsf{Compress}_{m,k}(\Pi b)$.*

Proof. Consider which blocks are sampled and which blocks are pruned when calling $\mathsf{Compress}_{m,k}(\mathsf{C}b)$. Clearly the block b will be included in both the output of $\mathsf{Compress}_{m,k}(\mathsf{C}b)$ as well as the output of $\mathsf{Compress}_{m,k}(\Pi b)$. All the other blocks selected by $\mathsf{Compress}_{m,k}(\mathsf{C}b)$ will already exist in Π, and in the correct positions. For, the blocks selected from a level are the last $2m$ of a level, or the last m spanning the level above, and adding block b at the end can only render a previously sampled block irrelevant, but not add further block requirements from the past. $\square$

Note also that, when mining a new block b, all the data required to compute the interlink pointers of b is readily available in $\pi\chi$, as π contains the most recent $2m$ blocks of every level, and only the most recent one is needed for interlinking.

Algorithm 33 The final logspace miner.

```
1: $\Pi \leftarrow \emptyset$
2: function $\mathsf{Init}_{m,k}(\overline{\Pi})$
3:     for $\Pi' \in \overline{\Pi}$ do
4:         $\Pi \leftarrow \mathsf{maxvalid}_{m,k}(\Pi', \Pi)$
5:     end for
6: end function
7: function $\mathsf{Mine}_{m,k}(\mathrm{x})$
8:     $b \leftarrow \mathsf{pow}(\Pi[-1], x)$
9:     if $b \neq \epsilon$ then
10:        $\Pi \leftarrow \mathsf{Compress}_{m,k}(\Pi b)$
11:        broadcast($\Pi$)
12:    end if
13: end function
14: upon BootstrapRequest do
15:    return $\Pi$
16: end upon
17: upon NewBlockReceived($\chi'$) do
18:    $\chi \leftarrow \Pi[-k{:}]$
19:    $\pi \leftarrow \Pi[{:}-k]$
20:    if $\chi'$ is a chain $\wedge\ \chi'[0] \in \chi$ then
21:        $b \leftarrow (\chi \cap \chi')[-1]$
22:        if $|\chi'\{b{:}\}| > |\chi\{b{:}\}|$ then
23:            Validate $\chi'$ state transitions starting from $b$
24:            $\Pi \leftarrow \mathsf{Compress}_{m,k}(\pi\chi\{{:}b\}\chi'\{b{:}\})$
25:            broadcast($\Pi$)
26:        end if
27:    end if
28: end upon
```

Our final light miner therefore works as follows. It maintains a *current* proof $\Pi = \pi\chi$ and mines using $\chi[-1]$ as the chain tip. If it is successful in mining b on top of χ, it replaces Π by setting it to $\Pi' = \mathsf{Compress}_{m,k}(\Pi b)$ and broadcasts this to the network. As all of the other online miners, light or full, will hold their own χ^* not differing more than k blocks from χ, it is, in fact, sufficient that it broadcasts the new $\chi' = \chi[1{:}]b$ portion of Π'. Now the newly computed Π corresponds to the chain $\mathsf{C}b$, which the miner never sees, as it has been pruned. Regardless, Π can be used to bootstrap new light miners from genesis.

Consider now the case that our light miner holds a $\Pi = \pi\chi$ and a different miner mines a new block b. By the Common Prefix property, this block will not deviate more than k blocks from the χ that our light miner already holds. Typically, it will be just a block on top of χ, but occassionally it could correspond to a chain reorganization up to k blocks long. In the case of a reorganization, the light miner requests the last k blocks χ' on top of which b was mined. These can be provided to us if the block b was mined by a light or a full miner, as both hold and can send χ'. The blocks in χ' will intersect the previously known χ at some fork point. The light miner checks that the transactions included in this χ' can be applied to the application data snapshot that the light miner has independently calculated for the fork point. This amounts to full block validation. The light miner also checks that the newly mined block really does correspond to a longer chain and that a reorganization is warranted by ensuring that there are more blocks in χ' after the fork point b than there are in χ after the fork point (i.e., that $|\chi'\{b{:}\}| > |\chi\{b{:}\}|$). It then replaces the stored proof by setting Π to be the proof corresponding to $\pi\chi$ when the portion of χ after the most recent common block between χ and χ' is replaced by the blocks in χ', i.e., it updates its stored proof to be $\Pi' = \mathsf{Compress}_{m,k}(\pi\chi\{{:}b\}\chi'\{b{:}\})$.

The light miner is illustrated in Algorithm 33. At this point, full nodes are no longer necessary. Light miners can bootstrap from genesis. They have all the data needed to mine on their own, and to validate newly mined blocks from the network. If a newly booting light miner wishes to synchronize with the network, they have sufficient data to help them do so. The blockchain protocol remains exactly the same as in traditional blockchains, but all the instances of *chains* are replaced by *proofs* instead. Light miners mine on top of their current proof instead of mining on top of a chain. When they discover a new block, they send the newly computed proof instead of a chain. This concludes our construction.

4.5 Block Suppression

We begin our analysis by developing a probabilistic framework to study whether the adversary can *suppress* blocks of her choice. The central definition here is the notion of a Q-block, a block that possesses a certain property — such as being a μ-superblock for some $\mu \in \mathbb{N}$. The main theorem we will eventually prove is a generalization of the Common Prefix property: That the Common Prefix property holds for blocks filtered by any attribute Q. This will allow us to prove our protocol is secure by instantiating Q-blocks as μ-superblocks.

We define a Q-block as a block satisfying a predicate Q on its hash. Note that this evaluation does not depend on any particular execution.

Definition 60 (Q-block)**.** *A* block property *is a predicate Q defined on a hash output $h \in \{0,1\}^\kappa$. Given a block property Q, a valid block with hash h is called a Q-block if $Q(h)$ is true.*

The block properties we are interested in will be evaluated as $Q(H(\langle ctr, s, x\rangle))$ in actual executions for particular blocks. As such, we will be interested in properties which are polynomially computable given h as the input.

Definitions of random variables.

Recall a query of a party is *successful* if it submits a triple (ctr, s, x) such that $H(ctr, s, x) \leq T$. Let us generalize these definitions for block properties. Consider a block property Q. Let $\xi_Q = \Pr[Q(h) | h \leq T]$, when h is uniformly distributed over the range of the hash function. For each round i, query $j \in [q]$, and $k \in [t]$ (the k^{th} party controlled by the adversary), we define Boolean random variables $X_Q(i), Y_Q(i)$ and $Z_Q(i,j,k)$ as follows. If at round i an honest party obtains a Q-block, then $X_Q(i) = 1$, otherwise $X_Q(i) = 0$. If at round i exactly one honest party obtains a Q-block, then $Y_Q(i) = 1$, otherwise $Y_Q(i) = 0$. Regarding the adversary, if at round i, the j^{th} query of the k^{th} corrupted party obtains a Q-block, then $Z_Q(i,j,k) = 1$, otherwise $Z_Q(i,j,k) = 0$. Define also $Z_Q(i) = \sum_{k=1}^{t}\sum_{j=1}^{q} Z_Q(i,j,k)$. For a set of rounds S, let $X_Q(S) = \sum_{r\in S} X_Q(r)$ and similarly define $Y_Q(S)$ and $Z_Q(S)$. We drop the subscript from all variables X, Y, Z, when the Q-block is simply the property of being a valid block. Further, if $X(i) = 1$, we call i a *successful round* and if $Y(i) = 1$, a *uniquely successful round*.

As in the backbone model, the probability f that at least one honest party computes a solution at given round is an important parameter. Writing $p = T/2^\kappa$ for the probability of success of a single query, we have

$$(1-f)pq(n-t) \le f = \mathbb{E}[X(i)] = 1-(1-p)^{q(n-t)} \le pq(n-t).$$

The following bounds relate the expectations of the random variables defined above to f, for all i and block properties Q.

$$\xi_Q f \le \mathbb{E}[X_Q(i)] \le \frac{\xi_Q f}{1-f}, \quad \xi_Q f(1-f) < \mathbb{E}[Y_Q(i)],$$
$$\mathbb{E}[Z_Q(i)] \le \frac{\xi_Q f}{1-f} \cdot \frac{t}{n-t}.$$

For the derivations of these inequalities see Garay et al. [58].

Typical executions. We now define our typical set of executions, extending the typical executions defined in Chapter 2. This follows the backbone model, but extended to include block properties. Informally, this set consists of those executions with polynomially many rounds and with the property that all the random variables of interest over sufficiently many (at least $\lambda = \Omega(\kappa)$) consecutive rounds do not deviate too much from their expectation. To this end, recall the following terms. An *insertion* occurs when, given a chain C with two consecutive blocks B and B', a block B^* created after B' is such that B, B^*, B' form three consecutive blocks of a valid chain. A *copy* occurs if the same block exists in two different positions. A *prediction* occurs when a block extends one which was computed at a later round.

Definition 61 (Typical execution). *For a real $\varepsilon \in (f, \frac{1}{4})$, integer λ, and a collection of polynomially many block properties $\mathcal{Q}$, we say an execution is* $\mathcal{Q}$-typical *(or simply typical), if the following hold.*

- *For any $Q \in \mathcal{Q}$ and any set S of at least λ/ξ_Q consecutive rounds we have*

$$(1-\varepsilon)\,\mathbb{E}[X_Q(S)] < X_Q(S) < (1+\varepsilon)\,\mathbb{E}[X_Q(S)], \tag{4.1}$$
$$(1-\varepsilon)\,\mathbb{E}[Y_Q(S)] < Y_Q(S), \tag{4.2}$$
$$Z_Q(S) < \mathbb{E}[Z_Q(S)] + \varepsilon\,\mathbb{E}[Y_Q(S)]. \tag{4.3}$$

- *No insertions, no copies, and no predictions occurred.*

Theorem 38. *If $t < (1-\delta)(n-t)$ with $\delta > 3\varepsilon + 3f$, an execution is typical with probability $1 - e^{-\Omega(\varepsilon^2 f\lambda)}$.*

Proof. The proof uses standard Chernoff bounds, along the lines of [58]. We just note that the variables $X_Q(i)$ (and similarly $Y_Q(i)$ and $Z_Q(i,j,k)$) are independent Bernoulli trials for each Q and successful with probability $\Theta(\xi_Q f)$. In addition, a union bound is applied over all Q. □

Lemma 39. *Assume $t < (1-\delta)(n-t)$ with $\delta > 3\varepsilon + 3f$ and a $\mathcal{Q}$-typical execution. Then, the following hold for any $Q \in \mathcal{Q}$ and any set S of at least λ/ξ_Q consecutive rounds.*

(a) $(1-\varepsilon)\xi_Q f|S| < X_Q(S) < (1+\varepsilon)\cdot\frac{\xi_Q f}{1-f}\cdot|S|$.

(b) $(1-\frac{\delta}{3})\xi_Q f|S| < (1-\varepsilon)\xi_Q f(1-f)|S| < Y_Q(S)$.

(c) $Z_Q(S) < (\frac{t}{n-t}\cdot\frac{1}{1-f}+\varepsilon)\cdot\xi_Q f|S| \leq (1-\frac{2\delta}{3})\xi_Q f|S|$.

(e) $Z_Q(S) < Y_Q(S)$.

Proof. This follows with straightforward calculations from the properties of a typical execution, the bounds on the expectations of the involved random variables, and the assumed bounds on $t/n, \delta, \varepsilon$ and f. ☐

We now establish an upper bound in the number of Q-blocks an adversary can suppress, regardless of what attack method she follows.

Uniquely successful rounds have the following important property [58].

Lemma 40 (Pairing)**.** *For any i and any pair of distinct blocks $\mathsf{C}[i]$ and $\mathsf{C}'[i]$, if $\mathsf{C}[i]$ was computed by an honest party in a uniquely successful round, then $\mathsf{C}'[i]$ was computed by the adversary.*

Proof. Let r be the uniquely successful round that $\mathsf{C}[i]$ was computed. No honest party would extend $\mathsf{C}'[i-1]$ at a round later than r, since every honest party would have a chain of length at least i. Similarly, if an honest party computed $\mathsf{C}'[i]$ at some round earlier than r, then no honest party would have extended $\mathsf{C}[i-1]$ at round r. Finally, $\mathsf{C}'[i]$ cannot have been computed by an honest party at round r, since r was a uniquely successful round. ☐

Lemma 41 (Suppression)**.** *If r is a uniquely successful round and the corresponding block does not belong to the chain of an honest party at a later round, then there is a set of consecutive rounds S such that $r \in S$ and $Y(S) \leq Z(S)$.*

Proof. Let C be the chain of the honest party that was successful at round r and u the depth of the corresponding block. Let r' be the first round after r in which an honest party has a chain C' which does not contain block $\mathsf{C}[u]$. Let $\mathsf{C}'[u']$ the last block of C' at round r'. Let $\mathsf{C}[u^*] = \mathsf{C}'[u^*]$ be the last honest block on the common prefix of C and C', and let r^* be its timestamp. We claim that the set $S = \{i : r^* < i < r'\}$ satisfies the requirements of the statement. Clearly, $r \in S$. Let us verify that $Y(S) \leq Z(S)$. Indeed, if $\mathsf{C}^*[v]$ is any block computed during a uniquely successful round $i \in S$, it must hold $u^* < v \leq u'$. The first inequality is because the party computing $\mathsf{C}^*[v]$ knows of $\mathsf{C}[u^*]$ (it was announced at round r^* and received by round $i > r^*$) and would not mine on a shorter chain. The second inequality holds because $v > u'$ contradicts an honest party having a chain of length u' at round $r' > i$ (since $\mathsf{C}^*[v]$ was received by round r'). The inequality then follows by Lemma 40, since it is always possible to find a block distinct from $\mathsf{C}^*[v]$ on C or C' (we may use C', unless $\mathsf{C}^*[v]$ is on C', in which case—due to the minimality of r'—we have $v < u$ and we can use C). ☐

An observation that follows from the above lemma is that if the adversary manages to suppress a Q-block from the chain of an honest party and this Q-block was computed in a uniquely successful round, then we can associate with it an adversarial block. In particular, if r is a uniquely successful round and the corresponding block does not belong to the chain of an honest party at a later round, then there is an *associated* adversarial block of the same height that was adopted by an honest party.

We now state and prove our Unsuppressibility Lemma. Informally, the lemma says that if the number of blocks the adversary obtained in a set of consecutive rounds is z and the number of the uniquely successful blocks the honest parties obtained in the same set of rounds is y, then there exist $y-2z$ blocks that will always belong to the chain of every honest party. It follows that if the power of the adversary is bounded below $1/3$ of the total power, then with overwhelming probability there will be a nonzero number of such blocks.

An important note with respect to the Unsuppressibility Lemma is the following. Fix all the randomness the random oracle requires for a given execution. This determines the successful queries of every party and therefore determines the parameters y and z above. The observation is that even if these random coins are revealed to the adversary at the beginning of the execution, one can determine precisely which $y-2z$ blocks —and no matter the adversary's strategy— will always belong to the chain of every honest party.

Lemma 42 (Unsuppressibility)**.** *In a typical execution, every set of consecutive rounds U has a subset S of uniquely successful rounds, such that*

- $|S| \geq Y(U) - 2Z(U) - 2\lambda f(\frac{t}{n-t} \cdot \frac{1}{1-f} + \varepsilon)$ *and*
- *after the last round in S the blocks corresponding to S belong to the chain of every honest party.*

Proof. Let U' be the set of consecutive rounds that contains U and also the λ rounds that come before and after U. By Lemma 41, we may take S to contain all those uniquely successful rounds $r \in U$ such that for any set of consecutive rounds $S' \subseteq U'$ containing r, $Y(S') > Z(S')$. Note that, in a typical execution, no such S' may contain elements outside U'. Letting $y = Y(U)$ and $z = Z(U)$, we need to show $y - |S| \leq 2z + 2(1-\frac{2\delta}{3})\lambda f$.

Let us focus on the uniquely successful rounds not in S. Consider a collection $\mathcal{T}$ of sets of consecutive rounds with the following properties.

- For all $T \in \mathcal{T}$, $Y(T) \leq Z(T)$.
- For each $r \in U \setminus S$, there is a $T \in \mathcal{T}$ that contains r.
- $|\mathcal{T}|$ is minimum among all collections with the above properties.

We now observe that the minimality condition on $\mathcal{T}$ implies that no round r with $Z_r > 0$ belongs to more than two sets of $\mathcal{T}$. If that was the case, then there would be three sets T_1, T_2, T_3 in $\mathcal{T}$ with $T_1 \cap T_2 \cap T_3 \neq \emptyset$. But then, we could keep the two sets with the leftmost and rightmost endpoints, contradicting the minimality of $\mathcal{T}$. Furthermore, no round in $U' \setminus U$ belongs to more than one set of $\mathcal{T}$. Thus,

$$y - |S| = \sum_{i \in U \setminus S} Y_i \leq \sum_{T \in \mathcal{T}} Y(T) \leq \sum_{T \in \mathcal{T}} Z(T) \leq 2z + Z(U' \setminus U).$$

The third inequality holds because every round in which the adversary was successful is counted at most twice inside U and at most once outside U (by the discussion above the inequalities). Finally, using $|U' \setminus U| \leq 2\lambda$ and Lemma 39(c) we obtain the stated bound. □

The proof of this lemma is quite generous to the adversary on two accounts. First, it reveals to the adversary all coin flips in the beginning of the execution. Second, it gives the adversary two choices for each one of his blocks, and assumes that he will be able to choose among these as he sees fit. Nevertheless, we conjecture that the bound $y - 2z$ cannot be substantially increased in the case the property is rare.

We can now prove that an adversary with less than 1/3 of the total mining power cannot create a chain with more Q-blocks than an honest chain. Such a task would require the adversary to both suppress many Q-blocks from the honest chain *and* to obtain many of them for the adversarial chain.

Lemma 43 (Q-block Common-Prefix)**.** *Assume $t < (\frac{1}{3} - \delta)n$ with $\delta > 3\varepsilon + 3f$ and a Q-typical execution. Consider a round at which a chain C is adopted by an honest party and suppose there exist another chain C' such that $\mathsf{C}' \setminus (\mathsf{C}' \cap \mathsf{C})$ has at least $22\lambda\xi_Q f$ Q-blocks. Then, with overwhelming probability, C has more Q-blocks than C'.*

Proof. Assume an execution in which the assumptions of the lemma hold. Let r^* be the round on which the last honest block on $\mathsf{C}^* = \mathsf{C} \cap \mathsf{C}'$ was computed (if no such block exists let $r^* = 0$) and define the set of rounds $S = \{i : r^* < i \leq r\}$. We will study the execution during the rounds in S. To that end, let W' denote the set of adversarial queries on $\mathsf{C}' \setminus \mathsf{C}^*$ at some round at least λ greater from r^*. Denote by W the rest of the adversarial queries in S.

We first observe that no query in W' could have suppressed a Q-block on C. As in the proof of Lemma 41, in such a case there would exist a set of consecutive rounds $|S^*| \geq \lambda$ such that $Y(S^*) \leq Z(S^*)$. This contradicts the last item of Lemma 39.

From this observation and the Unsuppressibility Lemma, there are at least $Y(S) - 2Z(W) - 2\lambda f(\frac{t}{n-t} \cdot \frac{1}{1-f} + \varepsilon)$ blocks that the adversary cannot suppress. Each of these is a Q-block independently with probability ξ_Q. Under our assumptions, $2(\frac{t}{n-t} \cdot \frac{1}{1-f} + \varepsilon) < \frac{1-\delta}{1-\varepsilon}$. We conclude that, with overwhelming probability, there are at least

$$(1-\varepsilon)\xi_Q \cdot \big[Y(S) - 2Z(W)\big] - (1-\varepsilon)\lambda\xi_Q f$$

Q-blocks on $\mathsf{C} \setminus \mathsf{C}^*$.

On the other hand, the number of Q-blocks on $\mathsf{C}' \setminus \mathsf{C}^*$ is at most the Q-blocks from the W' queries plus the Q-blocks from the initial λ rounds. The latter can be shown to be at most $3\lambda\xi_Q f$. For the former, using Lemma 44 (with $F_j = 1$ when $j \in W'$ and $M_j = 1$ when it resulted in a Q-block) and Lemma 39, in a typical execution, are at most $(1+\varepsilon)\xi_Q Z(W')$. Thus, there at most

$$(1+\varepsilon)\xi_Q p|W'| + 3\lambda\xi_Q f.$$

Q-blocks on $\mathsf{C}' \setminus \mathsf{C}^*$. Since these are at least $22\lambda\xi_Q f$, it can be shown that the difference between the last two displayed expressions is at least $(1-\varepsilon)\xi_Q \cdot [Y(S) - 2Z(S)]$. This is positive in a typical execution in which the power of the adversary is bounded below $\frac{1}{3} - \delta$ the total power. □

In the above proof, we made use of the following lemma pertaining to the distribution of products of random variables. For completeness, we also include its proof here.

Lemma 44. *For each $j \in \mathbb{N}$, let F_j and M_j be Boolean random variables such that $\mathbb{E}[M_j] = \zeta$ and M_j is independent of F_i for $i \le j$ and independent of M_i for $i \ne j$. For any $\varepsilon \in (0,1)$,*

$$\Pr\left[\sum F_j M_j > (1+\varepsilon)\zeta \sum F_j \bigwedge \sum F_j M_j \ge k\right] \le e^{-\Omega(\varepsilon^2 k)}.$$

Proof. Since $\sum_{n \ge k} e^{-\Omega(\varepsilon^2 n)} = e^{-\Omega(\varepsilon^2 k)}$, by the union bound it suffices to show that

$$\Pr\left[(1+\varepsilon)\zeta \sum F_j < k \bigwedge \sum F_j M_j = k\right] \le e^{-\Omega(\varepsilon^2 k)}. \tag{4.4}$$

In the summations below, let α range over words in $\{0,1\}^*$ and β be any word in $\{0,1\}^\ell$ of weight k. For a fixed α, define $J_\alpha = \{j \in \mathbb{N} : F_j = 1\}$ and $B = (M_j)_{j \in J_\alpha}$. Also, for $j \in \mathbb{N}$, let E_j denote the event $\{(\forall i < j)(F_i = \alpha_i \text{ and } i \in J \Rightarrow M_i = \beta_i)\}$. Then,

$$\begin{aligned}
\Pr[B = \beta] &= \sum_\alpha \Pr[B = \beta, A = \alpha] \\
&= \sum_\alpha \prod_j \Pr[F_j = \alpha_j | E_j] \prod_{j \in J} \Pr[B_j = \beta_j | E_j, F_j = \alpha_j] \\
&= \sum_\alpha \prod_j \Pr[F_j = \alpha_j | E_j, B = \beta] \prod_{j \in J} \Pr[M_j = \beta_j] \\
&= \sum_\alpha \Pr[A = \alpha | B = \beta] \cdot \zeta^k (1-\zeta)^{\ell-k} \le \zeta^k (1-\zeta)^{\ell-k}.
\end{aligned}$$

Thus, letting β range over all words in $\{0,1\}^*$ of length less than $\frac{k}{(1+\varepsilon)\zeta}$ and weight k ending with 1, the left-hand side of (4.4) is equal to

$$\sum_\beta \Pr[B = \beta] \le \sum_{k \le \ell < \frac{k}{(1+\varepsilon)\zeta}} \binom{\ell-1}{k-1} \zeta^k (1-\zeta)^{\ell-k}.$$

That is, the probability is at most that of a random variable following a negative binomial distribution with parameters k (the number of successes) and ζ (the probability of success) is less than $\frac{k}{(1+\varepsilon)\zeta}$. The bound follows from standard Chernoff bounds. $\square$

4.6 Analysis

We are now ready to prove the construction of Algorithms 31 and 32 secure and succinct. For security, we denote Π the proof presented by the honest party and Π' the proof presented by the adversary (but these can be given to the verifier algorithm in any order). We can safely assume that these proofs were both generated at round r (the adversary could have generated the proof earlier, but not later, than the honest party). The honest proof Π was generated based on some honest underlying chain C using Algorithm 31. On the other hand, we have no guarantees about how the adversarial proof Π' was generated. It may be based on some underlying chain mined according to protocol, or not. In any case Π' *does* form a chain and its blocks must have been mined in order, as the verifier ensures this. However, there may not exist intermediate blocks covering the whole proof-of-work as desired.

Security mandates that the verifier chooses the honest proof, Π. However, it is possible that the verifier also chooses the adversarial proof, Π', without raising any issue, *as long as it extends the honest proof* at a fork point no longer than k from the tip. To see why this is fine, note that an adversary can already do this at the full blockchain: According to the Common Prefix property, she can fork at a block at most k blocks deep from the honest blockchain's end and have up to k blocks following the fork point. In this case, if it happens that the verifier has chosen Π', we require that $(\Pi' \cap \mathsf{C})[-1] \in \mathsf{C}[-k{:}]$. This means that the adversarial proof extends the honest chain at some fork point in $\mathsf{C}[-k{:}]$. But let us contemplate what this entails: It means that the portion $\Pi'\{(\Pi' \cap \mathsf{C})[-1]{:}\}$ is just a valid 0-level extension of the honest chain. As such, requiring $|\Pi'\{(\Pi' \cap \mathsf{C})[-1]{:}\}| \geq |\mathsf{C}\{(\Pi' \cap \mathsf{C})[-1]{:}\}|$ would produce a competitive adversarial chain that is *longer* than the honest chain and it would be perfectly acceptable to a full node (and by the common prefix property, this difference cannot be larger than k blocks long). We must clearly allow for this possibility — but it is not a problem, as this situation can occur in full node executions, too. This property also holds trivially in case the honest proof is chosen.

Theorem 45 (Security)**.** *Consider an arbitrary $\frac{1}{3}$-bounded PPT adversary $\mathcal{A}$ in a typical execution. Let Π be a proof generated by an honest party at round r using Algorithm 31 by passing his underlying chain C. Let Π' be an arbitrary proof generated by the adversary at round r. Let Π^* be the proof accepted by the verifier using Algorithm 32. Then $|\Pi^*\{(\Pi^* \cap \mathsf{C})[-1]{:}\}| \geq |\mathsf{C}\{(\Pi^* \cap \mathsf{C})[-1]{:}\}|$ with overwhelming probability.*

Proof. Let $\mathsf{C}' = \Pi'$. We need to show that, either Π will be the proof accepted by the verifier, or Π^* is a proof extending the honest chain that is longer at level 0, as mandated by the theorem statement.

Let us consider first the case that a μ of Algorithm 32 as above exists. When $\mu = 0$, the verifier determines the longer chain and always correctly accepts the corresponding proof. That is, the verifier will either choose $\Pi^* = \Pi$, or, in case $\Pi^* = \Pi'$, the verifier will choose the adversarial proof Π' which contains a χ' that extends the honest chain's χ at level 0 (up to k blocks long) with a longer alternative. This is the only case in which Π' can win. For the other cases, we will now argue that the adversary cannot win.

Let us now focus on the case $0 < \mu \leq \ell$. Note that, since $\mathcal{D}[\mu-1] \cap \mathcal{D}'[\mu-1] = \emptyset$ (by the minimality of μ), *both* superchains must have at least m blocks after their common block b. The Q-block Common-Prefix Lemma implies that Π is accepted.

Next, consider the case that no such μ exists. Clearly, $\ell \neq \ell'$ (otherwise $\mathcal{D}[\ell] \cap \mathcal{D}'[\ell']$ would contain the genesis block) and we need to argue that $\ell > \ell'$. Assume —towards a contradiction— that $\ell < \ell'$ and consider the statement of the Q-block Common-Prefix Lemma instantiated with blocks of level $\ell + 1$ as the Q-blocks. Together with $\ell < \ell'$, it implies that C' has fewer than m Q-blocks after the common block with C (since C has fewer Q-blocks than C' in total, it must also have fewer on its fork; and they must necessary share a common block, since both must begin with genesis). But then, both C and C' have fewer than m Q-blocks after their common block. Since $\mathcal{D}[\ell] \cap \mathcal{D}'[\ell] = \emptyset$ by assumption, this cannot be the case. □

Theorem 46 (Succinctness)**.** *In a typical execution with $t < (\frac{1}{3} - \delta)n$ with $3\epsilon + 3f < \delta < \frac{1}{3}$ and letting $m = \lambda$, an honest miner's state is in $O(m^2 \log(r))$ at round r.*

Proof. As $t < (\frac{1}{3} - \delta)n$ with $3\epsilon + 3f < \delta < \frac{1}{3}$, therefore $c = \mathbb{E}[Y] - 2\,\mathbb{E}[Z] - 2\lambda f(\frac{t}{n-t}\frac{1}{1-f} + \epsilon)$ will be a positive constant and for sets of consecutive rounds U with $|U| \geq \lambda$, we will have $Y(U) - 2Z(U) - 2\lambda f(\frac{t}{n-t}\frac{1}{1-f} + \epsilon) > (1 - \epsilon)c|U|$.

Consider a state Π generated by an honest prover and suppose for contradiction that $|\Pi| \in \omega(m \log(r))$, where r indicates the current round number. From the security of the scheme, this state will correspond to some underlying chain C such that Π is the compression of C. Consider the variables $(\mathcal{D}, \chi) = \mathsf{Dissolve}_{m,k}(\mathsf{C})$. As $|\chi| = k$ is constant, therefore $|\bigcup_{d \in \mathcal{D}} d| \in \omega(m \log(r))$. Let $\ell = |\mathcal{D}|$. It holds that $\ell \in O(\log(|\mathsf{C}|))$. Consequently, $\sum_{d \in \mathcal{D}} |d| \in \omega(m \log(r))$. Therefore there must exist some μ such that $|\mathcal{D}[\mu]| \in \Omega(\lambda^2)$. Consider the maximum such μ.

We distinguish two cases.

Case 1: $\mu = \ell$. Then consider $\mathcal{D}[\ell]$. Let u_0 denote the round during which $\mathcal{D}[\ell][0]$ was generated and u_1 denote the round during which $\mathcal{D}[\ell][-1]$ was generated and consider the set U of consecutive rounds from u_0 to u_1. As $\mathcal{D}[\ell]$ forms a chain, we have that $|U| \geq |\mathcal{D}[\ell]| > \lambda$. Applying the Unsuppressibility Lemma, we obtain that at least $|S| \geq c|U| = c|\mathcal{D}[\ell]| \in \Omega(\lambda)$ rounds of U must have been uniquely successful and belong to the chain of every honest party. Therefore $|\mathcal{D}[\ell]{\uparrow^{\ell+1}}| \geq (1 - \epsilon)\frac{|S|}{2}$. By the definition of ℓ this is impossible.

Case 2: $0 \leq \mu < \ell$. By maximality of μ, we have $|\mathcal{D}[\mu + 1]| \in O(\lambda)$, but $|\mathcal{D}[\mu]| \in \Omega(\lambda^2)$. By the definition of $\mathcal{D}[\mu] = \mathsf{C}[{:}{-k}]{\uparrow^{\mu}} [-2m{:}] \cup \mathsf{C}[{:}{-k}]{\uparrow^{\mu}} \{\mathsf{C}[{:}{-k}]{\uparrow^{\mu+1}} [-m]{:}\}$, clearly $|\mathsf{C}[{:}-k]{\uparrow^{\mu}} [-2m{:}]| = 2m$ so necessarily $\mathsf{C}[{:}-k]{\uparrow^{\mu}} \{\mathcal{D}[\mu+1][-m]{:}\} \in \Omega(\lambda^2)$. Therefore there exist blocks A and B in $\mathcal{D}[\mu+1]$ and $\mathcal{D}[\mu]$ such that $|\mathcal{D}[\mu+1]\{A{:}Z\}| = 1$, but $|\mathcal{D}[\mu]\{A{:}Z\}| \in \omega(\lambda)$. Similarly to case 1, consider the rounds u_0 and u_1 during which blocks A and Z were generated respectively and the set of consecutive rounds U from u_0 to u_1 with $|U| \in \omega(\lambda)$. Using the Unsuppressibility Lemma, there must exist a set of uniquely successful rounds $|S| \geq c|U|$ whose blocks have been adopted by all honest parties and of which at least $(1 - \epsilon)\frac{|S|}{2} \geq 0$ will be of level $\mu + 1$. Therefore there must exist a block between A and Z in $\mathcal{D}[\mu + 1]$.

Both cases are contradictions. □

The previous theorem allows us to make miners reject incoming state that is too large (more than polylogarithmic) without processing them fully.

We note here that our analysis critically relies on the honest majority assumption holding *throughout* the execution. The reason why our verifiers can maintain a valid chain is that, once they receive a chain C which is the longest, they inductively know that $\mathsf{C}[-k]$ must contain valid application data snapshot. Then, since they have all the last k blocks, they can validate the transactions δ on the snapshot obtained before further mining on top of them.

Comparing this result to Theorem 31 of Chapter 3, we observe that here, for the first time, succinctness is proven against all adversaries, and not just in the optimistic setting. The whole machinery of Q-blocks and suppression attacks was required before we could complete such a proof. Using an identical approach, the succinctness of charity NIPoPoWs (of Chapter 3) can also be proven succinct in all settings, against 1/3 adversaries.

4.7 Blinded mining

The protocol presented in the previous section is secure against $\frac{1}{3}$-bounded adversaries. We note that the reason why an adversary with more mining power is able to attack the scheme is because they are able to see which blocks are Q-blocks so that they can selectively supress them. This motivates us to design a protocol in which the Q-blockness of a block is not known to the adversary until the block has been stabilized by being buried under k subsequent blocks and can no longer be supressed. Therefore, we wish to keep the superblock status secret for a duration of k blocks.

We modify the honest protocol to work as follows. The pow function works as usual and accepts as input the usual parameters x (the transaction data) and s (the reference to the previous block) and finds a ctr such that $H(ctr||x||s) \leq T$. The honest party broadcasts x and s, but keeps ctr private. It then generates a commitment $s' = \text{Com}(H(ctr||x||s))$ which commits to the block hash $H(ctr||x||s)$. Next, it generates a zero knowledge proof π proving the following statement: I know an ctr such that s' commits to $H(ctr||x||s)$ and $H(ctr||x||s) \leq T$. The commitment and the zero-knowledge proof are then broadcast by the party.

Mining is performed by setting the first s to be a predefined string (genesis) as usual. Each next miner then builds on top of s', the commitment to the previous block, instead of building on top of the usual $H(ctr||x||s)$. As the new block points to the commitment, this is sufficient commitment to ensure proof-of-work security.

The value ctr is kept secret by the generating party until the newly generated block is buried under subsequent $2k$ blocks according to the view of the party which originally generated the block. At that point, the party reveals ctr through a revealing transaction $\text{tx}_{\text{reveal}}$ which includes the ctr value as well as the commitment opening for s' showing that the particular ctr value is the one that was committed to. This transaction is included by an honest miner as long as the chain does not contain a previous commitment opening for that particular block.

To incentivize this honest behavior, we mandate that the miners are paid only *after* opening their commitment. However, if a commitment opening is done prior to k blocks after the commitment is placed in a block, the rewards are slashed and the miner is not paid. For more details, see Section 4.10.

4.7.1 The Trapdoor Random Oracle

The above protocol in which the ctr of each block remains secret can be abstracted by the concept of a *trapdoor random oracle* in which the party that mines a new block makes the first query to the random oracle using a *secret witness sw* which is associated with a unique *public witness pw*. The random oracle returns a response y as usual, but also associates a secret ξ with this response, following a distribution Ξ which is predetermined. The secret ξ is only recoverable and verifiable using the secret witness sw. In our protocol, an honest miner reveals sw at a later time in order to prove to other parties whether their previously mined block is a Q-block. The Q-blockness of a block is then determined by any predicate applied on ξ. The adversary can, of course, keep the secret witness sw withheld.

The Trapdoor Random Oracle is illustrated in Figure 4.2 and works as follows. A query is made to the Random Oracle by invoking the query-p or query-s methods, passing an x (the query string) as well as a public witness pw (for query-p) or a secret witness sw (for query-s). The public query returns y, while the private query returns

Algorithm 34 The Trapdoor Random Oracle functionality RO_Ξ parameterized by security parameters κ, λ and a distribution ensemble $(\Xi)_\kappa$.

```
$T_f \leftarrow \emptyset$
$T_p \leftarrow \emptyset$
function query-f($sw$)
    if $\nexists pw : (pw, sw) \in T_f$ then
        $pw \xleftarrow{\$} \{0,1\}^\lambda$
        $T_f \leftarrow T_f \cup \{(pw, sw)\}$
    end if
    $pw \leftarrow$ the $pw$ such that $(pw, sw) \in T_f$
    return $pw$
end function
function query-p($x$)
    if $\nexists(y, \xi) : (x, y, \xi) \in T_p$ then
        $y \xleftarrow{\$} \{0,1\}^\kappa$
        $\xi \leftarrow \Xi$
        $T_p \leftarrow T_p \cup \{(x, y, \xi)\}$
    end if
    $(y, \xi) \leftarrow$ the $(y, \xi)$ such that $(x, y, \xi) \in T_p$
    return $y$
end function
function query-s($x, sw$)
    parse $x$ into $x'pw$
    if parsing failed then
        return $\perp$
    end if
    if query-f($sw$) $\neq pw$ then
        $\triangleright$ Ensure sw is the correct trapdoor
        return $\perp$
    end if
    query-p(x)
    $(y, \xi) \leftarrow$ the $(y, \xi)$ such that $(x, y, \xi) \in T_p$
    return $\xi$
end function
```

The *trapdoor* random oracle functionality RO_Ξ is parameterized by the security parameters κ, λ and a distribution ensemble $(\Xi)_\kappa$. Upon initialization, it sets T_p and T_f to $\emptyset$. It supplies the following methods:

- query-f(sw): If $(sw, pw) \in T_f$ for some value pw, return pw. Otherwise choose a $pw \leftarrow \{0,1\}^\lambda$ and set $T_f = T_f \cup \{(sw, pw)\}$, then return pw.
- query-p(x): If $(x, y, \xi) \in T_p$ for some values y, ξ, return y. Otherwise choose $y \leftarrow \{0,1\}^\kappa$ and $\xi \leftarrow \Xi_\kappa$ and set $T_p = T_p \cup \{(x, y, \xi)\}$, then return y.
- query-s(x, sw): Parse $x = x'||pw$ for some x', pw. If the parsing fails, return $\perp$. Otherwise, if query-f(pw) $\neq sw$, then the trapdoor is incorrect, so return $\perp$. Otherwise, if $(x, y, \xi) \in T_s$ for some values y, ξ, return ξ. Otherwise, choose $y \leftarrow \{0,1\}^\kappa$ and $\xi \leftarrow \Xi_\kappa$ and set $T_s = T_s \cup \{(x, y, \xi)\}$, then return ξ.

Figure 4.2: The *trapdoor* random oracle functionality RO_Ξ.

$$\begin{tikzcd}
\text{x, sw} \arrow[r] \arrow[d] & \xi \\
\text{x, pw} \arrow[r] & \text{y}
\end{tikzcd}$$

Figure 4.3: The RO_Ξ functionality and the directions of its available queries.

ξ. If the invocation is not new, the method returns the cached value. Otherwise, the extended Random Oracle functionality generates a y uniformly at random as usual for the public query. In addition, it generates a value ξ by sampling from the distribution Ξ for the private query. It also provides a functionality query-f which allows retrieving pw given sw. The query directions offered by the functionality are depicted in Figure 4.3.

Next we show how the trapdoor random oracle can be implemented for a distribution ensemble equal to $\{0,1\}^\kappa$. Then we have the following theorem:

Theorem 47. *Algorithm 35 implements the functionality* RO_Ξ *in Figure 4.3 assuming random oracles G, H.*

Given the above functionality, block generation can be performed as usual, but invoking the Trapdoor Random Oracle to perform blinded mining. In this case, the input to the Trapdoor Random Oracle includes all of the usual data ($ctr||x||s$) as well as a secret witness sw which is revealed at a later time.
Remark. In practice, our scheme can be implemented using the following technique to avoid commitments. Query $H(H(ctr)||x||s) \leq T$ to check if the query gives a

Algorithm 35 The Trapdoor Random Oracle Implementation parameterized by ROs $G(\cdot), H(\cdot)$ and a distribution ensemble $(\Xi)_\kappa = \{0,1\}^\kappa$.

```
1: $T_f \leftarrow \emptyset$
2: $T_p \leftarrow \emptyset$
3: function query-f($sw$)
4:     $pw \leftarrow G(sw)$
5:     return $pw$
6: end function
7: function query-p($x$)
8:     parse $x$ into $x'pw$
9:     $y \leftarrow H(pw||x')$
10:    return $y$
11: end function
12: function query-s($x, sw$)
13:    parse $x$ into $x'pw$
14:    if query-f($sw$) $\neq pw$ then
15:        return $\perp$
16:    end if
17:    $\xi \leftarrow H(H(pw||x')||sw)$
18:    return $\xi$
19: end function
```

valid block. If so, then reveal $H(ctr)||x||s$. Anyone on the network can check the block's validity. Whether the block is a Q-block can be determined by checking whether, for some $\mu \in \mathbb{N}$ the following inequality holds:

$$H(ctr||H(H(ctr)||x||s)) \leq 2^{-\mu}$$

This can only be checked once ctr is revealed (provided ctr contains sufficient entropy), which corresponds to the Trapdoor Random Oracle secret. The particular order of evaluation for H ensures that the query for block validity $H(H(ctr)||x||s)$ has to be submitted by the adversary before they are able to learn the Q-status of the block through the query $H(ctr||H(H(ctr)||x||s))$.

4.8 Logspace mining against 1/2

Based on the trapdoor oracle implementation of the previous section, we modify the mining protocol so that the proof-of-work equation becomes

$$H(G(ctr)||mtr||interlink) \leq T$$

where G is a pre-image resistant hash function. The difference with the plain protocol is that the nonce ctr is hashed prior to being put into the proof-of-work equation. The property of a block being a μ-superblock is then defined by the equation

$$H(H(G(ctr)||mtr||interlink)||ctr) \leq \frac{2^\kappa}{2^\mu}$$

Note that, in order to determine the Q-block status of a block, the inner evaluation of H must have already been completed.

Mining then works as follows. The honest miners try to solve the new proof-of-work equation. If they succeed, they broadcast their solution as usual. They get paid their rewards into a coinbase transaction. However, that coinbase transaction remains locked for now. The nodes receiving a blockchain check that the new proof-of-work equation has been solved correctly. After k blocks (where k is the Common Prefix parameter) have elapsed and the adopted blockchain is C, the miner who solved the proof-of-work of block $\mathsf{C}[-k]$ discloses the value of *ctr* with a reference to the respective block. This information is broadcast and included in a transaction which is confirmed in blocks following C. This condition "unfreezes" the coinbase payout of the miner.

4.8.1 Security

We show how to strengthen the Common-Prefix Lemma (Lemma 43), taking advantage of the fact that the adversary decides to suppress a block without knowing if it satisfies the block property or not.

Lemma 48 (Q-block Common-Prefix Lemma)**.** *Assume $t < (\frac{1}{2} - \delta)n$ with $\delta > 3\varepsilon + 3f$ and a Q-typical execution. Consider a round at which a chain C is adopted by an honest party and suppose there exist another chain C' such that $\mathsf{C}' \setminus (\mathsf{C}' \cap \mathsf{C})$ has at least $12\lambda\xi_Q f/\varepsilon$ Q-blocks. Then, with high probability, C has more Q-blocks than C'.*

Proof. Define r^*, S, W, W' as in the proof of Lemma 43. As in that proof, the number of Q-blocks on $\mathsf{C}' \setminus \mathsf{C}^*$ is at most

$$(1+\varepsilon)\xi_Q p|W'| + 3\lambda\xi_Q f.$$

To upper bound the number of Q-blocks the adversary can suppress we are going to apply Lemma 44. Consider every block computed in S that was suppressed by the adversary. Each such block has an associated adversarial block (see Lemma 41 and the paragraph following its proof). Let J contain each honest query j in S that attempted to create a suppressed block. Let $F_j = 1$ for each $j \in J$, unless the suppressed block was already more than k blocks deep at the time its associated block was created. Let $M_j = 1$ if the suppressed block was a valid Q-block. By Lemma 44, the adversary suppressed at most $(1+\varepsilon)\xi_Q p|W|$. Thus the number of Q-blocks on $\mathsf{C} \setminus \mathsf{C}^*$ is at least

$$Y_Q(S) - (1+\varepsilon)\xi_Q p|W|.$$

Subtracting from this the upper bound above we obtain

$$\begin{aligned}
&Y_Q(S) - (1+\varepsilon)\xi_Q pqt|S| - 3\lambda\xi_Q f \\
&\qquad > Y_Q(S) - \frac{(1+\varepsilon)(1-\delta)}{1-f}\xi_Q f|S| - 3\lambda\xi_Q f \\
&\qquad > Y_Q(S) - \frac{(1+\varepsilon)(1-\delta)}{1-f}\xi_Q f|S| - \varepsilon\xi_Q f|S| \\
&\qquad\qquad > Y_Q(S) - Z_Q(S) > 0.
\end{aligned}$$

The first and the last inequality use Lemma 39. For the middle inequality, observe that if in a typical execution $|S| \leq 3\lambda/\varepsilon$, then less than $12\lambda\xi_Q f/\varepsilon$ Q-blocks have been computed in total. □

4.9 Computational Ledgers

We note here that a full analysis of the logspace mining protocol would require a redefinition of the concepts of *persistence* and *liveness* to account for the fact that the blockchain application state at which the network converges is a valid historical possibility. We sketch these definitions for reference here.

The blockchain protocol is parametrized by a *blockchain application* tuple $(\mathcal{L}, \mathcal{S}, S_0, \delta)$ which consists of the following:

1. $\mathcal{L}$: A (potentially infinite) *transaction language* containing all valid transactions.
2. $\mathcal{S}$: A (potentially infinite) *set of valid states* that the system can be in.
3. $S_0 \in \mathcal{S}$: A *genesis state* that the system begins with.
4. $\delta : \mathcal{S} \times \mathcal{L} \longrightarrow \mathcal{S} \cup \{\perp\}$: An efficiently computable *transition function* that takes a transaction tx, a previous state $S \in \mathcal{S}$ and returns a next state $S' \in \mathcal{S}$, or $\perp$ if the transaction tx cannot be applied on top of state S.

The transition function δ can be extended to apply multiple transactions in a transition function $\delta^* : \mathcal{S} \times \mathcal{L}^* \longrightarrow \mathcal{S} \cup \{\perp\}$ which is defined recursively: $\delta^*(S, \epsilon) = S$ and $\delta^*(S, \mathsf{tx}\,\overline{\mathsf{tx}})) = \begin{cases} \perp, \text{ if } S = \perp \\ \delta^*(S, \overline{\mathsf{tx}}), \text{ otherwise} \end{cases}$.

Each block B in a blockchain contains application data which consists of the previous state $B.S \in \mathcal{S}$ and a sequence of transactions $B.\overline{\mathsf{tx}}$ with $\forall \mathsf{tx} \in B.\overline{\mathsf{tx}} \in \mathcal{L}$ that this block confirms. It must hold that the transitions described by a block are valid i.e., that $\delta^*(B.S, B.\overline{\mathsf{tx}}) \neq \perp$ and that each next block's previous state is the previous block's next state, i.e., $\forall i > 0 : \delta^*(\mathsf{C}[i-1].S, \mathsf{C}[i-1].\overline{\mathsf{tx}}) = \mathsf{C}[i].S$. The genesis block $\mathcal{G}$ contains a commitment to the genesis state $\mathcal{G}.S = S_0$.

Whenever a full node receives a blockchain C they check its validity by ensuring that the above properties hold. Then, the reported *stable ledger state* $\mathsf{L}_p[r]$ for each honest party p at round r is the state S that is committed to the tip of the stable portion of the blockchain.

To define computational persistence, we set up the game illustrated in Algorithm 36. Let $r_1 \leq r_2 \in \mathbb{N}$ be rounds and p_1, p_2 be honest parties. Consider the ledger states $\mathsf{L}_{p_1}[r_1]$ and $\mathsf{L}_{p_2}[r_2]$ adopted by p_1 and p_2 at rounds $r_1 \leq r_2$. In this game, an adversary $(\mathcal{A}_1, \mathcal{A}_2)$ participates in an execution in which he attempts to cause p_1 and p_2 to arrive at ledger states $\mathsf{L}_{p_1}, \mathsf{L}_{p_2}$ at rounds r_1, r_2 such that the ledger of p_2 could not have possibly been the extension of the ledger of p_1. We capture the success of the adversary through a simulator who remains unable to produce a transaction sequence $\overline{tx}$ which conciliates ledgers $\mathsf{L}_{p_1}[r_1]$ and $\mathsf{L}_{p_2}[r_2]$, i.e., $\delta^*(\mathsf{L}_{p_1}[r_1], \overline{tx}) = \mathsf{L}_{p_2}[r_2]$. We remark here that the mere existence of $\overline{tx}$ is insufficient; it must be efficiently computable by a simulator. An example in which a transaction sequence exists but is not efficiently computable is the case where the adversary causes the honest parties to transition from a state S_1 in which an honest

Algorithm 36 The challenger for the ledger persistence game.

```
1:  function persistence-game$_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{Z},\mathcal{A}^*,\Pi}(\kappa)$
2:      $v \leftarrow \text{view}^{t,n}_{\Pi,\mathcal{A}_1,\mathcal{Z}}$
3:      $r_1, r_2, p_1, p_2 \leftarrow \mathcal{A}_2(v)$
4:      if $r_2 < r_1 \vee (p_1, p_2$ are not honest parties in $v)$ then
5:          return $\mathsf{false}$
6:      end if
7:      $\overline{tx} \leftarrow \mathcal{S}(v, r_1, r_2, p_1, p_2)$
8:      $\mathsf{L}_{p_1}, \mathsf{L}_{p_2} \leftarrow$ ledgers of $p_1, p_2$ in $v$
9:      if $r_1 < r_2$ then
10:         return $(\delta^*(\mathsf{L}_{p_1}[r_1], \overline{\mathsf{tx}}) \neq \mathsf{L}_{p_2}[r_2])$
11:     else
12:         return $(\delta^*(\mathsf{L}_{p_1}[r_1], \overline{\mathsf{tx}}) \neq \mathsf{L}_{p_2}[r_2]) \wedge (\delta^*(\mathsf{L}_{p_2}[r_2], \overline{\mathsf{tx}}) \neq \mathsf{L}_{p_1}[r_1])$
13:     end if
14: end function
```

Algorithm 37 The challenger for the ledger liveness game.

```
1:  function liveness-game$_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{Z},\mathcal{A}^*,\Pi,u}(\kappa)$
2:      $v \leftarrow \text{view}^{t,n}_{\Pi,\mathcal{A}_1,\mathcal{Z}}$
3:      $r, p \leftarrow \mathcal{A}_2(v)$
4:      $\overline{tx} \leftarrow$ transactions issued continuously in $v$ for $u$ rounds prior to $r$
5:      if $p$ are is not honest in $v$ then
6:          return $\mathsf{false}$
7:      end if
8:      $\overline{tx}' \leftarrow \mathcal{A}^*(v, r, p)$
9:      $\mathsf{L}_p \leftarrow$ ledger of $p$ in $v$
10:     return $(\overline{tx} \not\subseteq \overline{tx}' \vee \delta^*(S_0, \overline{\mathsf{tx}}') \neq \mathsf{L}_p[r])$
11: end function
```

party holds certain coins secured by a public signature scheme, to a state S_2 in which the coins have been transferred to the adversary's control. In this case, the simulator will remain unable to produce the signature needed to relinquish control of the honest party's coins, and the adversary will be deemed successful.

Definition 62 (Computational persistence)**.** *A protocol* Π *has* computational persistence *if there is a negligible function* negl *such that for all probabilistic polynomial-time adversaries* $(\mathcal{A}_1, \mathcal{A}_2)$ *and all environments* $\mathcal{Z}$ *there exists a probabilistic polynomial-time simulator* $\mathcal{A}^*$ *such that*

$$\Pr[\text{persistence-game}_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{Z},\mathcal{A}^*,\Pi}(\kappa)] \leq \mathsf{negl}\,.$$

For the case of *computational liveness*, we introduce the similar game illustrated in Algorithm 37. In this game, parameterized by the *liveness parameter* u, the environment issues a sequence of transactions $\overline{tx}$ each of which is given as input to all honest parties continuously for u rounds prior to some round r. The adversary attempts to cause an honest party p to arrive at a ledger state $\mathsf{L}_p[r]$ at round r on which one or more of the transactions in $\overline{tx}$ have not been taken into account. Again, we express the adversary's success with the inability of a probabilistic polynomial-time simulator to produce a transaction sequence $\overline{tx}'$ with $\overline{tx}' \subseteq \overline{tx}$ which transitions the genesis state S_0 to the adopted ledger state $\mathsf{L}_p[r]$ of party p.

Definition 63 (Computational liveness)**.** *A protocol* Π *has* computational liveness *if there is a negligible function* negl *such that for all probabilistic polynomial-time adversaries* $(\mathcal{A}_1, \mathcal{A}_2)$ *and all environments* $\mathcal{Z}$ *there exists a probabilistic polynomial-time simulator* $\mathcal{A}^*$ *such that*

$$\Pr[\text{liveness-game}_{\mathcal{A}_1,\mathcal{A}_2,\mathcal{Z},\mathcal{A}^*,\Pi}(\kappa)] \leq \mathsf{negl}\,.$$

4.10 Deploying with a Soft Fork

It should be clear that the protocols proposed in the previous sections can be deployed in new blockchains from Genesis and can also be deployed in existing blockchains using a hard fork. It is interesting to consider whether it is also possible to deploy these protocols using a soft fork. We explore this question for the concrete case of Ethereum; other cryptocurrencies can be soft forked in a similar manner.

The protocol of Section 4.4 against 1/3 adversaries can easily be deployed using a soft fork. In order to do that, the interlink set described in Chapter 3 needs to be included in every block. To achieve backwards compatibility, it can be placed into an *interlink Merkle tree* whose root is placed in the `extraData` field. Whenever a proof is constructed, the prover must prove to the verifier that the proof forms a chain. Hence, along with each block in the proof, the respective pointer to the previous block in the proof must be presented. Therefore the prover accompanies each block with a Merkle Tree proof-of-inclusion which proves that the pointer is included within the interlink Merkle tree.

The protocol of Section 4.8 is more challenging to deploy. Again, we include the witness-augmented interlink vector into a Merkle tree whose root is stored in the `extraData` field. Recall that the current block header format of Ethereum is $H(ctr||x||previd)$, where ctr denotes the mining nonce, x indicates the application data, and $previd$ denotes the hash of the parent block. We require upgraded miners to mine by searching for a ctr' such that $H(B) \leq T$, where $B = H(ctr')||x||previd$. The blinded block value is then defined as $\xi = H(ctr'||H(H(ctr')||x||previd))$ and the block level is defined as the maximum μ such that $\xi \leq \frac{T}{2^\mu}$. In this scheme, the value $H(ctr')$ looks like the value ctr to unupgraded miners. Note that at this point, it is indistinguishable for the parties that did not mine the block in question whether it was mined using the old (ctr) or the new protocol ($H(ctr')$) prior to the value ctr' being disclosed, as, in the old protocol, ctr is chosen uniformly at random, while in the new protocol, the value $H(ctr')$ is determined by a Random Oracle. Consider a block B mined by an honest miner p. The value ctr' is revealed after the block B is burried under $2k + 2\ell$ blocks in the view of p. The revelation of ctr' takes place by having p create a transaction tx_{Reveal} which contains ctr' as well as the index of B in p's chain. As B is stable, this position will be the same for all other honest parties as well.

While the above scheme works in our cryptographic treatment, it is worth considering the incentives of such a scheme. In particular, observe that, without further modification, miners are not incentivized to mine using the new protocol and can continue mining with the unupgraded protocol, while never revealing any preimage ctr'. Hence, we want to incentivize miners to reveal the value ctr'. For this purpose, we can design the protocol so that the mining rewards are only paid out conditioned on the fact that ctr' is properly revealed.

This can be done with a soft fork as follows: The block beneficiary field is modified to pay out to a smart contract instead of a regular beneficiary address. The smart contract for this purpose is illustrated in Algorithm 38. It is deployed once into a known address and upgraded miners require that the beneficiary of every block is the particular smart contract.

The smart contract is parameterized by two constant hard-coded parameters k and ℓ, the common prefix and liveness parameters respectively. The parameter k ensures that a block buried under k blocks is considered stable and the parameter ℓ ensures that after ℓ blocks are broadcast during which an honest transaction appears in the mempool, it will have the chance to be included in the blockchain.

The smart contract is used by an honest miner as follows. The miner places one special transaction tx_{Claim} at the beginning of his newly mined block. That transaction calls the ClaimBlock function of the smart contract and claims the block to the miner's address before anyone else is able to do so. As the beneficiary address is taken up by the smart contract's address, this allows the miner to claim the block with his public key. The miner then waits for $2k + 2\ell$ blocks to pass and subsequently calls the OnTimeReveal function in the tx_{Reveal} transaction (and note that if anyone else happens to call this function, the rightful miner is still paid out). He passes the block number to be claimed, idx, as well as the nonce preimage ctr' that was used when he mined the block in question and the surrounding data in the block header α and β. The smart contract verifies that the nonce $H(ctr')$ was indeed used by comparing against the block hash, marks the block reward as paid, and pays the miner. Due to liveness guarantees, the transaction in which the miner calls OnTimeReveal will be included from $3k + 2\ell$ blocks until $3k + 3\ell$ blocks after the block in question was mined and hence will fall within the $\mathsf{desiredPeriod}$ mandated by the smart contract. In case the miner reveals late, the reward is lost.

It is undesirable that the miner reveals ctr' to another party before k blocks have passed. In case that happens, the party which has received the revelation can slash the miner's rewards. This is performed as follows. The party who has early knowledge of ctr' calls the function EarlyCommit within k blocks of the block in question, committing to a claim that they will perform an early reveal of the preimage. The commitment includes the address of the claimant as well as the secret preimage, so that contesting claimants cannot copy the claim and enter into a transaction race. This transaction will be included after at most ℓ blocks have passed. After it is included, the claimant waits for k blocks for confirmation to ensure no chain reorg will cause his commitment to be reversed. Hence, $2k + \ell$ blocks after the mined block in question, the claimant calls the function EarlyReveal in which an early revelation of the preimage is verified by the smart contract by ensuring the claim matches the given commitment. In that case, the claimant is paid a small percentage of the block rewards (in our example we use 10%), while the rest of the rewards are slashed and cannot be later claimed by the miner. Note that the full amount cannot be paid to the claimant, as we wish to discourage the miner from being the claimant himself. The transaction making the claim will take at most ℓ blocks to be included, and hence the total period for early claiming has

Algorithm 38 The BlindedMining smart contract used as a beneficiary in mining for soft fork deployment on Ethereum

```
contract BlindedMining$_{k,\ell}$
    uint earlyPeriod $\leftarrow 2k + 2\ell$
    uint desiredPeriod $\leftarrow$ earlyPeriod $+ k + \ell$
    float slashingFraction $\leftarrow 0.1$
    mapping(uint $\rightarrow$ addr) blockClaimed
    mapping(uint $\rightarrow$ bool) blockRevealed
    mapping(uint $\rightarrow$ mapping(addr $\rightarrow$ string)) commitments
    mapping(uint $\rightarrow$ uint) blockValue
    payable function $\oplus()$
        blockValue[block.number] $\leftarrow$ msg.value
    end function
    function ClaimBlock
        require(blockClaimed[block.number] = address(0))
        blockClaimed[block.number] $\leftarrow$ msg.sender
        blockNonce[block.number] $\leftarrow$ ctr
    end function
    function IsEarly(idx)
        return block.number $<$ idx $+$ earlyPeriod
    end function
    function IsLate(idx)
        return block.number $>$ idx $+$ desiredPeriod
    end function
    function IsOnTime(idx)
        return $\neg$IsEarly(idx) $\wedge$ $\neg$IsLate(idx)
    end function
    function EarlyCommit(idx, commitment)
        commitments[idx][msg.sender] $\leftarrow$ commitment
    end function
    function EarlyReveal(idx, ctr', $\alpha, \beta$)
        require(BlockHash(idx) $= H(\alpha || H($ctr'$) || \beta) \wedge$ IsEarly(idx))
        require(commitments[idx][msg.sender] $= H($ctr'||salt||msg.sender$)$)
        $v \leftarrow$ slashingFraction $*$ blockValue[idx]
        blockRevealed[idx] $\leftarrow$ true
        msg.sender.transfer(v)
    end function
    function OnTimeReveal(idx, ctr', $\alpha, \beta$)
        require(BlockHash(idx) $= H(\alpha || H($ctr'$) || \beta)$)
        require(IsOnTime($idx$) $\wedge \neg blockRevealed[idx]$)
        blockRevealed[idx] $\leftarrow$ true
        blockClaimed[idx].transfer(blockValue[idx])
    end function
end contract
```

a duration of $2k + 2\ell$ blocks. The reason we allow on-time revealing only after this period has passed is to allow for claimants that have learned the preimage during the first k blocks after the mined block to have a fair chance of committing and revealing it during the early period. The honest miner will make the claim only after $3k + 2\ell$ blocks have passed, so as to avoid the potential for chain reorgs to allow for subsequent early revelation (once the blockchain has $3k + 2\ell$ blocks, no new blockchain can be adopted which has different transactions prior to the $2k + 2\ell$ point).

In summary, this scheme disincentivizes miners to reveal prior to seeing that k blocks have buried their newly mined block (in which case they are guaranteed to be slashed) and incentivizes them to reveal after $3k + 2\ell$ blocks have passed (in which case they are guaranteed not to be slashed), but before $3k + 3\ell$ blocks have passed (so that they still receive their reward). This is sufficient for the construction of our blinded mining scheme.

We note here that smart contracts like the one described can be used as soft forks for any desirable reward distribution change in Ethereum.

4.11 Discussion

We have presented a scheme in which full miners are replaced with logarithmic-space miners. Our new mining protocol allows miners to only keep storage growing logarithmically in time. Furthermore, the data communicated to newly bootstrapped nodes is also logarithmic. We focused on optimizing the *consensus data* portion of blockchains (i.e., block headers) without concern for the *application data* portion. Our techniques can be composed with application data optimization techniques.

We have proven our scheme succinct and secure against all 1/3 adversaries. Our treatment requires *uninterrupted* honest computational majority throughout the execution, is in the *static* difficulty model, and works only for *proof-of-work* blockchains. Let us discuss these aspects of our construction.

Using our new mathematical framework, the *charity* construction of Chapter 3 (with *certificates of badness* removed) can also be proven secure against 1/3 adversaries. However, now that we have the tooling of unsupressibility available to us, and we can prove the *distill* construction secure and succinct, we have the additional benefit of the simplicity of comparison at a uniform level. However, as we will see in Chapter 5, this *distill* construction will not work in the variable difficulty setting, and we will have to resort back to the *charity* construction explored previously.

Temporary dishonest majority. One important difference between our scheme and the existing blockchain protocols is that traditional full nodes are able to verify the *whole* state evolution of the system from genesis. This allows them to recover in case of temporary dishonest majority [7, 14], while our system cannot do so. Let us consider what could happen in case an adversary temporarily has the upper hand in a blockchain where everybody is mining using our protocol. Let C denote the chain of the honest parties that has converged. The adversary begins mining on top of the honest tip. She eventually produces $k + 1$ new blocks on top of $\mathsf{C}[-1]$, generating an adversarial chain C^*, prior to the honest parties advancing by $k + 1$ blocks — a Common Prefix violation. In the block $\mathsf{C}^*[-k - 1]$, the adversary places an invalid snapshot; say, a snapshot in which she owns a lot of money. The rest of the blocks in $\mathsf{C}^*[-k{:}]$ are filled with valid transactions. This adversary can then compress this consensus state into a convincing proof, as state transitions buried $k + 1$ blocks beyond the tip are never checked. As soon as the honest parties transition to this adversarial chain, the attack concludes, and no more adversarial supremacy is required. It is critical to understand what assumptions our protocol mandates: An *uninterrupted* honest majority throughout the execution. It remains an open question whether it is possible to construct logarithmic space mining protocols that can withstand temporary adversarial supremacy.

Variable difficulty. We have built and analyzed our logarithmic mining protocol in the *constant* difficulty setting, i.e., requiring that the target T is a constant. We strongly suspect, but have not provided proof, that similar protocols to ours work in the variable difficulty setting. One important change in the protocol that is required before it can be adapted to variable difficulty settings is that the χ portion of the proofs cannot be a constant number of blocks long. Instead, it must be a suffix which corresponds to *sufficient work having been performed*, the difficulty of which must correspond to the current target. Simply pruning k blocks long is insufficient. As such, the verifier must first gauge the difficulty of the network prior to taking conclusive decisions. We explore variable difficulty constructions in Chapter 5.

Comparison to other NIPoPoWs. The protocol explored in this chapter *is* a Non-Interactive Proof of Proof-of-Work, akin to the charity NIPoPoWs of the previous chapters, and FlyClient [32]. Our difference with FlyClient is the ability to generate *online* proofs, proofs that can be updated as the blockchain grows. Contrary to our construction, FlyClient requires the sampling of past blocks to change as new blocks are added to the tip of the blockchain. This is due to their use of the Fiat–Shamir heuristic [55]. More concretely, a block that was not sampled in the past may need to be sampled in the future. In our protocol, previously pruned blocks never need to be salvaged. As *any* block has a potential for future samplability in FlyClient, no blocks can be discarded, and mining cannot be logarithmic. We are thus the first to propose a NIPoPoW which is *online*, *succinct*, and *secure* against all minority (1/3) adversaries. All of these are necessary prerequisites to achieve the desired goal of logarithmic-space mining.

Chapter 5

Variable Difficulty and Bounded Delay

We now adapt the construction of the previous sections to the variable difficulty model. The chain compression construction and verifier are identical to that the previous section, but with a modified upchain operator as follows. Let C be a chain and $\mu \in \mathbb{N}$. Define the *upchain* $\mathsf{C}{\uparrow}^{\mu}$ of C to be the subchain $\{B \in \mathsf{C} : H(B) \leq 4^{-\mu}T_0\}$ where T_0 is the minimum-difficulty target.

We conjecture that the above construction is secure and succinct in the variable difficulty model as defined in [61]. We sketch below the arguments to support this conjecture, assuming that our Unsupressibility Lemma and the Q-block Common-Prefix Lemma transfer to the variable difficulty model.

Definition 64 (Superqueries)**.** *A random oracle query is called μ-successful if its output does not fall above $2^{\mu-\kappa}T_0$.*

Let

$$X_r{\uparrow}^{\mu} = \begin{cases} 1, \text{ if an honest party queried } H \text{ with a } \mu\text{-successful query during } r \\ 0, \text{ otherwise} \end{cases} .$$

$$Y_r{\uparrow}^{\mu} = \begin{cases} 1 & , \text{ if an honest party queried } H \text{ with a } \mu\text{-successful query during } r \\ & \quad \text{and } r \text{ was a uniquely successful and isolated round} \\ 0 & , \text{ otherwise} \end{cases} .$$

$$Z_{rk}{\uparrow}^{\mu} = \begin{cases} 1, \text{ if the } k^{\text{th}} \text{ adversarial query during round } r \text{ was } \mu\text{-successful} \\ 0, \text{ otherwise} \end{cases} .$$

Let $Z_r = \sum_{k=1}^{t} Z_{rk}$.

Let S be a set of consecutive rounds. Then let $X^S{\uparrow}^{\mu} = \sum_{r \in S} X_r{\uparrow}^{\mu}$, $Y^S{\uparrow}^{\mu} = \sum_{r \in S} Y_r{\uparrow}^{\mu}$ and $Z^S{\uparrow}^{\mu} = \sum_{r \in S} Z_r{\uparrow}^{\mu}$.

Definition 65 (Ground level). *Let r be a round. The* ground level *of r is defined as $\mu^*_r = \lceil \lg(\frac{n_r}{\eta n_0}) \rceil$. Let S be a set of rounds. Then the* ground level *of S is defined as $\mu^*_S = \max_{r \in S} \mu^*_r$.*

Definition 66 (Variable Superblock Typicality). *An execution $\mathcal{E}$ is* superblock-typical *if for all $\mu \in \mathbb{N}$ and for all sets S of consecutive rounds such that $|S| \geq \lambda 2^{\mu}$, the following hold:*

- $|X^S{\uparrow}^{\mu}| < (1+\epsilon)pn(S)2^{-\mu}|S|$.
- *If $\mu \geq \mu^*_S$, then* $|Y^S{\uparrow}^{\mu}| \geq (1-\epsilon)(1-\theta f)^{\Delta+1}pn(S)2^{-\mu}|S|$.
- $|Z^S{\uparrow}^{\mu}| < (1+\epsilon)pt(S)2^{-\mu}|S|$.

Theorem 49 (Variable Superblock Typicality). *Consider an ITM system $(\mathcal{Z}, C)$ which runs in L steps. The probability of the event "$\mathcal{E}$ is* not *superblock-typical" is bounded by $e^{-\Omega(\kappa)+\lg L}$.*

Sketch. □

Lemma 50 (Pairing). *Let r be a Δ-isolated uniquely successful round and B be the block generated by the honest parties during r extending chain C. Consider any other chain C' and a block B' such that $\mathsf{diff}(\mathsf{C}) \leq \mathsf{diff}(\mathsf{C}') < \mathsf{diff}(\mathsf{C}B)$ or $\mathsf{diff}(\mathsf{C}) < \mathsf{diff}(\mathsf{C}'B') \leq \mathsf{diff}(\mathsf{C}B)$. Then B' was computed by the adversary.*

Proof. Observe that $\mathsf{diff}(\mathsf{C}') < \mathsf{diff}(\mathsf{C}B)$ and $\mathsf{diff}(\mathsf{C}'B') > \mathsf{diff}(\mathsf{C})$. We will show that B' could not have been honestly generated during any round r. We distinguish the cases $u - \Delta \leq r < u + \Delta$, $r \geq u + \Delta$, and $r < u - \Delta$. Since u is a Δ-isolated uniquely successful round, block B' could not have been generated by an honest party during any round r with $u - \Delta \leq r < u + \Delta$. During any round $r \geq u + \Delta$, all honest parties have received $\mathsf{C}B$ and so would extend only chains with difficulty at least $\mathsf{diff}(\mathsf{C}B)$, and so they would not extend C' since $\mathsf{diff}(\mathsf{C}') < \mathsf{diff}(\mathsf{C}B)$. During any round $r < u - \Delta$, if an honest party had generated B', then it would have been received by all other honest parties by round u. Since $\mathsf{diff}(\mathsf{C}'B') > \mathsf{diff}(\mathsf{C})$, this contradicts that the honest party extended C during u. □

Definition 67 (Δ-expansion). *Let S be a set of rounds. We define the Δ-expansion of S to be the set $S_{\pm\Delta} = \{r' \in \mathbb{N} : \exists r \in S : |r - r'| \leq \Delta\}$.*

Lemma 51 (Cost of Suppression). *Consider a Δ-isolated uniquely successful round r during which block B was produced by an honest party. If at a later round $r' > r + \Delta$ some honest party has a chain that does not contain B, then there exists a set S of consecutive rounds such that $r \in S$ and $A(J) \geq Q(S)$, where J is the set of adversarial queries made during $S_{\pm\Delta}$.*

Proof. Let C denote the chain that B extends. Let $r' > r + \Delta$ be the first round during which an honest party has a chain C' which does not contain B. It will hold that $\mathsf{diff}(\mathsf{C}') \geq \mathsf{diff}(\mathsf{C}B)$. Let $b = (\mathsf{C} \cap \mathsf{C}')[-1]$ and let b^* be the most recent block preceding b which was honestly generated during a Δ-isolated uniquely successful round. Let r^* denote the round during which b^* was generated and C^* be the chain that b^* extends. Let $S = \{r^* + \Delta, \cdots, r' - \Delta\}$. Then S contains r. Let T

be the set of Δ-isolated uniquely successful rounds in S. For any $r_1, r_2 \in T$ with $r_1 < r_2$ producing b_1 and b_2 and extending C_1 and C_2 respectively, it will hold that $\mathsf{diff}(\mathsf{C}_1 b_1) \leq \mathsf{diff}(\mathsf{C}_2)$ and the difficulties do not overlap. Let J be the set of adversarial queries made during $S_{\pm\Delta}$. We claim that $A(J) \geq Q(S)$. Consider any r'' in T during which block b'' was honestly generated and let b'' extend chain C''. Since r^* was a successful round and $r'' > r^* + \Delta$, therefore $\mathsf{diff}(\mathsf{C}'') \geq \mathsf{diff}(\mathsf{C}^* b^*)$. We will use Lemma 50 to pair the difficulty of block b'' against adversarially generated difficulty. We distinguish two cases. **Case** $b'' \notin \mathsf{C}'$: As $r'' + \Delta \leq r'$, therefore $\mathsf{diff}(\mathsf{C}'' b'') \leq \mathsf{diff}(\mathsf{C}')$ and b'' can be paired against adversarial difficulty in C'. **Case** $b'' \in \mathsf{C}'$: Since r is Δ-isolated uniquely successful, we must have $r'' \leq r - \Delta$ or $r'' \geq r + \Delta$. If $r'' \leq r - \Delta$ then $\mathsf{diff}(\mathsf{C}'' b'') \leq \mathsf{diff}(\mathsf{C}B)$, otherwise B would not have been honestly generated. If, on the other hand, $r'' \geq r + \Delta$, then by the minimality of r', again we obtain $\mathsf{diff}(\mathsf{C}'' b'') \leq \mathsf{diff}(\mathsf{C}B)$. Therefore we can pair b'' against adversarial difficulty in C. To see this, observe that, if $b'' \in \mathsf{C}' \setminus \mathsf{C}$, then it does not belong to C and we are done; but if $b'' \notin \mathsf{C}' \setminus \mathsf{C}$, then b'' cannot be in $\mathsf{C}\{b^*{:}b\}$ by the definition of b^*.

We conclude that

$$A(J) = \sum_{r \in J} A(r) \geq \sum_{r \in T} Q(r) = \sum_{r \in S} Q(r) = Q(S)\,.$$

□

Lemma 52 (Unsuppressibility)**.** *In an (ϵ, η, θ)-typical execution, every set of consecutive rounds U has a subset S of Δ-isolated uniquely successful rounds such that:*

- $|S| \geq Q(U) - 3A(U) - 2\ell(1+\epsilon)\tau\alpha(J)/\epsilon$
- *at any round after $S_{\pm\Delta}$, every honest party has a chain which contains all the blocks corresponding to S.*

where J is the set of adversarial queries in $U_{\pm\Delta}$.

Sketch. Let $U' = \{\min U - \ell, \cdots, \max U + \ell\}$. Let S contain all those $r \in U$ such that for any $S' \subseteq U'$ it holds that $Q(S') > A(J')$ where J' denotes the set of adversarial queries in $S'_{\pm\Delta}$. From typicality it follows that any S' with elements outside of U' cannot have $Q(S') > A(J')$, as $|S'| \geq \lambda$. Consider the minimum collection $\mathcal{T}$ of sets of consecutive rounds with the following properties:

- For all $T \in \mathcal{T} : Q(T) \leq A(T_{\pm\Delta})$.
- $U \setminus S \subseteq \bigcup \mathcal{T}$.

Now, observe that due to the minimality of $\mathcal{T}$, no round r in $U \setminus S$ is covered by more than 2 different spans $T_1, T_2 \in \mathcal{T}$. For, if it were covered by different spans $T_1, T_2, T_3 \in \mathcal{T}$, then we could keep only these two spans among them that are the left-most and right-most.

Call a round $r \in U \setminus S$:

- a *sentinel* if it is covered only by one span in $\mathcal{T}$;
- a *non-sentinel* otherwise.

Let $\mathcal{T}_{\pm\Delta} = \{T_{\pm\Delta} : T \in \mathcal{T}\}$ and let us examine how many $T_{\pm\Delta} \in \mathcal{T}_{\pm\Delta}$ can cover a particular round r. Consider an arbitrary round $r \in U \setminus S$.

If r is a non-sentinel, then it is covered by two spans $T_1, T_2 \in \mathcal{T}$. Consider the largest round $r' < r$ with $r' \in U \setminus S$, and note that such a round must exist. Necessarily, r' will be covered be either T_1 or T_2, or both, but by no other span (to see this, observe that if, say, r' was covered by T_2 and not T_1 but also by some other span T_3, then T_2 would be needless and this would contradict the minimality of $\mathcal{T}$). As $r - r' \geq \Delta$, any span T' which covers rounds prior to r' cannot extend to the right to cover r; that is, $\max T'_{\pm\Delta} < r$. A symmetric argument can be made for any spans on the right side of r. Therefore, non-sentinel rounds can only be covered by up to two spans in $\mathcal{T}_{\pm\Delta}$.

If r is a sentinel, then up to two spans (one from the left and one from the right) can be Δ-extended to cover it, so it can be covered by up to three spans in $\mathcal{T}_{\pm\Delta}$.

Let $q = Q(U)$ and $a = A(U)$. We need to show that $q - |S| \leq 3a + 2\ell(1+\epsilon)\tau\alpha(J)/\epsilon$.

$$\begin{aligned} q - |S| = Q(U \setminus S) &\leq \sum_{T\in\mathcal{T}} Q(T) \leq \sum_{T\in\mathcal{T}} A(T_{\pm\Delta}) \\ &\leq 3a + A(U' \setminus U) \leq 3a + 2\ell(1+\epsilon)\tau\alpha(J)/\epsilon \end{aligned}$$

□

Remark 8 (Bounds in the q-bounded setting)**.** *We note that the $|S| \gtrsim Q(U) - 3A(U)$ bound above requires a $\frac{1}{4}$-bounded adversary. This bound is due to the Δ-bounded delay setting. However, the same proof can show that $|S| \gtrsim Q(U) - 2A(U)$ in the case of the variable difficulty model in the q-bounded synchronous setting. Therefore, a $\frac{1}{3}$-bounded adversary can be tolerated. The same bounds in the synchronous setting apply in all of the next theorems, establishing security against a $\frac{1}{3}$ adversary in the variable difficulty q-bounded synchronous setting and against a $\frac{1}{4}$ adversary in the variable difficulty Δ-bounded delay setting.*

Theorem 53 (Variable Super Common Prefix)**.** *Consider an (ϵ, η, θ)-typical execution in a (γ, s)-respecting environment and suppose that for all r it holds that $t_r < (\frac{1}{4} - \delta)n_r$ with $\frac{\delta}{2} > 2\epsilon + \theta f$. Consider a chain C adopted by an honest party on round r and let C' be any other chain at round r. Let $b = (\mathsf{C} \cap \mathsf{C}')[-1]$ and b^* be the most recent honest block in $\mathsf{C} \cap \mathsf{C}'$ produced at round r_{b^*}. Let $S = \{r_{b^*}, \cdots, r\}$ and $\mu \geq \mu^*_S$. If $|C'\{b{:}\}{\uparrow}^{\mu}| \geq 22\lambda f 2^{-\mu}$, then $|C\{b{:}\}{\uparrow}^{\mu}| > |C'\{b{:}\}{\uparrow}^{\mu}|$.*

Sketch. Let r^* be the round in which the most recent honestly generated block in $(\mathsf{C} \cap \mathsf{C}')[-1]$. Let $S = \{r^*, \cdots, r\}$. Let W' denote the set of adversarial queries on $\mathsf{C}' \setminus \mathsf{C}$ at round $r' \geq r + \ell$ and W the set of adversarial queries in $S \setminus W'$.

We first observe that no query in W' could have suppressed a μ-superblock on $\mathsf{C} \setminus \mathsf{C}'$. As in the proof of Lemma 51, in such a case there would exist a set of consecutive rounds $|S^*| \geq \ell$ such that $Q(S^*) \leq A(J)$ where J is the number of adversarial queries in $S^*_{\pm\Delta}$. This contradicts typicality.

□

Lemma 54 (Variable Multilevel Common Prefix)**.** *Consider an (ϵ, η, θ)-typical execution with $\tau \leq 2$. Let C be an honestly adopted chain at round r and let C' be any other chain. Consider any $\mu' \geq \mu^*_r$ such that $|(\mathsf{C}' \setminus \mathsf{C}){\uparrow}^{\mu'}| > \frac{m}{2}$. Then there exists some $\mu > \mu'$ such that $|(\mathsf{C} \setminus \mathsf{C}'){\uparrow}^{\mu}| > \frac{m}{2}$ and $2^{\mu}|(\mathsf{C} \setminus \mathsf{C}'){\uparrow}^{\mu}| > 2^{\mu'}|(\mathsf{C}' \setminus \mathsf{C}){\uparrow}^{\mu'}|$.*

Definition 68 (Admissible Chain). *Consider a chain π during a round r and let k' be the* minimum *parameter such that diff($\pi[-k':]) \geq k\frac{n_r}{\eta n_0}$. Let $b = \pi[:-k'][-1]$ and consider the round r_b during which b was produced. We say that π is* admissible *at round r if:*

- *b belongs to the chain of an honest party during r (persistence)*
- *$r_b \geq r - \frac{k}{\eta f}$ (liveness)*

Lemma 55 (Honest Chopping). *Consider a NIPoPoW π produced by an honest party during round r. Then π is admissible at r.*

Sketch. Let k' be the *minimum* parameter such that $\text{diff}(\pi[-k':]) \geq k\frac{n_r}{\eta n_0}$. Clearly $b = \pi[-k':][-1]$ belongs to the chain of the honest party that produced π. It suffices to show that $r_b \geq r - \frac{k}{\eta f}$. The condition holds because $m > k \geq k'$. □

Theorem 56 (Variable NIPoPoW Admissibility). *Consider an (ϵ, η, θ)-typical execution in a (γ, s)-respecting environment such that for all r we have $t_r < (\frac{1}{4} - \delta)n_r$. Let π and π' be NIPoPoWs generated by an honest prover B and adversarial prover $\mathcal{A}$ respectively during round r. Let π^* be the NIPoPoW selected by the honest verifier among π and π'. Suppose $\vec{n}$ is increasing during $[r]$ and that $V([r])$ is appropriately upper bound. Then π^* is admissible at r.*

Sketch. If π is chosen, then both admissibility conditions hold by Lemma 55. Therefore, we will show that, if π' is chosen, then it is admissible. Let $\mathcal{M} = \{\mu : \pi{\uparrow^\mu} \cap \pi'{\uparrow^\mu} \neq \emptyset\}$. **Case 1: $\mathcal{M} = \emptyset$.** Consider $r^* = 0$ and let $S = \{0, \cdots, r\}$. If $\mathcal{M} = \emptyset$ then... chopping off will get us to genesis, which is honest and recent. **Case 2: $\mathcal{M} \neq \emptyset$.** Let $\mu = \min \mathcal{M}$. Let $b = (\pi{\uparrow^\mu} \cap \pi'{\uparrow^\mu})[-1]$ and let b^* be the most recent honest block preceding b. Let r^* be the round during which b^* was generated and set $S = \{r^*, \cdots, r\}$. Let $\mu^* = \lceil \lg \frac{n_{r_\pi}}{n_0} \rceil$. **Case 2a: $\mu > \mu^*$.** By the minimality of μ, it will hold that $|\pi\{b:\}{\uparrow^\mu}| \geq m$ (otherwise $b \in D[\mu-1] \cap D'[\mu-1] \neq \emptyset$). Furthermore, $\mathsf{C}\{b:\}{\uparrow^\mu} = \pi\{b:\}{\uparrow^\mu}$. Apply Theorem 53 to obtain $|\pi\{b:{\uparrow^\mu}\}| > |\pi'\{b:\}{\uparrow^\mu}|$. Therefore, in this case, π will be chosen. **Case 2b: $\mu \leq \mu^*$.** If the adversarial proof is chosen, then chopping off will get us to an honest fresh block because, from typicality, $\mathsf{diff}(\pi'\{b:\}) < A(\{r_{b^*}, \cdots, r\})$ must also be small. □

Remark 9 (Dropping difficulty). *The above theorem establishes that a NIPoPoW verifier picks an admissible proof in cases of non-decreasing difficulty. If difficulty is allowed to decrease, then we can still establish a persistence guarantee in sacrifice of liveness. Let $b = (\pi{\uparrow^\mu} \cap \pi'{\uparrow^\mu})[-1]$ where $\mu = \min \mathcal{M}$ as in the construction and let r^* be the round during which the most recent honestly generated block preceding b was generated. If $\mathcal{M} = \emptyset$ then set $r^* = 0$. Let $S = \{r^*, \cdots, r\}$ and $\mu^* = \max_{r' \in S} \mu^*_{r'}$. First, observe that it is sufficient to only require difficulty to be non-decreasing during S, but can be allowed to decrease before S, and this ensures both liveness and persistence for the NIPoPoW verifier. In case difficulty has decreased* during *S, then chopping off the minimum-length suffix with at least $2^{\mu^*}k$ difficulty from the selected proof π^* ensures that the remaining chain has a tip adopted by all honest parties (establishing persistence), but may be stale (sacrificing liveness), as it could have been generated prior to round $r - \frac{k}{\eta f}$.*

Discovering the current difficulty. The last missing piece in creating a full variable-difficulty verifier is to instruct the verifier to *chop off* the correct amount of difficulty from an admissible chain so that they obtain a stable and fresh block, achieving persistence and liveness comparable to an honest full node. If they chop off too little, they will arrive at a blockchain with non-stable tip, sacrificing persistence. On the other hand, if they chop off too much, they will sacrifice liveness, as their tip will be too old. Our goal is to ensure the NIPoPoW verifier has comparable persistence and liveness to a full node. To chop off the correct amount, the verifier needs to estimate the value of the ground level μ^*. The ground level can be estimated if the verifier can estimate the current mining population. The verifier first finds an estimate $\widetilde{n}$ for the population, and then sets $\mu^* = \lceil \lg \frac{n_0}{\eta \widetilde{n}} \rceil$. The question is then how to estimate the population.

If we had a randomness beacon emitting every $\frac{m}{f}$ rounds, a NIPoPoW verifier waking up from genesis and with only logarithmic state and communication available could estimate the population after $2\frac{m}{f}$ rounds, since at least two beacon pulses will be emitted during them. The verifier works as follows in this case. When the beacon emits the first pulse, it begins collecting any block it sees on the network, regardless of which chain it belongs to —or indeed whether it belongs to a chain at all— until the second beacon pulse is detected. These blocks are collected into a set S. The verifier then evalutes $\mathsf{diff}(S)$ as an estimate of the current difficulty and estimates the current population as $\widetilde{n} = \frac{\mathsf{diff}(S) f 2^\kappa}{n_0 m T_0}$. This certainly constitutes a lower bound for the amount of difficulty queried during the epoch, but may include stale blocks that were queried by the adversary and rebroadcast to confuse the NIPoPoW verifier. To avoid this, since we have a beacon available, we require every honest party to include the beacon randomness in every block they generate. That way, the verifier only counts those blocks which include a reference to the same randomness that it sees.

Based on the above protocol, we can now tackle the lingering question of how to avoid the strong assumption that such a beacon entails. Note that the need for the beacon stems from the requirement that the adversary should not be able to predict its value and therefore is required to produce *fresh* blocks to convince the honest NIPoPoW verifier about the difficulty currently being queried. The same unpredictability can be obtained if the NIPoPoW verifier provides this value. In fact, we can implement this without any changes to the standard protocol. When the NIPoPoW verifier wakes up during round r, it produces a fresh random nonce R. It then creates a transaction tx containing R. We call this transaction a *difficulty weather balloon* because it will be diffused to the network to allow the verifier to make difficulty measurements. Once the balloon has been diffused, from the *liveness* property of the ledger maintained by the full nodes, it will be included in a block which will become stable within u rounds where $u = \frac{m}{16\tau u f} + \frac{\gamma k}{\eta f(1-\epsilon)(1-\theta f)}$ denotes the liveness parameter.

The verifier counts all blocks that appear on the network from round $r + u$ to round $r + u + \frac{m}{f}$. A block is collected into the set S if it extends a chain C whose suffix $\mathsf{C}[-\lceil \frac{(1+\epsilon)(m+u)}{\eta} \rceil{:}]$ contains a block containing tx. The requirement to look only at the suffix ensures the NIPoPoW verifier does not need more than constant communication resources and is not required to look at the full chain. The following lemma ensures that the verifier will count all the difficulty contributed by the honest parties.

Lemma 57. *Consider a typical execution and a transaction tx diffused at round r. Then all blocks honestly generated during rounds $\{r + \Delta + u, \cdots, r + u + \frac{m}{f}\}$ will* *each result in a chain* C *whose suffix* $\mathsf{C}[-\lceil\frac{(1+\epsilon)(m+u)}{\eta}\rceil{:}]$ *will contain* tx*.*

Sketch. From the liveness of the underlying ledger, after u rounds tx will be included in the chain of every honest party. However, since tx was generated at r, the chain cannot have grown more than $\frac{-(1+\epsilon)(m+u)}{\eta}$ during $u + \frac{m}{f}$ rounds. □

The above ensures that $\mathsf{diff}(S) \geq Q(\{r+u, \cdots, r+u+\frac{m}{f}\})$. This satisfies the persistence requirement of the verifier. On the other hand, the freshness of tx ensures that the adversary could not have contributed too much work to S. The following lemma makes this more precise:

Lemma 58. *Consider an* (ϵ, η, θ)*-typical execution. Let* tx *be a transaction diffused during* r *and let* S *be the set of blocks observed by an honest party during rounds* $r+u, \cdots, r+u+\frac{m}{f}$ *each of which creates a chain* C *in which* tx *is included in* $\mathsf{C}[-\lceil\frac{(1+\epsilon)(m+u)}{\eta}\rceil{:}]$*. Let* $U = \{r, \cdots, r+u+\frac{m}{f}\}$*.*

Then we have:

$$Q(\{r+u-\Delta, \cdots, r+u+\frac{m}{f}+\Delta\}) \leq \mathit{diff}(S) \leq Q(U) + A(U)$$

Sketch. The first inequality follows from Lemma 57. For the second, observe that the adversary cannot use any queries made prior to r. □

Removing balloon interactivity. The above protocol introduces some interactivity in the sense that the verifier must produce the nonce in the balloon. However, such interactivity is unnecessary as long as the verifier remains online for the duration of the difficulty sampling. We can modify our protocol to be non-interactive as follows. Initially, the verifier comes online at round r and expects to be provided with NIPoPoWs supporting an admissible chain. These NIPoPoWs must be generated and diffused within round r. Any interested parties (including honest and adversarial parties) generate such NIPoPoWs and diffuse them on the network. At round $r+1$, any honest parties who have diffused NIPoPoWs look for competing proofs that have been provided by other honest parties or the adversary. They collect all such proofs into a set $\diamond$, which they order and encode. They subsequently calculate the hash h of the result. This hash commits to all NIPoPoWs and can also be calculated by the NIPoPoW verifier. The honest parties then diffuse a balloon transaction containing h as a nonce. The NIPoPoW verifier calculates the same nonce from the NIPoPoWs it has received and observes the difficulty collection on the network as before. As soon as one epoch completes, the NIPoPoW verifier can use the difficulty estimate as before.

Theorem 59 (Security)**.** *Consider an* (ϵ, η, θ)*-typical execution in the variable difficulty* Δ*-bounded delay setting so that for every* i *we have* $t_i < (\frac{1}{4} - \delta)n_i$*. Then the NIPoPoW prover of Algorithm 21, coupled with the verifier which follows the balloon technique, is a secure PoPoW protocol.*

Sketch. The theorem follows directly from Theorem 56 and Lemma 58. □

Theorem 60 (Succinctness)**.** *If* $m \in O(\log|\mathsf{C}|)$ *and computational power is non-decreasing, then the superblock NIPoPoW construction is succinct.*

Sketch. The blocks of any level are sufficiently concentrated in the second half of the chain (modulo the blocks that were suppressed by the adversary, which are bounded in number due to the Unsuppressibility Lemma). Thus, after $O(\log|C|)$ levels of the proof, the chain will be exhausted for a total length in $O(m \log |C|)$. $\square$

We remark that the number of blocks that must be inspected in any balloon sampling is constant, and therefore the inclusion of such samples does not change the succinctness of the verifier.

Chapter 6

Proofs of Proof-of-Stake

In this short chapter, we introduce a novel cryptographic primitive, *ad-hoc threshold multisignatures (ATMS)*, fundamental building block for cross-chain certification in proof-of-stake. We discuss specific instantiations of ATMS in Section 6.2. We implement them in three distinct ways. The first one simply concatenates signatures of elected slot leaders. While secure, the disadvantage of this implementation is that the size of the sidechain certificate is $\Theta(k)$ signatures. An improvement can be achieved by employing multisignatures and Merkle-tree hashing for verification key aggregation; using this we can drop the sidechain-certificate size to $\Theta(r)$ signatures where r slot leaders do not participate in its generation; in the optimistic case $r \ll k$ and thus this scheme can be a significant improvement in practice. Finally, we show that STARKs and bulletproofs [21, 31] can be used to bring down the size of the certificate to be optimally succinct in the random oracle model. We observe that in the case of an active sidechain (e.g., one that returns assets at least once per epoch) our construction with succinct sidechain certificates has optimal storage requirements in the mainchain.

These ATMS will be leveraged in the next chapters to build full sidechains in the proof-of-stake model.

6.1 Ad-Hoc Threshold Multisignatures

We introduce a new primitive, *ad-hoc threshold multisignatures (ATMS)*, which borrow properties from multisignatures and threshold signatures and are ad-hoc in the sense that signers need to be selected on the fly from an existing key set. In Section 7.1.2 we describe how ATMS are useful for periodically updating the "anchor of trust" that the mainchain parties have w.r.t. the sidechain they are not following.

ATMS are parametrized by a threshold t. On top of the usual digital signatures functionality, ATMS also provide a way to: (1) aggregate the public keys of a subset of these parties into a single aggregate public key avk; (2) check that a given avk was created using the right sequence of individual public keys; and (3) aggregate $t' \geq t$ individual signatures from t' of the parties into a single aggregate signature that can then be verified using avk, which is impossible if less than t individual signatures are used.

The definition of an ATMS is given below.

Definition 69. *A t-ATMS is a tuple of algorithms* $\Pi = (\mathit{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver}, \mathit{AKey}, \mathit{ACheck}, \mathit{ASig}, \mathit{AVer})$ *where:*

$\mathit{PGen}(1^\kappa)$ *is the parameter generation algorithm that takes the security parameter* 1^κ *and returns system parameters* $\mathcal{P}$*.*

$\mathsf{Gen}(\mathcal{P})$ *is the key-generation algorithm that takes* $\mathcal{P}$ *and produces a public/private key pair* (vk_i, sk_i) *for the party invoking it.*

$\mathsf{Sig}(sk_i, m)$ *is the signature algorithm as in an ordinary signature scheme: it takes a private key and a message and produces a (so-called* local*) signature* σ*.*

$\mathsf{Ver}(m, pk_i, \sigma)$ *is the verification algorithm that takes a public key, a message and a signature and returns* true *or* false*.*

$\mathit{AKey}(\mathcal{VK})$ *is the key aggregation algorithm that takes a sequence of public keys* $\mathcal{VK}$ *and aggregates them into an* aggregate public key avk*.*

$\mathit{ACheck}(\mathcal{VK}, avk)$ *is the aggregation-checking algorithm that takes a public key sequence* $\mathcal{VK}$ *and an aggregate public key* avk *and returns* true *or* false*, determining whether* $\mathcal{VK}$ *were used to produce* avk*.*

$\mathit{ASig}(m, \mathcal{VK}, \langle(vk_1, \sigma_1), \cdots, (vk_d, \sigma_d)\rangle)$ *is the signature-aggregation algorithm that takes a message* m*, a sequence of public keys* $\mathcal{VK}$ *and a sequence of* d *pairs*

$$\langle(vk_1, \sigma_1), \cdots, (vk_d, \sigma_d)\rangle$$

where each σ_i *is a local signature on* m *verifiable by* vk_i *and each* vk_i *is in a distinct position within* $\mathcal{VK}$*,* ASig *combines these into a multisignature* σ *that can later be verified with respect to the aggregate public key* avk *produced from* $\mathcal{VK}$ *(as long as* $d \geq t$*, see below).*

$\mathit{AVer}(m, avk, \sigma)$ *is the aggregate-signature verification algorithm that takes a message* m*, an aggregate public key* avk*, and a multisignature* σ*, and returns* true *or* false*.*

Definition 70 (ATMS correctness)**.** *Let* Π *be a* t*-ATMS scheme initialized as* $\mathcal{P} \leftarrow \mathit{PGen}(1^\kappa)$*, let* $(vk_1, sk_1), \cdots, (vk_n, sk_n)$ *be a sequence of keys generated via* $\mathsf{Gen}(\mathcal{P})$*, let* $\mathcal{VK}$ *be a sequence containing (not necessarily unique) keys from the above and* avk *be generated by invoking* $avk \leftarrow \mathit{AKey}(\mathcal{VK})$*. Let* m *be any message and let* $\langle(vk_1, \sigma_1), \cdots, (vk_d, \sigma_d)\rangle$ *be any sequence of key/signature pairs provided that* $d \geq t$ *and every* vk_i *appears in a unique position in the sequence* $\mathcal{VK}$*, where* σ_i *is generated as* $\sigma_i = \mathsf{Sig}(sk_i, m)$*. Let* $\sigma \leftarrow \mathit{ASig}(m, \mathcal{VK}, \langle(vk_1, \sigma_1), \cdots, (vk_d, \sigma_d)\rangle)$*. The scheme* Π *is* correct *if for every such message and sequence the following hold:*

1. $\mathsf{Ver}(m, vk_i, \sigma_i)$ *is* true *for all* i*;*

2. $\mathit{ACheck}(\mathcal{VK}, avk)$ *is* true*;*

3. $\mathit{AVer}(m, avk, \sigma)$ *is* true*.*

We define the security of an ATMS in the definition below, via a cryptographic game given in Algorithm 39.

Algorithm 39 The game $\mathsf{ATMS}_{\Pi,\mathcal{A}}$

The game is parameterized by a security parameter κ and an integer $p(\kappa)$.

```
 1: function $\mathsf{ATMS}_{\Pi,\mathcal{A}}(\kappa, p(\kappa))$
 2:     $\mathcal{VK} \leftarrow \epsilon; \mathcal{SK} \leftarrow \epsilon; Q^{\mathsf{sig}} \leftarrow \emptyset; Q^{\mathsf{cor}} \leftarrow \emptyset$
 3:     $\mathcal{P} \leftarrow \mathsf{PGen}(1^{\kappa})$
 4:     $(m, \sigma, avk, \mathsf{keys}) \leftarrow \mathcal{A}^{\mathcal{O}^{\mathsf{gen}}, \mathcal{O}^{\mathsf{sig}}(\cdot,\cdot), \mathcal{O}^{\mathsf{cor}}(\cdot)}(\mathcal{P})$
 5:     $q \leftarrow 0$
 6:     for $vk$ in keys do
 7:         if $vk \notin \mathcal{VK} \vee vk \in Q^{\mathsf{sig}}[m] \cup Q^{\mathsf{cor}}$ then
 8:             $q \leftarrow q + 1$
 9:         end if
10:     end for
11:     return $\mathsf{AVer}(m, avk, \sigma) \wedge \mathsf{ACheck}(\mathsf{keys}, avk) \wedge q < t$
12: end function
13: function $\mathcal{O}^{\mathsf{gen}}$
14:     $(vk, sk) \leftarrow \mathsf{Gen}(\mathcal{P})$
15:     $\mathcal{VK} \leftarrow \mathcal{VK} \| vk$
16:     $\mathcal{SK} \leftarrow \mathcal{SK} \| sk$ return $vk$
17: end function
18: function $\mathcal{O}^{\mathsf{sig}}(i, m)$
19:     $Q^{\mathsf{sig}}[m] \leftarrow Q^{\mathsf{sig}}[m] \cup \{\mathcal{VK}[i]\}$ return $\mathsf{Sig}(\mathcal{SK}[i], m)$
20: end function
21: function $\mathcal{O}^{\mathsf{cor}}(i)$
22:     $Q^{\mathsf{cor}} \leftarrow Q^{\mathsf{cor}} \cup \{\mathcal{VK}[i]\}$ return $\mathcal{SK}[i]$
23: end function
```

Figure 6.1: The ATMS security game $\mathsf{ATMS}_{\Pi,\mathcal{A}}$.

Definition 71 (Security). *A t-ATMS scheme* $\Pi = (\mathit{PGen}, \mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver}, \mathit{AKey}, \mathit{ACheck}, \mathit{ASig}, \mathit{AVer})$ *is* secure *if for any PPT adversary $\mathcal{A}$ and any polynomial p there exists some negligible function* negl *such that* $\Pr[\mathit{ATMS}_{\Pi,\mathcal{A}}(\kappa, p(\kappa)) = 1] < \mathsf{negl}(\kappa)$.

The quantity q in the ATMS game counts how many keys the adversary is in control of among her chosen keys keys which will be used for aggregate-signature verification. The sequence keys can contain both adversarially-generated keys as well as some of the keys $\mathcal{VK}$ honestly generated by the challenger. The variable q counts the number of adversarially controlled keys in keys. This includes those keys in keys for which the adversary has obtained a signature for the message in question (through the use of the oracle $\mathcal{O}^{\mathsf{sig}}(\cdot)$) or which the adversary has corrupted completely (through the use of the oracle $\mathcal{O}^{\mathsf{cor}}(\cdot)$), as well as those keys which have been generated by the adversary herself and therefore are not in $\mathcal{VK}$.

It is straightforward to see that if Π is a secure ATMS, then the tuple (PGen, $\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver}$) is a EUF-CMA-secure signature scheme.

Looking ahead, note that since the AKey algorithm is only invoked with the public keys of the participants, it can be invoked by anyone, not just the parties who hold the respective secret keys, as long as the public portion of their keys is published. Furthermore, notice that the above games allow the adversary to generate more public/private key pairs of their own and combine them at will.

Concrete instantiations of the ATMS primitive are presented in the next sections.

6.2 Constructing Ad-Hoc Threshold Multisignatures

In this section we give several ways to instantiate the ATMS primitive. We order them by increasing succinctness but also increasing complexity.

6.2.1 Plain ATMS

Given a EUF-CMA-secure signature scheme, combining signatures and keys can be implemented by plain concatenation. Subsequently, combined verification requires all signatures to be verified individually. This illustrates that the ATMS primitive is easy to realize if no concern is given to succinctness. The size of these aggregate signatures and aggregate keys is *quadratic* in the security parameter κ: for the aggregate key $2k$ individual keys of size κ bits each are concatenated (with $k = \Theta(\kappa)$), while the aggregate signature consists of at least $k+1$ individual signatures of size κ bits.

6.2.2 Multisignature-based ATMS

The previous construction can be improved by employing an appropriate multisignature scheme. In the construction below, we consider the multisignature scheme Π_{MGS} from [26].

We make use of a homomorphic property of this scheme: any d individual signatures $\sigma_1, \ldots, \sigma_d$ created using secret keys belonging to (not necessarily unique) public keys $vk_1, \ldots, vk_d$ can be combined into a multisignature $\sigma = \prod_{i=1}^{d} \sigma_i$ that can then be verified using an aggregated public key $avk = \prod_{i=1}^{d} vk_i$.

Our multisignature-based t-ATMS construction works as follows: the procedures PGen, Gen, Sig and Ver work exactly as in Π_{MGS}. Given a set S, denote by $\langle S \rangle$ a Merkle-tree commitment to the set S created in some arbitrary, fixed, deterministic way. Procedure AKey, given a sequence of public keys $\mathcal{VK} = \{vk_i\}_{i=1}^{n}$ returns $avk = (\prod_{i=1}^{n} vk_i, \langle \mathcal{VK} \rangle)$. Since AKey is deterministic, $\mathsf{ACheck}(\mathcal{VK}, avk)$ simply recomputes it to verify avk. ASig takes the message m, d pairs of signatures with their respective public keys $\{\sigma_i, vk_i\}_{i=1}^{d}$ and $n-d$ additional public keys $\{\widehat{vk}_i\}_{i=1}^{n-d}$ and produces an aggregate signature

$$\sigma = \left(\prod_{i=1}^{d} \sigma_i, \{\widehat{vk}_i\}_{i=1}^{n-d}, \{\pi_{\widehat{vk}_i}\}_{i=1}^{n-d} \right) \tag{6.1}$$

where $\pi_{\widehat{vk}_i}$ denotes the (unique) inclusion proof of $\widehat{vk}_i$ in the Merkle commitment

$$\left\langle \{vk_i\}_{i=1}^{d} \cup \{\widehat{vk}_i\}_{i=1}^{n-d} \right\rangle .$$

Finally, the procedure AVer takes a message m, an aggregate key avk, and an aggregate signature σ parsed as in (6.1), and does the following: (a) verifies that each of the public keys $\widehat{vk}_i$ indeed belongs to a different leaf in the commitment $\langle \mathcal{VK} \rangle$ in avk using membership proofs $\pi_{\widehat{vk}_i}$; (b) computes avk' by dividing the first part of avk by $\prod_{i=1}^{n-d} \widehat{vk}_i$; (c) returns *true* if and only if $d \geq t$ and the first part of σ verifies as a Π_{MGS}-signature under avk'.

Note that the scheme Π_{MGS} requires vk_i to be accompanied by a (non-interactive) proof-of-possession (POP) [129] of the respective secret key. This POP can be appended to the public key and verified when the key is communicated in the protocol. For conciseness, we omit these proofs-of-knowledge from the description (but we include them in the size calculation below).

Asymptotic Complexity. This provides an improvement in our use case over the plain scheme: In the optimistic case where each of the $2k$ committee members create their local signatures, both the aggregate key avk and the aggregate signature σ are *linear* in the security parameter, which is optimal. If $r < k$ of the keys do *not* provide their local signatures, the construction falls back to being *quadratic* in the worst case if $r = k - 1$. However, for the practically relevant case where $r \ll k$ and almost all slot leaders produced a signature, this construction is clearly preferable.

Concrete space requirements. Concrete signature sizes in this scheme for practical parameters could be as follows. We set $k = 2160$ (as is done in the Cardano implementations of [89]) and for the signature of [26] we have in bits: $|vk_i| = 272$, $|\sigma_i| = 528$ (N. Di Prima, V. Hanquez, personal communication, 16 Mar 2018), with $|vk_i + POP| = |vk_i| + |\sigma_i| = 800$ bits. Assuming 256-bit hash function is used for the Merkle tree construction, the size of the data which needs to be included in **MC** in the optimistic case during an epoch transition is $|avk| + |\sigma| + |\langle \mathsf{pending} \rangle| = |vk_i + POP| + 2|H(\cdot)| + |\sigma_i| = 800 + 512 + 528 = 1840$ bits per epoch. In a case where 10% of participants fail to sign, the size will be $|avk| + |\sigma| = |vk_i + POP| + 2|H(\cdot)| + |\sigma_i| + 0.1 \cdot 2 \cdot k(|vk_i + POP| + \log(k)|H(\cdot)) = 800 + 512 + 528 + 432 \cdot (500 + 12 \cdot 256) = 1544944$, or about 190 KB per epoch (which is approximately 5 days).

6.2.3 ATMS From Proofs of Knowledge

While the aggregate signatures construction seems sufficient for practice, it still requires a $\mathsf{sc\_cert}$ transaction that is in the worst case quadratic in the security parameter. The approach below, based on proofs of knowledge, improves on that.

We define $avk \leftarrow \mathsf{AKey}(\mathcal{VK})$ to be the root of a Merkle tree that has $\mathcal{VK}$ at its leaves. Let $\mathsf{Sig}, \mathsf{Ver}$ come from any secure signature scheme. In our ATMS, the local signature is equal to $s_i = \mathsf{Sig}(sk_i, m)$, where sk_i is the secret key that corresponds to the vk_i verification key. Letting $S' = \{s_i\}$ be the signatures generated by a sequence $\mathcal{VK}'$ containing keys in $\mathcal{VK}$, the $\mathsf{ASig}(\mathcal{VK}, S', m)$ algorithm reconstructs the Merkle tree from $\mathcal{VK}$ and determines the membership proof π_i for each $vk_i \in \mathcal{VK}'$. Regarding the non-interactive argument of knowledge, the statement of interest is (avk, m) with witness $\{\pi_i, (s_i, vk_i)\}_{i \in S'}$ such that for all i we have that $\mathsf{Ver}(vk_i, m, s_i) = 1$ and π_i is a valid Merkle tree proof pointing to a unique leaf for every i. π_i demonstrates that vk_i is in avk. We also require $|S'| \geq t$. It is possible to construct succinct proofs for this statement using SNARKs [24] or even without any trusted setup using e.g., STARKs [21] or Bulletproofs [31] in the Random Oracle model [19]. In both cases the actual size of the resulting signature will be at most logarithmic in k, while in the case of STARKs the verifier will also have time complexity logarithmic in k.

Chapter 7

Sidechains

We now have all the required tools and primitives to build sidechains. In this chapter we construct full blockchain interoperability. We begin in Section 7.1 by giving the definition of what sidechains are and a construction for stake-based blockchains first, making use of the ATMS primitive defined and constructed in Chapter 6. In Section 7.2 we show how one can construct sidechains for work-based blockchains using the NIPoPoWs primitive which was discussed in Chapter 3. These two sidechain constructions are bidirectional. In Section 7.3 we explore how money can be destroyed on one system and re-created in another, giving rise to unidirectional sidechains.

7.1 Bidirectional Sidechains with Stake Sources

In this section we give the first formal definition of security desiderata for a system of pegged ledgers (popularly often called sidechains). We start by conveying its intuition and then proceed to the formal treatment. We will first present a generic framework defining sidechain security upon which we will build a solution for stake-based blockchains. In the later sections of this chapter, we will explore how this can be instantiated for proof-of-work.

To prove any meaningful security guarantees for the executed protocol without further restrictions (as it, for example, does not prevent the adversary from corrupting all the participating parties), we will consider such additional assumptions, and will only provide security guarantees as long as such assumptions are satisfied. These assumptions will be specific to the protocol in consideration, and will be an explicit part of our statements.[1]

Without loss of generality, we give a detailed presentation of our construction on a generic PoS protocol based on Ouroboros PoS [89] (see Chapter 2 for an overview of Ouroboros). We give an overview of our construction for other proof-of-stake schemes in Section 7.1.5, in particular for Ouroboros Praos [45], Ouroboros Genesis [13], Snow White [22], and Algorand [113].

In this section, we formalize the notion of sidechains by proposing a rigorous cryptographic definition, the first one to the best of our knowledge. The definition

[1]As an example, we will be assuming that a majority of a certain pool of stake is controlled by uncorrupted parties.

is abstract enough to be able to capture the security for blockchains based on proof-of-work, proof-of-stake, and other consensus mechanisms.

A critical security feature of a sidechain system that we formalise is the *firewall property* in which a catastrophic failure in one of the chains, such as a violation of its security assumptions, does not make the other chains vulnerable providing a sense of limited liability.[2] The firewall property formalises and generalises the concept of a blockchain *firewall* which was described in high level in [12]. Informally the blockchain firewall suggests that no more money can ever return from the sidechain than the amount that was moved into it. Our general firewall property allows relying on an arbitrary definition of exactly how assets can correctly be moved back and forth between the two chains, we capture this by a so-called *validity language*. In case of failure, the firewall ensures that transfers from the sidechain into the mainchain are rejected unless there exists a (not necessarily unique) plausible history of events on the sidechain that could, in case the sidechain was still secure, cause the particular transfers to take place.

In this section, we outline a concrete exemplary construction for sidechains for proof-of-stake blockchains. For conciseness our construction is described with respect to a generic PoS blockchain consistent with the Ouroboros protocol [89] that underlies the Cardano blockchain, which is currently one of the largest pure PoS blockchains by market capitalisation,[3] nevertheless we also discuss how to modify our construction to operate for Ouroboros Praos [45], Ouroboros Genesis [13], Snow White [22] and Algorand [113].

We prove our construction secure using standard cryptographic assumptions. We show that our construction (i) supports safe cross-chain value transfers when the security assumptions of both chains are satisfied, namely that a majority of honest stake exists in both chains, and (ii) in case of a one-sided failure, maintains the firewall property, thus containing the damage to the chains whose security conditions have been violated.

A critical consideration in a sidechain construction is safeguarding a new sidechain in its initial "bootstrapping" stage against a "goldfinger" type of attack [94, 27]. Our construction features a mechanism we call *merged-staking* that allows mainchain stakeholders who have signalled sidechain awareness to create sidechain blocks even without moving stake to the sidechain. In this way, sidechain security can be maintained assuming honest stake majority among the entities that have signaled sidechain awareness that, especially in the bootstrapping stage, are expected to be a large superset of the set of stakeholders that maintain assets in the sidechain.

Our techniques can be used to facilitate various forms of 2-way peggings between two chains. As an illustrative example we focus on a parent-child mainchain-sidechain configuration where sidechain nodes follow also the mainchain (what we call direct observation) while mainchain nodes need to be able to receive crypto-

[2]To follow the analogy with the term of limited liability in corporate law, a catastrophic sidechain failure is akin to a corporation going bankrupt and unable to pay its debtors. In a similar fashion, a sidechain in which the security assumptions are violated may not be able to cover all of its debtors. We give no assurances regarding assets residing on a sidechain if its security assumptions are broken. However, in the same way that stakeholders of a corporation are personally protected in case of corporate bankruptcy, the mainchain is also protected in case of sidechain security failures. Our security will guarantee that each incoming transaction from a sidechain will always have a valid explanation in the sidechain ledger independently of whether the underlying security assumptions are violated or not. A simple embodiment of this rule is that a sidechain can return to the mainchain at most as many coins as they have been sent to the sidechain over all time.

[3]See `https://coinmarketcap.com`.

graphically certified signals from the sidechain maintainers, taking advantage of the proof-of-stake nature of the underlying protocol. This is achieved by having mainchain nodes maintain sufficient information about the sidechain that allows them to authenticate a small subset of sidechain stakeholders that is sufficient to reliably represent the view of a stakeholder majority on the sidechain. This piece of information is updated in regular intervals to account for stake shifting on the sidechain. Exploiting this, each withdrawal transaction from the sidechain to the mainchain is signed by this small subset of sidechain stakeholders. To minimise overheads we batch this authentication information and all the withdrawal transactions from the sidechain in a single message that will be prepared once per "epoch." We will refer to this signaling as *cross-chain certification*.

In greater detail, adopting some terminology from [89] (see Chapter 2), the sidechain certificate is constructed by obtaining signatures from the set of so-called *slot leaders* of the last $\Theta(k)$ slots of the previous epoch, where k is the security parameter. Subsequently, these signatures will be combined together with all necessary information to convince the mainchain nodes (that do not have access to the sidechain) that the sidechain certificate is valid.

After treating stake-based sidechains, we turn to work-based sidechains in later sections.

7.1.1 Defining Security of Pegged Ledgers

We consider a setting where a set of parties run a protocol maintaining n ledgers $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_n$, each of the ledgers potentially carrying many different assets. (This protocol might of course be a combination of subprotocols for each of the ledgers.)

For each $i \in [n]$, we denote by $\mathbb{A}_i$ the security assumption required by $\mathbf{L}_i$: For example, $\mathbb{A}_i$ may denote that there has never been a majority of hashing power (or stake in a particular asset, on this ledger or elsewhere) under the control of the adversary; that a particular entity (in case of a centralized ledger) was not corrupted; and so on. We assume that all $\mathbb{A}_i$ are *monotone* in the sense that once violated, they cannot become true again. Formally, $\mathbb{A}_i$ is a sequence of events $\mathbb{A}_i[t]$ for each time slot t that satisfy $\neg\mathbb{A}_i[t] \Rightarrow \neg\mathbb{A}_i[t+1]$ for all t.

There is an a priori unlimited number of (types of) assets, each asset representing e.g. a different cryptocurrency. For simplicity we assume that assets of the same type are fungible, but our treatment easily covers also non-fungible assets. We will allow specific rules of behavior for each asset (called *validity languages*), and each asset behaves according to these rules on each of the ledgers where it is present.

We will fix an operator $\mathsf{merge}(\cdot)$ that merges a set of ledger states $\mathcal{L} = \{\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n\}$ into a single ledger state denoted by $\mathsf{merge}(\mathcal{L})$. We will discuss concrete instantiations of $\mathsf{merge}(\cdot)$ later, for now simply assume that some canonical way of merging all ledger states into one is given.

Informally, at any point t during the execution, our security definition only provides guarantees to the subset $\mathcal{S}$ of ledgers that have their security assumptions $\mathbb{A}_i[t]$ satisfied (and hence are all considered uncorrupted). We require that:

- each ledger in $\mathcal{S}$ individually maintains both persistence and liveness;
- for each asset A, when looking at the sequence of all A-transactions σ that occurred on the ledgers in $\mathcal{S}$ (sequentialized via the merge operator), there must exist a hypothetical sequence of A-transactions τ that could have happened

on the compromised ledgers, such that the merge of σ and τ would be valid according to the validity language of A.

We now proceed to formalize the above intuition.

Definition 72 (Assets, syntactically valid transactions). *For an asset* A*, we denote by* $\mathcal{T}_\mathsf{A}$ *the* valid transaction set of A*, i.e., the set of all syntactically valid transactions involving* A*. For a ledger* $\mathbf{L}$ *we denote by* $\mathcal{T}_\mathbf{L}$ *the set of transactions that can be included into* $\mathbf{L}$*. For notational convenience, we define* $\mathcal{T}_{\mathsf{A},\mathbf{L}} \triangleq \mathcal{T}_\mathsf{A} \cap \mathcal{T}_\mathbf{L}$*. Let* $\mathsf{Assets}(\mathbf{L})$ *denote the set of all assets that are supported by* $\mathbf{L}$*. Formally,* $\mathsf{Assets}(\mathbf{L}) \triangleq \{\mathsf{A} : \mathcal{T}_{\mathsf{A},\mathbf{L}} \neq \emptyset\}$.

We assume that each transaction pertains to a particular asset and belongs to a particular ledger, i.e., for distinct $\mathsf{A}_1 \neq \mathsf{A}_2$ and $\mathbf{L}_1 \neq \mathbf{L}_2$, we have that $\mathcal{T}_{\mathsf{A}_1} \cap \mathcal{T}_{\mathsf{A}_2} = \emptyset$ and $\mathcal{T}_{\mathbf{L}_1} \cap \mathcal{T}_{\mathbf{L}_2} = \emptyset$. However, our treatment can be easily generalized to alleviate this restriction.

Definition 73 (Asset validity language). *For an asset* A *we denote* $\mathbb{V}_\mathsf{A} \subseteq \mathcal{T}_\mathsf{A}^*$ *the* validity language pertaining to the asset.

We will be interested only in validity languages that are generated by extension (see Definition 15) and which have transaction uniqueness (see Definition 16).

The following definition allows us to focus on a particular asset or ledger within a sequence of transactions.

Definition 74 (Ledger state projection). *Given a ledger state* L*, we call a* projection of L with respect to a set $\mathcal{X}$ *(and denote by* $\pi_\mathcal{X}(\mathsf{L})$*) the ledger state that is obtained from* L *by removing all transactions not in* $\mathcal{X}$*. To simplify notation, we will use* π_A *and* $\pi_\mathcal{I}$ *as a shorthand for* $\pi_{\mathcal{T}_\mathsf{A}}$ *and* $\pi_{\bigcup_{i\in\mathcal{I}} \mathcal{T}_{\mathbf{L}_i}}$*, denoting the projection of the transactions of a ledger state with respect to particular asset* A *or a particular set of individual ledger indices. Naturally, for a language* $\mathbb{V}$ *we define the* projected language $\pi_\mathcal{X}(\mathbb{V}) := \{\pi_\mathcal{X}(w) : w \in \mathbb{V}\}$*, which contains all the sequences of transactions from the original language, each of them projected with respect to* $\mathcal{X}$*.*

The concept of *effect transactions* below captures ledger interoperability at the syntactic level.

Definition 75 (Effect Transactions). *For two ledgers* $\mathbf{L}$ *and* $\mathbf{L}'$*, the* effect mapping *is a mapping of the form* $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'} : \mathcal{T}_\mathbf{L} \to (\mathcal{T}_{\mathbf{L}'} \cup \{\perp\})$*. A transaction* $\mathsf{tx}' = \mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \neq \perp$ *is called the* effect transaction *of the transaction* tx*.*

Intuitively, for any transaction $\mathsf{tx} \in \mathcal{T}_\mathbf{L}$, let $\mathsf{tx}' = \mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(tx)$ be the corresponding transaction. The transaction $\mathsf{tx}' \in \mathcal{T}_{\mathbf{L}'} \cup \{\perp\}$ identifies the necessary effect on ledger $\mathbf{L}'$ of the event of the inclusion of the transaction tx into the ledger $\mathbf{L}$. With foresight, in an implementation of a system of ledgers where a "pegging" exists, the transaction $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx})$ has to be eventually valid and includable in $\mathbf{L}'$ in response to the inclusion of tx in $\mathbf{L}$. Additionally, we assume that an effect transaction is always clearly identifiable as such, and its corresponding "sending" transaction can be derived from it; our instantiation does have this property.

We use a special symbol $\perp$ to indicate that the transaction tx does not necessitate any action on $\mathbf{L}'$ (this will be the case for most transactions). We will now be interested mostly in transactions that *do* require an action on the other ledger.

Definition 76 (Cross-Ledger Transfers)**.** *For two ledgers* $\mathbf{L}$ *and* $\mathbf{L}'$ *and an effect mapping* $\mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\cdot)$*, we refer to a transaction in* $\mathcal{T}_{\mathbf{L}}$ *that requires some effect on* $\mathbf{L}'$ *as a* $(\mathbf{L},\mathbf{L}')$-cross-ledger transfer transaction *(or* cross-ledger transfer *for short). The set of all cross-ledger transfers is denoted by* $\mathcal{T}^{\mathsf{cl}}_{\mathbf{L},\mathbf{L}'} \subseteq \mathcal{T}_{\mathbf{L}}$*, formally* $\mathcal{T}^{\mathsf{cl}}_{\mathbf{L},\mathbf{L}'} \triangleq \{\mathsf{tx} \in \mathcal{T}_{\mathbf{L}} : \mathsf{effect}_{\mathbf{L}\to\mathbf{L}'}(\mathsf{tx}) \neq \bot\}$*.*

Given ledger states $\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n$, we need to consider a joint ordered view of the transactions in all these ledgers. This is provided by the merge operator. Intuitively, merge allows us to create a combined view of multiple ledgers, putting all of the transactions across multiple ledgers into a linear ordering. We expect that even if certain ledgers are missing from its input, merge is still able to produce a global ordering for the remaining ledgers. With foresight, this ability of the merge operator will enable us to reason about the situation when some ledgers fail: In that case, the respective inputs to the merge function will be missing. The merge function definition below depends on the effect mappings, we keep this dependence implicit for simpler notation.

Definition 77 (Merging ledger states)**.** *The* $\mathsf{merge}(\cdot)$ *function is any mapping taking a subset of ledger states* $\mathcal{L} \subseteq \{\mathsf{L}_1, \mathsf{L}_2, \ldots, \mathsf{L}_n\}$ *and producing a ledger state* $\mathsf{merge}(\mathcal{L})$ *such that:*

1. ***Partitioning.*** *The ledger states in* $\mathcal{L}$ *are disjoint subsequences of* $\mathsf{merge}(\mathcal{L})$ *that cover the whole sequence* $\mathsf{merge}(\mathcal{L})$*.*
2. ***Topological soundness.*** *For any* $i \neq j$ *such that* $\mathsf{L}_i, \mathsf{L}_j \in \mathcal{L}$ *and any two transactions* $\mathsf{tx} \in \mathsf{L}_i$ *and* $\mathsf{tx}' \in \mathsf{L}_j$*, if* $\mathsf{tx}' = \mathsf{effect}_{\mathbf{L}_i\to\mathbf{L}_j}(\mathsf{tx})$ *then* tx *precedes* tx' *in* $\mathsf{merge}(\mathcal{L})$*.*

We will require that our validity languages are *correct* in the following sense.

Definition 78 (Correctness of $\mathbb{V}_{\mathsf{A}}$)**.** *A validity language* $\mathbb{V}_{\mathsf{A}}$ *is* correct *with respect to a mapping* $\mathsf{merge}(\cdot)$*, if for any ledger states* $\mathcal{L} \triangleq (\mathsf{L}_1, \ldots, \mathsf{L}_n)$ *such that* $\pi_{\mathsf{A}}(\mathsf{merge}(\mathcal{L})) \in \mathbb{V}_{\mathsf{A}}$*, indices* $i \neq j$*, and any cross-ledger transfer* $\mathsf{tx} \in \mathsf{L}_i \cap \mathcal{T}^{\mathsf{cl}}_{\mathbf{L}_i,\mathbf{L}_j}$ *such that* $\mathsf{effect}_{\mathbf{L}_i\to\mathbf{L}_j}(\mathsf{tx}) = \mathsf{tx}' \neq \bot$ *is not in* L_j*, we have*

$$\pi_{\mathsf{A}}(\mathsf{merge}(\mathsf{L}_1, \ldots, \mathsf{L}_i, \ldots, \mathsf{L}_j \,\|\, \mathsf{tx}', \ldots, \mathsf{L}_n)) \in \mathbb{V}_{\mathsf{A}} \; .$$

The above definition makes sure that if a cross-ledger transfer of an asset A is included into some ledger $\mathbf{L}_i$ and mandates an effect transaction on $\mathbf{L}_j$, then the inclusion of this effect transaction will be consistent with $\mathbb{V}_{\mathsf{A}}$. Note that this does not yet guarantee that the effect transaction will indeed be included into $\mathbf{L}_j$, this will be provided by the liveness of $\mathbf{L}_j$ required below.

We are now ready to give our main security definition. In what follows, we call a *system-of-ledgers protocol* any protocol run by a (possibly dynamically changing) set of parties that maintains an evolving state of n ledgers $\{\mathbf{L}_i\}_{i\in[n]}$.

Definition 79 (Pegging security)**.** *A system-of-ledgers protocol* Π *for* $\{\mathbf{L}_i\}_{i\in[n]}$ *is* pegging-secure *with liveness parameter* $u \in \mathbb{N}$ *with respect to:*

- *a set of assumptions* $\mathbb{A}_i$ *for ledgers* $\{\mathbf{L}_i\}_{i\in[n]}$*,*
- *a merge mapping* $\mathsf{merge}(\cdot)$*,*
- *validity languages* $\mathbb{V}_{\mathsf{A}}$ *for each* $\mathsf{A} \in \bigcup_{i\in[n]} \mathsf{Assets}(\mathbf{L}_i)$*,*

if for all PPT adversaries, all slots t and for $\mathcal{S}_t \triangleq \{i : \mathbb{A}_i[t] \text{ holds}\}$ we have that except with negligible probability in the security parameter:

Ledger persistence: *For each $i \in \mathcal{S}_t$, $\mathbf{L}_i$ satisfies the persistence property.*

Ledger liveness: *For each $i \in \mathcal{S}_t$, $\mathbf{L}_i$ satisfies the liveness property parametrized by u.*

Firewall: *For all $\mathsf{A} \in \bigcup_{i \in \mathcal{S}_t} \mathsf{Assets}(\mathbf{L}_i)$,*

$$\pi_{\mathsf{A}}\left(\mathsf{merge}\left(\{\mathbf{L}_i^{\cup}[t] \; : \; i \in \mathcal{S}_t\}\right)\right) \in \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathsf{A}}) \; .$$

Intuitively, the firewall property above gives the following guarantee: If the security assumption of a particular sidechain has been violated, we demand that the sequence of transactions σ that appears in the still uncompromised ledgers is a valid projection of some word from the asset validity language onto these ledgers. This means that there exists a sequence of transactions τ that *could have happened* on the compromised ledgers, such that it would "justify" the current state of the uncompromised ledgers as a valid state. Of course, we don't know whether this sequence τ actually occurred on the compromised ledger, however, given that this ledger itself no longer provides any reliable state, this is the best guarantee we can still offer to the uncompromised ledgers.

Looking ahead, when we define a particular validity language for our concrete, fungible, constant-supply asset, we will see that this property will translate into the mainchain maintaining "limited liability" towards the sidechain: the amount of money transferred back from the sidechain can never exceed the amount of money that was previously moved towards the sidechain, because no plausible history of sidechain transactions can exist that would justify such a transfer.

7.1.2 Implementing Pegged Ledgers

We present a construction for pegged ledgers that is based on Ouroboros PoS [89], but also applicable to other PoS systems such as Snow White [22] and Algorand [113]. Our protocol will implement a system of ledgers with pegging security according to Definition 79 under an assumption on the relative stake power of the adversary that will be detailed below.

The main challenge in implementing pegged ledgers is to facilitate secure cross-chain transfers. We consider two approaches to such transfers and refer to them as *direct observation* or *cross-chain certification.* Consider two pegged ledgers $\mathbf{L}_1$ and $\mathbf{L}_2$. Direct observation of $\mathbf{L}_1$ means that every node of $\mathbf{L}_2$ follows and validates $\mathbf{L}_1$; it is easy to see that this enables transfers from $\mathbf{L}_1$ to $\mathbf{L}_2$. On the other hand, cross-chain certification of $\mathbf{L}_2$ means that $\mathbf{L}_1$ contains appropriate cryptographic information sufficient to validate data issued by the nodes following $\mathbf{L}_2$. This allows transfers of assets from $\mathbf{L}_2$, as long as they are certified, to be accepted by $\mathbf{L}_1$-nodes without following $\mathbf{L}_2$. The choice between direct observation and cross-chain certification can be made independently for each direction of transfers between $\mathbf{L}_1$ and $\mathbf{L}_2$, any of the 4 variants is possible (cf. Figure 7.1).

Another aspect of implementing pegged ledgers in the PoS context is the choice of stake distribution that underlies the PoS on each of the chains. We again consider two options, which we call *independent staking* and *merged staking.* In independent staking, blocks on say $\mathbf{L}_1$ are "produced by" coins from $\mathbf{L}_1$ (in other words, the

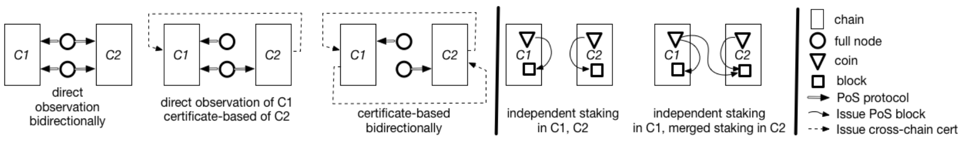

C1
C2
direct observation bidirectionally
direct observation of C1 certificate-based of C2
certificate-based bidirectionally
independent staking in C1, C2
independent staking in C1, merged staking in C2
chain
full node
coin
block
PoS protocol
Issue PoS block
Issue cross-chain cert

Figure 7.1: Deployment options for PoS Sidechains.

block-creating rights on $\mathbf{L}_1$ are attributed based on the stake distribution recorded on $\mathbf{L}_1$ only). In contrast, with merged staking, blocks on $\mathbf{L}_1$ are produced either by coins on $\mathbf{L}_1$, or coins on $\mathbf{L}_2$ that have, via their staking key, declared support of $\mathbf{L}_1$ (but otherwise remain on $\mathbf{L}_1$); see Figure 7.1. Also here, all 4 combinations are possible.

In our construction we choose an exemplary configuration between two ledgers $\mathbf{L}_1$ and $\mathbf{L}_2$, so that direct observation is applied to $\mathbf{L}_1$, cross-chain certification to $\mathbf{L}_2$, independent-staking in $\mathbf{L}_1$ and merged staking in $\mathbf{L}_2$. As a result, all stakeholders in $\mathbf{L}_2$ also keep track of chain development on $\mathbf{L}_1$ (and hence run a full node for $\mathbf{L}_1$) while the opposite is not necessary, i.e., $\mathbf{L}_1$ stakeholders can be oblivious of transactions and blocks being added to $\mathbf{L}_2$. This illustrates the two basic possibilities of pegging and can be easily adapted to any other of the configurations between two ledgers in Figure 7.1.

In order to reflect the asymmetry between the two chains in our exemplary construction we will refer to $\mathbf{L}_1$ as the "mainchain" **MC**, and to $\mathbf{L}_2$ as the "sidechain" **SC**. To elaborate further on this concrete asymmetric use case, we also fully specify how the sidechain can be initialized from scratch, assuming that the mainchain already exists.

The pegging with the sidechain will be provided with respect to a specific asset of **MC** that will be created on **MC**. Note that **MC** as well as **SC** may carry additional assets but for simplicity we will assume that staking and pegging is accomplished only via this single primary asset.

The presentation of the construction is organized as follows. In Section 7.1.2 we use the ATMS primitive (introduced in Chapter 6) as a black box to build secure pegged ledgers with respect to concrete instantiations of the functions merge and effect and a validity language $\mathbb{V}_{\mathfrak{A}}$ for asset $\mathfrak{A}$ given in Section 7.1.2.

A Concrete Asset $\mathfrak{A}$

We now present an example of a simple fungible asset with fixed supply, which we denote $\mathfrak{A}$, and describe its validity language $\mathbb{V}_{\mathfrak{A}}$. This will be the asset (and validity language) considered in our construction and proof. While $\mathbb{V}_{\mathfrak{A}}$ is simple and natural, it allows us to exhibit the main features of our security treatment and illustrate how it can be applied to more complex languages such as those capable of capturing smart contracts; we omit such extensions in this version. Note that our language is account-based, but a UTXO-based validity language can be considered in a similar manner.

Instantiating $\mathbb{V}_{\mathsf{A}}$

The validity language $\mathbb{V}_{\mathfrak{A}}$ for the asset $\mathfrak{A}$ considers two ledgers: the mainchain ledger $\mathbf{L}_0 \triangleq \mathbf{MC}$ and the sidechain ledger $\mathbf{L}_1 \triangleq \mathbf{SC}$. For this asset, every transaction $\mathsf{tx} \in \mathcal{T}_{\mathfrak{A}}$ has the form $\mathsf{tx} = (\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma)$, where:

- txid is a transaction identifier that prevents replay attacks. We assume that txid contains sufficient information to identify lid by inspection and that this is part of syntactic transaction validation.
- $\mathsf{lid} \in \{0, 1\}$ is the ledger index where the transaction belongs.
- $\mathsf{send} \in \{0, 1\}$ is the index of the sender ledger $\mathbf{L}_{\mathsf{send}}$ and sAcc is an account on this ledger, this is the sender account. For simplicity, we assume that sAcc is the public key of the account.
- $\mathsf{rec} \in \{0, 1\}$ is the index of the recipient ledger $\mathbf{L}_{\mathsf{rec}}$ and rAcc is an account (again represented by a public key) on this ledger, this is the recipient account. We allow either $\mathbf{L}_{\mathsf{send}} = \mathbf{L}_{\mathsf{rec}}$, which denotes a *local transaction*, or $\mathbf{L}_{\mathsf{send}} \neq \mathbf{L}_{\mathsf{rec}}$, which denotes a *remote transaction* (i.e., a cross-ledger transfer).
- v is the amount to be transferred.
- σ is the signature of the sender, i.e. made with the private key corresponding to the public key sAcc on the plaintext $(\mathsf{txid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v)$.

The correctness of lid is enforced by the ledgers, i.e., for both $i \in \{0, 1\}$ the set $\mathcal{T}_{\mathfrak{A}, \mathbf{L}_i}$ only contains transactions with $\mathsf{lid} = i$. Note that although we sometimes notationally distinguish between an account and the public key that is associated with it, for simplicity we will assume that these are either identical or can always be derived from one another (this assumption is not essential for our construction).

The membership-deciding algorithm for $\mathbb{V}_{\mathfrak{A}}$ is presented in Algorithm 40. It processes the sequence of transactions $(\mathsf{tx}_1, \mathsf{tx}_2, \ldots, \mathsf{tx}_m)$ given to it as input in their order. Assuming transactions are syntactically valid, the function verifies for each transaction tx_i the freshness of txid, validity of the signature, and availability of sufficient funds on the sending account. For an intra-ledger transaction (i.e., one that has $\mathsf{send} = \mathsf{rec}$), these are all the performed checks.

More interestingly, $\mathbb{V}_{\mathfrak{A}}$ also allows for cross-ledger transfers. Such transfers are expressed by a pair of transactions in which $\mathsf{send} \neq \mathsf{rec}$. The first transaction appears in $\mathsf{lid} = \mathsf{send}$, while the second transaction appears in $\mathsf{lid} = \mathsf{rec}$. The two transactions are identical except for this change in lid (this is the only exception to the txid-freshness requirement). Every receiving transaction has to be preceded by a matching sending transaction. Cross-chain transactions have to, similarly to intra-ledger transactions, conform to laws of balance conservation.

Note that $\mathbb{V}_{\mathfrak{A}}$ does not require that every "sending" cross-ledger transaction on the sender ledger is matched by a "receiving" transaction on the receiving ledger. Hence, if the asset $\mathfrak{A}$ is sent from ledger $\mathbf{L}_{\mathsf{send}}$ but has not yet arrived on $\mathbf{L}_{\mathsf{rec}}$ then validity for this asset is *not* violated. All the validity language ensures is that appending the sidechain_receive transaction to the rec will eventually be a valid way to extend the receiving ledger, as long as the sidechain_send transaction has been included in send.

Algorithm 40 The transaction sequence validator (membership-deciding algorithm for $\mathbb{V}_{\mathfrak{A}}$).

```
function valid-seq($\vec{tx}$)
    balance $\leftarrow$ Initial stake distribution
    seen $\leftarrow \emptyset$
    $\triangleright$ Traverse transactions in order
    for tx $\in \vec{tx}$ do
        $\triangleright$ Destructure tx into its constituents
        $(\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma) \leftarrow \mathsf{tx}$
        if $\neg\mathrm{valid}(\sigma)$ then
            return false
        end if
        if lid = send then
            $\triangleright$ Replay protection
            if seen[txid] $\neq 0$ then
                return false
            end if
            $\triangleright$ Law of conservation
            if balance[send][sAcc] $- v < 0$ then
                return false
            end if
        else
            $\triangleright$ The case lid = rec $\neq$ send
            if seen[txid] $\neq 1$ then
                return false
            end if
            $\triangleright$ Cross-ledger validity
            $\mathsf{tx}' \leftarrow \mathsf{effect}^{-1}_{\mathbf{L}_{(1-\mathsf{lid})} \to \mathbf{L}_{\mathsf{lid}}}(\mathsf{tx})$
            if tx′ has not appeared before then
                return false
            end if
        end if
        if seen[txid] $= 0$ then
            $\triangleright$ Update sender balance when money departs
            balance[send][sAcc] $-= v$
        end if
        $\triangleright$ Update receiver balance when money arrives
        if (seen[txid] $= 0 \wedge$ send = rec) $\vee$ (seen[txid] $= 1 \wedge$ send $\neq$ rec) then
            balance[rec][rAcc] $+= v$
        end if
        seen[txid]$+ = 1$
    end for
    return true
end function
```

Instantiating $\mathsf{effect}_{\mathbf{L}_i \to \mathbf{L}_j}$

For the simple asset $\mathfrak{A}$ outlined above, every cross-ledger transfer is a "sending" transaction tx with $\mathbf{L}_{\mathsf{lid}} = \mathbf{L}_{\mathsf{send}} \neq \mathbf{L}_{\mathsf{rec}}$ appearing in $\mathbf{L}_{\mathsf{send}}$, and its effect transaction is a "receiving" transaction tx' with $\mathbf{L}_{\mathsf{lid}} = \mathbf{L}_{\mathsf{rec}} \neq \mathbf{L}_{\mathsf{send}}$ in $\mathbf{L}_{\mathsf{rec}}$ that is otherwise identical (except for the different $\mathsf{lid}' = 1 - \mathsf{lid}$). Hence, we define $\mathsf{effect}_{\mathbf{L}_{\mathsf{send}} \to \mathbf{L}_{\mathsf{rec}}}(\mathsf{tx}) = \mathsf{tx}'$ exactly for all these transactions and no other.

Instantiating $\mathsf{merge}(\cdot)$

It is easy to construct a canonical function $\mathsf{merge}(\cdot)$ once we see its inputs not only as ledger states (i.e., sequences of transactions) but we also exploit the additional structure of the blockchains carrying those ledgers. The *canonical merge* of the set of ledger states $\mathcal{L}$ is the lexicographically minimum topologically sound merge, in which transactions of ledger L_i are compared favourably to transactions in L_j if $i < j$. However, note that the construction we provide below will work for any topologically sound merge function.

One can easily observe the following statement.

Proposition 61. *The validity language $\mathbb{V}_{\mathfrak{A}}$ is correct (according to Definition 78) with respect to the* merge *function defined above.*

The Sidechain Construction

We now describe the procedures for running a sidechain in the configuration outlined at the beginning of this section: with independent staking on **MC** and merged staking on **SC**; direct observation of **MC** and cross-chain certification of **SC**. We describe the sidechain's creation, maintenance, and the way assets can be transferred to it and back. The protocol we describe below is quite complex, we hence choose to describe different parts of the protocol in differing levels of detail. This level is always chosen with the intention to allow the reader to easily fill in the details. A graphical depiction of our construction that can serve as a reference is given in Figure 7.2.

Notation Where applicable, we denote the analogues of the mainchain objects on the sidechain with an additional overline. In our pseudocode, we use the statement "**post** tx **to** $\mathbf{L}$" to refer to the action of broadcasting the transaction tx to the maintainers of the ledger $\mathbf{L}$ so that they include it in the ledger eventually as prescribed by the protocol. Unless indicated otherwise, we also denote by MC (resp. SC) the current *ledger state* of the ledger **MC** (resp. **SC**) as viewed by the party executing the protocol. Similarly, we denote by $\mathsf{C}_{\mathbf{MC}}$ (resp. $\mathsf{C}_{\mathbf{SC}}$) the currently held *chain* corresponding to the ledger **MC** (resp. **SC**). Hence, for example MC always represents the state stored in the stable part of the chain $\mathsf{C}_{\mathbf{MC}}$.

Helper Transactions and Data The construction uses a set of *helper transactions* which can be included in both blockchains, but do not get reported in the respective ledgers. These helper transactions store the appropriate metadata which is implementation-specific and allow the pegging functionality to be maintained. The transaction types sidechain_support, sidechain_certificate, sidechain_success

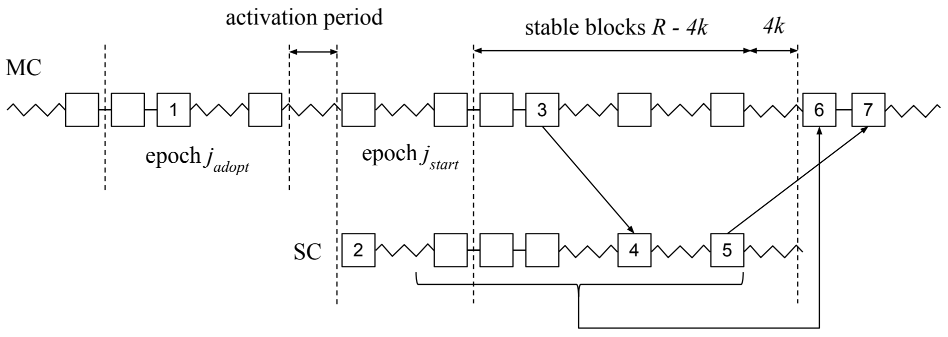


Figure 7.2: Our sidechain construction. Blocks are shown as rectangles. Adjacent blocks connect with straight lines. Squiggly lines indicate some blocks are omitted. **MC** is at the top, **SC** at the bottom. Epochs are separated by dashed lines. $e_{j_{\text{adopt}}}$ is the epoch of first signalling; $e_{j_{\text{start}}}$ is the activation epoch. Blocks of interest: 1. The first block signalling **SC** awareness; 2. The **SC** genesis block; 3. A $\mathsf{tx}_{\mathsf{send}}$ transaction for a deposit; 4. A $\mathsf{tx}_{\mathsf{rec}}$ transaction for a deposit; 5. A $\mathsf{tx}_{\mathsf{send}}$ transaction for withdrawal; 6. A sc_cert transaction signalling trust transition within **SC** and certifying pending withdrawals; 7. A $\mathsf{tx}_{\mathsf{rec}}$ transaction for withdrawal, certified in a sc_cert transaction e.g. in block 6.

and sidechain_failure, whose nature will be detailed later, are of this kind. Moreover, our concrete implementation of pegged ledgers extends certain transactions with additional information (such as Merkle-tree inclusion proofs) that are, for convenience, understood to be stripped off these transactions when the blockchain is interpreted as a ledger.

Initialisation The creation of a new sidechain **SC** starts by any of the stakeholders of the mainchain adopting the code that implements the sidechain. This action does not require the stakeholders to put stake on the sidechain but merely to run the code to support it (e.g. by installing a pluggable module into their client software). In the following this is referred to as "adopting the sidechain" and captured by the predicate SidechainAdoption. The adoption is announced at the mainchain by a special transaction detailed below. Each sidechain is identified by a unique identifier $\mathsf{id}_{\mathbf{SC}}$.

Let j_{adopt} denote the epoch on **MC** when the first adoption transaction has appeared; the sidechain **SC** – if its activation succeeds as discussed below – will start at the beginning of some later epoch j_{start} and will have its slots and epochs synchronized with **MC**. The software module implementing the sidechain comes with a set of deterministic rules describing the requirements for the successful activation of the sidechain, as well as for determining j_{start}. These rules are sidechain-specific and are captured in a predicate ActivationSuccess and a function ActivationEpoch, respectively. One typical such example is the following: the sidechain starts at the beginning of **MC**-epoch j_{start} for the smallest j_{start} that satisfies: (i) $j_{\mathsf{start}} - j_{\mathsf{adopt}} > c_1$; (ii) at least c_2-fraction of stake on **MC** is controlled by stakeholders that have adopted **SC**; for some constants c_1,c_2. Additionally, if such a successful activation does not occur until a failure condition captured by a predicate $\mathsf{ActivationFailure}$ is met (e.g. until a predetermined period of $c_3 > c_1$ epochs has passed), the sidechain initialization is aborted.

The activation process then follows the steps outlined below, the detailed description is given in Algorithm 41).

Algorithm 41 Sidechain initialisation procedures.

The algorithm is run by every stakeholder U that adopted the sidechain. We denote by (vk, sk) its public and private keys.

```
 1: upon $\mathsf{SidechainAdoption}(\mathsf{id}_{\mathbf{SC}})$ do
 2:     $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{initializing}$
 3:     $(vk', sk') \leftarrow \mathsf{Gen}(\mathcal{P})$
 4:     $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{sidechain\_support}(\mathsf{id}_{\mathbf{SC}}, vk, vk'))$
 5:     post $\mathsf{sidechain\_support}(\mathsf{id}_{\mathbf{SC}}, vk, vk', \sigma)$ to $\mathbf{MC}$
 6: end upon
 7: upon $\mathbf{MC}.\mathsf{NewEpoch}()$ do
 8:     $j \leftarrow \mathbf{MC}.\mathsf{EpochIndex}()$
 9:     if $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] = \mathsf{initializing}$ then
10:         if $\mathsf{ActivationFailure}()$ then
11:             $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{failed}$
12:             post $\mathsf{sidechain\_failure}(\mathsf{id}_{\mathbf{SC}})$ to $\mathbf{MC}$
13:         else if $\mathsf{ActivationSuccess}()$ then
14:             $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] \leftarrow \mathsf{initialized}$
15:             $j_{\mathsf{start}} \leftarrow \mathsf{ActivationEpoch}()$
16:             Post $\mathsf{sidechain\_success}(\mathsf{id}_{\mathbf{SC}})$ to $\mathbf{MC}$
17:         end if
18:     end if
19:     if $\mathsf{sidechain\_state}[\mathsf{id}_{\mathbf{SC}}] = \mathsf{initialized} \wedge j = j_{\mathsf{start}}$ then
20:         $\overline{\eta}_{j_{\mathsf{start}}} \leftarrow H(\mathsf{id}_{\mathbf{SC}}, \eta_{j_{\mathsf{start}}})$
21:         $\mathcal{VK}_{j_{\mathsf{start}}} \leftarrow 2k$ last slot leaders of $e_{j_{\mathsf{start}}}$ in $\mathbf{SC}$
22:         $avk^{j_{\mathsf{start}}} \leftarrow \mathsf{AKey}(\mathcal{VK}_{j_{\mathsf{start}}})$
23:         $\overline{\mathsf{G}} \leftarrow \left(\mathsf{id}_{\mathbf{SC}}, \overline{\mathsf{SD}}_{j_{\mathsf{start}}}, \overline{\eta}_{j_{\mathsf{start}}}, \mathcal{P}, avk^{j_{\mathsf{start}}}\right)$
24:         $\mathsf{C}_{\mathbf{SC}} \leftarrow (\overline{\mathsf{G}})$
25:     end if
26: end upon
```

First, every stakeholder U_i of $\mathbf{MC}$ (holding a key pair (vk, sk)) that supports the sidechain posts a special transaction $\mathsf{sidechain\_support}(\mathsf{id}_{\mathbf{SC}}, vk, vk')$, signed by sk into the mainchain. Here vk' is a public key from an ATMS key pair freshly generated by U_i; its role is explained in Section 7.1.2 below.

If the sidechain activation succeeds, then during the first slot of epoch j_{start} the stakeholders of $\mathbf{MC}$ that support $\mathbf{SC}$ construct the genesis block $\overline{\mathsf{G}} = (\mathsf{id}_{\mathbf{SC}}, \overline{\mathsf{SD}}_{j_{\mathsf{start}}}, \overline{\eta}_{j_{\mathsf{start}}} \triangleq H(\mathsf{id}_{\mathbf{SC}}, \eta_{j_{\mathsf{start}}}), \mathcal{P}, avk^{j_{\mathsf{start}}})$ for $\mathbf{SC}$. $\eta_{j_{\mathsf{start}}}$ is the randomness for leader election on $\mathbf{MC}$ in epoch j_{start} (derived on $\mathbf{MC}$ in epoch $j_{\mathsf{start}} - 1$). It is reused to compute the initial sidechain randomness $\overline{\eta}_{j_{\mathsf{start}}}$ as well, further $\overline{\eta}_{j'}$ for $j' > j_{\mathsf{start}}$ are determined independently on $\mathbf{SC}$ using the Ouroboros coin-tossing protocol.[4]

[4]This can be interpreted as using $\mathbf{MC}$ to implement the setup functionality needed to bootstrap $\mathbf{SC}$.

Furthermore, $\mathcal{P}$ and $avk^{j_{\mathsf{start}}}$ are public parameters and an aggregated public key of an ATMS scheme; their creation and role is discussed in Section 7.1.2 below. Note that $\overline{\mathsf{G}}$ is defined mostly for notational compatibility, as $\overline{\mathsf{SD}}_{j_{\mathsf{start}}}$ is empty at this point anyway. $\overline{\mathsf{G}}$ can be constructed as soon as $\eta_{j_{\mathsf{start}}}$ is known and stable.

The stakeholders that adopted **SC** create a transaction $\mathsf{sidechain\_success}(\mathsf{id}_{\mathbf{SC}})$ and post it into **MC** to signify that **SC** has been initialized. If the sidechain creation expires, then, after the first block of the next epoch after expiration occurs, the stakeholders of **MC** that supported **SC** post the transaction $\mathsf{sidechain\_failure}(\mathsf{id}_{\mathbf{SC}})$ to **MC**. We assume that both predicates $\mathsf{ActivationSuccess}$ and $\mathsf{ActivationFailure}$ can be evaluated based on the state of **MC** only, and hence spurious success/failure transactions will be considered invalid.

Maintenance Once the sidechain is created, both the mainchain and the sidechain need to be maintained by their respective set of stakeholders (detailed below) running their respective instance of the Ouroboros protocol.

In the case of the mainchain, the maintenance procedure is given in Algorithm 42. This algorithm is run by all stakeholders controlling stake that is recorded on the mainchain. Each stakeholder, on every new slot, collects all the candidate **MC**-chains from the network (modelled via the $\mathsf{Diffuse}$ functionality) and filters them for both consensus-level validity (using $\mathbf{MC}.\mathsf{ValidateConsensusLevel}$) and transaction validity (using the $\mathrm{verifier}_{\mathbf{MC}}$ predicate given in Algorithms 44 and 43). Out of the remaining valid chains, he chooses his new state $\mathsf{C}_{\mathbf{MC}}$ via $\mathsf{PickWinningChain}$. Then the stakeholder evaluates whether he is an eligible leader for this slot, basing its selection on the stake distribution SD_j and randomness η_j, which are determined once per epoch in accordance with the Ouroboros protocol. If the stakeholder finds out he is a slot leader, he creates a new block B by including all transactions currently valid with respect to $\mathsf{C}_{\mathbf{MC}}$ (as per the predicate $\mathrm{verifytx}_{\mathbf{MC}}$ given also in Algorithm 43), appends it to the chain $\mathsf{C}_{\mathbf{MC}}$ and diffuses the result[5] for other parties to adopt.

The maintenance procedure for **SC** is similar, hence we only describe here how it differs from Algorithm 42. Most importantly, it is executed by all stakeholders who have adopted **SC**, irrespectively of whether they own any stake on **SC**. Recall that the slots and epochs of the **SC**-instance of Ouroboros are aligned with the slots and epochs of **MC**.

The first difference is that all ocurrences of **MC** and $\mathsf{C}_{\mathbf{MC}}$ are naturally replaced by **SC** and $\mathsf{C}_{\mathbf{SC}}$, respectively. This also means that the validity of received chains (resp. transactions), determined on line 13 (resp. 21), is decided based on predicate $\mathrm{verifier}_{\mathbf{SC}}(\cdot, \mathsf{C}_{\mathbf{MC}})$ (resp. $\mathrm{verifytx}_{\mathbf{SC}}(\cdot)$) instead of the predicate $\mathrm{verifier}_{\mathbf{MC}}(\cdot)$ (resp. $\mathrm{verifytx}_{\mathbf{MC}}(\cdot)$). Additionally, note that $\mathrm{verifytx}_{\mathbf{SC}}$ must be called with a sequence of transactions containing both the transactions in **SC** as well as the transactions in **MC** interspersed and timestamped, similarly to the way done in Line 2 of Algorithm 46. This is straightforward to implement, as the sidechain maintainers also directly observe the mainchain. The predicates $\mathrm{verifytx}_{\mathbf{SC}}$ and $\mathrm{verifier}_{\mathbf{SC}}$ are given in Algorithms 45 and 46, respectively.

[5]As in [89, 45], we simplify our presentation by diffusing the complete chains, although a practical implementation would only diffuse the block B.

Algorithm 42 Mainchain maintenance procedures.

The algorithm is run by every stakeholder U with stake on $\mathbf{MC}$ in every epoch $j \geq j_{\mathsf{start}}$, sk denotes the secret key of U. An analogous mainchain-maintaining procedure was running also before j_{start} and is omitted.

```
 1: upon $\mathbf{MC}.\mathsf{NewSlot}()$ do
 2:     $sl \leftarrow \mathbf{MC}.\mathsf{SlotIndex}()$
 3:     ▷ First slot of a new epoch
 4:     if $sl \mod R = 1$ then
 5:         $j \leftarrow \mathbf{MC}.\mathsf{EpochIndex}()$
 6:         $\mathsf{SD}_j \leftarrow \mathbf{MC}.\mathsf{GetDistr}(j)$
 7:         $\eta_j \leftarrow \mathbf{MC}.\mathsf{GetRandomness}(j)$
 8:     end if
 9:     $\mathcal{C} \leftarrow$ chains received via $\mathsf{Diffuse}$
10:     ▷ Consensus-level validation
11:     $\mathcal{C}_{\mathsf{valid}} \leftarrow \mathsf{Filter}(\mathcal{C}, \mathbf{MC}.\mathsf{ValidateConsensusLevel})$
12:     ▷ Transaction-level validation
13:     $\mathcal{C}_{\mathsf{validtx}} \leftarrow \mathsf{Filter}(\mathcal{C}_{\mathsf{valid}}, \mathrm{verifier}_{\mathbf{MC}}(\cdot))$
14:     ▷ Apply chain selection rule
15:     $\mathsf{C}_{\mathbf{MC}} \leftarrow \mathbf{MC}.\mathsf{PickWinningChain}(\mathsf{C}_{\mathbf{MC}}, \mathcal{C}_{\mathsf{validtx}})$
16:     ▷ Decide slot leadership based on $\mathsf{SD}_j$ and $\eta_j$
17:     if $\mathbf{MC}.\mathsf{SlotLeader}(U, j, sl, \mathsf{SD}_j, \eta_j)$ then
18:         $\mathsf{prev} \leftarrow H(\mathsf{C}_{\mathbf{MC}}[-1])$
19:         $\vec{\mathsf{tx}}_{\mathsf{state}} \leftarrow$ transaction sequence in $\mathsf{C}_{\mathbf{MC}}$
20:         $\vec{\mathsf{tx}} \leftarrow$ current transactions in mempool
21:         $\vec{\mathsf{tx}}_{\mathsf{valid}} \leftarrow \mathrm{verifytx}_{\mathbf{MC}}(\vec{\mathsf{tx}}_{\mathsf{state}} \parallel \vec{\mathsf{tx}})[|\vec{\mathsf{tx}}_{\mathsf{state}}| :]$
22:         $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{prev}, \vec{\mathsf{tx}}_{\mathsf{valid}})$
23:         $B \leftarrow (\mathsf{prev}, \vec{\mathsf{tx}}_{\mathsf{valid}}, \sigma)$
24:         $\mathsf{C}_{\mathbf{MC}} \leftarrow \mathsf{C}_{\mathbf{MC}} \parallel B$
25:         $\mathsf{Diffuse}(\mathsf{C}_{\mathbf{MC}})$
26:     end if
27: end upon
```

Algorithm 43 The **MC** transaction verifier.

```
1: function $\text{verifytx}_{\textbf{MC}}(\vec{\mathsf{tx}})$
2:     $\mathsf{bal} \leftarrow$ initial stake; $avk \leftarrow$ initial aggregate key
3:     $\mathsf{seen} \leftarrow \emptyset$; $\mathsf{pool} \leftarrow 0$; $\mathsf{pfs\_mtrs} \leftarrow \emptyset$; $\mathsf{pfs\_used} \leftarrow \emptyset$
4:     for $\mathsf{tx} \in \vec{\mathsf{tx}}$ do
5:         if $\mathsf{type}(\mathsf{tx}) = \mathsf{sc\_cert}$ then
6:             $(m, \sigma) \leftarrow \mathsf{tx}$
7:             if $\neg\mathsf{AVer}(m, avk, \sigma)$ then
8:                 continue
9:             end if
10:            $(\mathsf{txs\_root}, avk') \leftarrow m$; $avk \leftarrow avk'$; $\mathsf{pfs\_mtrs}[\mathsf{txs\_root}] \leftarrow \mathsf{true}$
11:        else
12:            $(\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma) \leftarrow \mathsf{tx}$
13:            $m \leftarrow (\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v)$
14:            if $\neg\text{Ver}(m, \mathsf{sAcc}, \sigma) \vee \mathsf{seen}[\mathsf{txid}] \neq 0$ then
15:                continue
16:            end if
17:            if $\mathsf{lid} = \mathsf{send}$ then
18:                if $\mathsf{bal}[\mathsf{sAcc}] - v < 0$ then
19:                    continue
20:                end if
21:                $\mathsf{bal}[\mathsf{sAcc}] \mathrel{-}= v$
22:            else if $\mathsf{send} \neq \mathsf{rec}$ then
23:                $\pi \leftarrow \mathsf{tx}.\pi$; $(mtr, \mathsf{inclusion\_pf}) \leftarrow \pi$
24:                if $\pi \in \mathsf{pfs\_used}$ then
25:                    continue
26:                end if
27:                if $mtr \notin \mathsf{pfs\_mtrs} \vee \neg\text{mtr-ver}(mtr, \mathsf{inclusion\_pf})$ then
28:                    continue
29:                end if
30:            end if
31:            if $\mathsf{lid} = \mathsf{rec}$ then
32:                $\mathsf{bal}[\mathsf{rAcc}] \mathrel{+}= v$
33:            end if
34:            if $\mathsf{send} \neq \mathsf{rec}$ then
35:                if $\mathsf{lid} = \mathsf{send}$ then
36:                    $\mathsf{pool} \mathrel{-}= v$
37:                else
38:                    $\mathsf{pool} \mathrel{+}= v$
39:                end if
40:            end if
41:        end if
42:        $\mathsf{seen} \leftarrow \mathsf{seen} \,\|\, \mathsf{tx}$
43:    end for
44:    return $\mathsf{seen}$
45: end function
```

Algorithm 44 The **MC** chain verifier.

```
function verifier$_{\mathbf{MC}}(\mathsf{C}_{\mathsf{mc}})$
    $\vec{\mathsf{tx}} \leftarrow \emptyset$
    for $B \in \mathsf{C}_{\mathsf{mc}}$ do
        for $\mathsf{tx} \in B$ do
            $\vec{\mathsf{tx}} \leftarrow \vec{\mathsf{tx}} \,\|\, \mathsf{tx}$
        end for
    end for
    return $\vec{\mathsf{tx}} \neq \mathrm{verifytx}_{\mathbf{MC}}(\vec{\mathsf{tx}})$
end function
```

Algorithm 45 The **SC** transaction verifier.

```
function verifytx$_{\mathbf{SC}}(\vec{\mathsf{tx}})$
    $\mathsf{bal}[\mathbf{MC}] \leftarrow$ Initial $\mathbf{MC}$ stake
    $\mathsf{bal}[\mathbf{SC}] \leftarrow$ Initial $\mathbf{SC}$ stake
    $\mathsf{mc\_outgoing\_tx} \leftarrow \emptyset$; $\mathsf{seen} \leftarrow \emptyset$
    for $\mathsf{tx} \in \vec{\mathsf{tx}}$ do
        $(\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v, \sigma, t) \leftarrow \mathsf{tx}$
        $m \leftarrow (\mathsf{txid}, \mathsf{lid}, (\mathsf{send}, \mathsf{sAcc}), (\mathsf{rec}, \mathsf{rAcc}), v)$
        if $\neg\mathrm{Ver}(\mathsf{sAcc}, m, \sigma) \lor \mathsf{seen}[\mathsf{txid}] \neq 0$ then
            continue
        end if
        if $\mathsf{lid} = \mathsf{send}$ then
            if $\mathsf{bal}[\mathsf{send}][\mathsf{sAcc}] - v < 0$ then
                continue
            end if
            if $\mathsf{lid} = \mathbf{MC} \land \mathsf{send} \neq \mathsf{rec}$ then
                $\mathsf{mc\_outgoing\_tx}[\mathsf{txid}] \leftarrow t + 2k$
            end if
        end if
        if $\mathsf{lid} = \mathsf{rec}$ then
            if $\mathsf{send} \neq \mathsf{rec}$ then
                ▷ *Effect pre-image tx immature*
                if $t < \mathsf{mc\_outgoing\_tx}[\mathsf{txid}]$ then
                    continue
                end if
            end if
            $\mathsf{bal}[\mathsf{rec}][\mathsf{rAcc}] \mathrel{+}= v$
        end if
        if $\mathsf{lid} = \mathsf{send}$ then
            $\mathsf{bal}[\mathsf{send}][\mathsf{sAcc}] \mathrel{-}= v$
        end if
        $\mathsf{seen} \leftarrow \mathsf{seen} \,\|\, \mathsf{tx}$
    end forreturn seen
end function
```

Algorithm 46 The **SC** verifier.

```
1: function verifier_SC($\mathsf{C}_{\mathsf{sc}}, \mathsf{C}_{\mathsf{mc}}$)
2:     $\vec{\mathsf{tx}} \leftarrow$ annotatetx_SC($\mathsf{C}_{\mathsf{sc}}, \mathsf{C}_{\mathsf{mc}}$) return $\vec{\mathsf{tx}} \neq$ verifytx_SC($\vec{\mathsf{tx}}$)
3: end function
```

Algorithm 47 The **SC** transaction annotation.

```
1: function annotatetx_SC($\mathsf{C}_{\mathsf{sc}}, \mathsf{C}_{\mathsf{mc}}$)
2:     $\vec{\mathsf{tx}} \leftarrow \emptyset$
3:     for each time slot $t$ do
4:         $\vec{\mathsf{tx}}' \leftarrow \epsilon$
5:         if $\mathsf{C}_{\mathsf{sc}}$ has a block generated at slot $t$ then
6:             $B \leftarrow$ the block in $\mathsf{C}_{\mathsf{sc}}$ generated at $t$
7:             for $\mathsf{tx} \in B$ do
8:                 $\vec{\mathsf{tx}}' \leftarrow \vec{\mathsf{tx}}' \,\|\, \mathsf{tx}$
9:             end for
10:        end if
11:        if $\mathsf{C}_{\mathsf{mc}}$ has a block generated at slot $t$ then
12:            $B \leftarrow$ the block in $\mathsf{C}_{\mathsf{mc}}$ generated at $t$
13:            for $\mathsf{tx} \in B$ do
14:                $\vec{\mathsf{tx}}' \leftarrow \vec{\mathsf{tx}}' \,\|\, \mathsf{tx}$
15:            end for
16:        end if
17:        for $\mathsf{tx} \in \vec{\mathsf{tx}}'$ do
18:            ▷ Mark the time of each tx in $\vec{\mathsf{tx}}'$
19:            $\mathsf{tx}.t \leftarrow t$
20:        end for
21:        $\vec{\mathsf{tx}} \leftarrow \vec{\mathsf{tx}} \,\|\, \vec{\mathsf{tx}}'$
22:    end forreturn $\vec{\mathsf{tx}}$
23: end function
```

Second, instead of the stake distribution SD_j determined on line 6, a different distribution $\overline{\mathsf{SD}}^*_j$ is determined to be used for slot leader selection in the j-th epoch of the sidechain. The distribution $\overline{\mathsf{SD}}^*$ contains all stake belonging to stakeholders that have adopted **SC**, irrespectively of whether this stake is located on **MC** or **SC** (we call such stake **SC**-*aware*). It can be obtained by combining the distribution $\overline{\mathsf{SD}}$ as recorded in **SC** with the distribution of **SC**-aware stake on **MC** (which is known to **SC**-maintainers via direct observation of **MC**). Note that the distribution used for epoch j reflects the stake distribution of **SC**-aware stake in the past, namely by slot $4k$ of epoch $j-1$, just as in **MC**. Naturally, this also implies that the fourth parameter for the SlotLeader predicate on line 17 is $\overline{\mathsf{SD}}^*_j$ instead of SD_j.

Finally, the block construction procedure on line 23 is adjusted so that in the last $2k$ slots of each epoch, the created blocks on the sidechain also contain an additional ATMS signature of a so-called sidechain certificate (how this certificate is constructed and used will be described below). Hence, whenever $sl \mod R > 10k$, line 23 is replaced by $B \leftarrow (\mathsf{prev}, \vec{\mathsf{tx}}_{\mathsf{valid}}, \sigma, \sigma_{\mathsf{sc\_cert}_{j+1}})$ where $\sigma_{\mathsf{sc\_cert}_{j+1}} = \mathsf{Sig}_{sk}(\mathsf{sc\_cert}_{j+1})$ and j is the current epoch index.

Depositing to SC Once $\mathbf{SC}$ is initialized, cross-chain transfers to it can be made from $\mathbf{MC}$. A cross-chain transfer operation in this case consists of two transactions $\mathsf{tx}_{\mathsf{send}}$ and $\mathsf{tx}_{\mathsf{rec}}$ that both have $\mathsf{send} = \mathbf{MC}$, $\mathsf{rec} = \mathbf{SC}$, and all other fields are also identical, except that each tx_i for $i \in \{\mathsf{send}, \mathsf{rec}\}$ contains $\mathsf{lid} = i$. The *sending transaction* $\mathsf{tx}_{\mathsf{send}}$ is meant to be included in $\mathbf{MC}$, while the *receiving transaction* $\mathsf{tx}_{\mathsf{rec}}$ is meant to be included in $\mathbf{SC}$.

Whenever a stakeholder on $\mathbf{MC}$ that has adopted $\mathbf{SC}$ wants to transfer funds to $\mathbf{SC}$, she diffuses $\mathsf{tx}_{\mathsf{send}}$ with the correct receiving account on $\mathbf{SC}$ and the desired amount. Honest slot leaders in $\mathbf{MC}$ include these transactions into their blocks just like any intra-chain transfer transactions. Maintainers of $\mathbf{MC}$ keep account of a variable $\mathsf{pool}_{\mathbf{SC}}$, initially set to zero. Whenever a $\mathsf{tx}_{\mathsf{send}}$ is included into $\mathbf{MC}$, they increase $\mathsf{pool}_{\mathbf{SC}}$ by the amount of this transaction.

When $\mathsf{tx}_{\mathsf{send}}$ becomes stable in $\mathbf{MC}$ (i.e., appears in MC, this happens at most $2k$ slots after its inclusion), the stakeholder creates and diffuses the corresponding $\mathsf{tx}_{\mathsf{rec}}$ which credits the respective amount of coins to rAcc in $\mathbf{SC}$, to be included into $\mathbf{SC}$. In practice, this is akin to a coinbase transaction, as the money was not transferred from an existing $\mathbf{SC}$ account.

Note that depositing from $\mathbf{MC}$ to $\mathbf{SC}$ is relatively fast; it merely requires a reliable inclusion of $\mathsf{tx}_{\mathsf{send}}$ into $\mathbf{MC}$ and consequently of $\mathsf{tx}_{\mathsf{rec}}$ into $\mathbf{SC}$, as guaranteed by the liveness of the underlying Ouroboros instances. The depositing algorithm code is shown in Algorithm 48.

Algorithm 48 Depositing from $\mathbf{MC}$ to $\mathbf{SC}$.

The algorithm is run by a stakeholder U in control of the secret key sk corresponding to the account sAcc on $\mathbf{MC}$.

```
function Send(sAcc, rAcc, $v$)          ▷ Send $v$ from sAcc on MC to rAcc on SC
    txid $\xleftarrow{\$} \{0,1\}^k$
    $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{MC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v)$
    $\mathsf{tx}_{\mathsf{send}} \leftarrow (\mathsf{txid}, \mathbf{MC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v, \sigma)$
    post $\mathsf{tx}_{\mathsf{send}}$ to MC
end function
function Receive(txid, sAcc, rAcc, $v$)
    wait until $\mathsf{tx}_{\mathsf{send}} \in \mathsf{MC}$          ▷ MC is the stable part of MC
    $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{SC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v)$
    $\mathsf{tx}_{\mathsf{rec}} \leftarrow (\mathsf{txid}, \mathbf{SC}, (\mathbf{MC}, \mathsf{sAcc}), (\mathbf{SC}, \mathsf{rAcc}), v, \sigma)$
    post $\mathsf{tx}_{\mathsf{rec}}$ to SC
end function
```

Withdrawing to MC The withdrawal operation is more cumbersome than the depositing operation since not all nodes of $\mathbf{MC}$ have adopted (i.e., are aware of and follow) the sidechain $\mathbf{SC}$. As transactions, the withdrawals have the same structure as deposits, consisting of $\mathsf{tx}_{\mathsf{send}}$ and $\mathsf{tx}_{\mathsf{rec}}$, with the only difference that now they both have $\mathsf{send} = \mathbf{SC}$ and $\mathsf{rec} = \mathbf{MC}$. The sending transaction will be handled in the same way as in the case of deposits, but the receiving transaction requires a different certificate-based treatment, as detailed below.

Whenever a stakeholder in $\mathbf{SC}$ wishes to withdraw coins from $\mathbf{SC}$ to $\mathbf{MC}$, she creates and diffuses the respective transaction $\mathsf{tx}_{\mathsf{send}}$ with the correct transfer details as before. If $\mathsf{tx}_{\mathsf{send}}$ is included in a block that belongs in one of the first $R-4k$ slots of some epoch then let j_{send} denote the index of this epoch, otherwise let j_{send} denote the index of the following epoch. The stakeholder then waits for the end of the epoch $e_{j_{\mathsf{send}}}$ to pass and $e_{j_{\mathsf{send}}+1}$ to begin.

At the beginning of $e_{j_{\mathsf{send}}+1}$, a special transaction called *sidechain certificate* $\mathsf{sc\_cert}_{j_{\mathsf{send}}+1}$ is generated by the maintainers of **SC**. It contains: (i) a Merkle-tree commitment to all withdrawal transactions $\mathsf{tx}_{\mathsf{send}}$ that were included into **SC** during last $4k$ slots of epoch $j_{\mathsf{send}}-1$ and the first $R-4k$ slots of epoch j_{send} (as these all are already stable by slot $R-2k$ of epoch j_{send}); (ii) other information allowing the maintainers of **MC** to inductively validate the certificate in every epoch. The construction of sc_cert is detailed below, for now assume that the transaction provides a proof that the included information about withdrawal transactions is correct. The transaction sc_cert is broadcast into the **MC** network to be included into **MC** at the beginning of $e_{j_{\mathsf{send}}+1}$ by the first honest slot leader.

The stakeholder who wishes to withdraw their money into **MC** now creates and diffuses the transaction $\mathsf{tx}_{\mathsf{rec}}$ to be included in **MC**. This transaction is only included into **MC** if it is considered valid, which means: (1) it is properly signed; (2) it contains a Merkle inclusion proof confirming its presence in some already included sidechain certificate; (3) its amount is less or equal to the current value of $\mathsf{pool}_{\mathbf{SC}}$. If included, **MC**-maintainers decrease the value of $\mathsf{pool}_{\mathbf{SC}}$ by the amount of this transaction. The code of the withdrawal algorithm is illustrated in Algorithm 49.

Algorithm 49 Withdrawing from **SC** to **MC**.

The algorithm is run by a stakeholder U in control of the secret key sk corresponding to the account sAcc on **SC**.

```
1: function Send(sAcc, rAcc, $v$)          ▷ Send $v$ from sAcc on SC to rAcc on MC
2:     $\mathsf{txid} \xleftarrow{\$} \{0,1\}^k$
3:     $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{SC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v)$
4:     $\mathsf{tx}_{\mathsf{send}} \leftarrow (\mathsf{txid}, \mathbf{SC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \sigma)$
5:     post $\mathsf{tx}_{\mathsf{send}}$ to SC
6: end function
7: function Receive(txid, sAcc, rAcc, $v$)
8:     wait until $\mathsf{tx}_{\mathsf{send}} \in \mathsf{C}_{\mathbf{SC}}$
9:     $j' \leftarrow$ epoch where $\mathsf{C}_{\mathbf{SC}}$ contains $\mathsf{tx}_{\mathsf{send}}$
10:    if ($\mathsf{tx}_{\mathsf{send}}$ included in slot $sl \leq R-4$ of $e_{j'}$) then
11:        $j_{\mathsf{send}} \leftarrow j'$
12:    else
13:        $j_{\mathsf{send}} \leftarrow j'+1$
14:    end if
15:    wait until $\mathsf{sc\_cert}_{j_{\mathsf{send}}+1} \in \mathsf{C}_{\mathbf{MC}}$
16:    $\pi \leftarrow$ Merkle-tree proof of $\mathsf{tx}_{\mathsf{send}}$ in $\mathsf{sc\_cert}_{j_{\mathsf{send}}+1}$
17:    $\sigma \leftarrow \mathsf{Sig}_{sk}(\mathsf{txid}, \mathbf{MC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \pi)$
18:    $\mathsf{tx}_{\mathsf{rec}} \leftarrow (\mathsf{txid}, \mathbf{MC}, (\mathbf{SC}, \mathsf{sAcc}), (\mathbf{MC}, \mathsf{rAcc}), v, \pi, \sigma)$
19:    post $\mathsf{tx}_{\mathsf{rec}}$ to MC
20: end function
```

The certificate transaction We now describe the construction of the sc_cert transaction, also called the *sidechain certificate*, formally described in Algorithm 50).

The role of the certificate produced by the end of epoch $j-1$ to be included in

Algorithm 50 Constructing sidechain certificate $\mathsf{sc\_cert}$.

The algorithm is run by every **SC**-maintainer at the end of each epoch, j denotes the index of the ending epoch.

```
1: function ConstructSCCert($j$)
2:     $T \leftarrow$ last $4k$ slots of $e_{j-1}$ and first $R-4k$ slots of $e_j$
3:     $\vec{\mathsf{tx}} \leftarrow$ transactions included in SC during $T$
4:     $\mathsf{pending}_{j+1} \leftarrow \{\mathsf{tx} \in \vec{\mathsf{tx}} : \mathsf{tx.send} \neq \mathsf{tx.rec}\}$
5:     $\mathcal{VK}_{j+1} \leftarrow$ keys of last $2k$ SC slot leaders in $e_{j+1}$
6:     $avk^{j+1} \leftarrow \mathsf{AKey}(\mathcal{VK}_{j+1})$
7:     $m \leftarrow (\langle \mathsf{pending}_{j+1} \rangle, avk^{j+1})$
8:     $\mathcal{VK}_j \leftarrow$ keys of last $2k$ SC slot leaders for $e_j$
9:     $\sigma_{j+1} \leftarrow \mathsf{ASig}\left(m, \{(vk_i, \sigma_i)\}_{i=1}^{d}, \mathcal{VK}_j\right)$
10:    $\mathsf{sc\_cert}_{j+1} \leftarrow (\langle \mathsf{pending}_{j+1} \rangle, avk^{j+1}, \sigma_{j+1})$ return $\mathsf{sc\_cert}_{j+1}$
11: end function
```

MC at the beginning of epoch j (denoted $\mathsf{sc\_cert}_j$) is to attest all the withdrawals that had their sending transactions included into **SC** in either the last $4k$ slots of e_{j-2} or the first $R-4k$ slots of e_{j-1}. To maintain a chain of trust for the **MC** maintainers that cannot verify these transactions by observing **SC**, we make use of ad-hoc threshold multisignatures introduced in Section 6.1. Namely, the $\mathsf{sc\_cert}_j$ transaction also contains an aggregate key avk^j of an ATMS, and is signed by the previous aggregate key avk^{j-1} included in $\mathsf{sc\_cert}_{j-1}$.

$\mathsf{sc\_cert}_j$ is generated by **SC**-maintainers and contains:

- **The epoch index** j**.**
- **The pending transactions from SC to MC.** Let $\vec{\mathsf{tx}}$ be the sequence of all transactions which are included in **SC** during either the last $4k$ slots of e_{j-2} or the first $R-4k$ slots of e_j. All transactions in $\vec{\mathsf{tx}}$ that have $\mathbf{SC} = \mathsf{send} \neq \mathsf{rec} = \mathbf{MC}$ are picked up and combined into a list $\mathsf{pending}_j$ (sorted in the same order as in **SC**). Let $\langle \mathsf{pending}_j \rangle$ denote a Merkle-tree commitment to this list.
- **The new ATMS key** avk^j**.** The key is created from the public keys of the slot leaders of the last $2k$ slots of the epoch j, using threshold $k+1$. Hence, it allows to verify whether a particular signature comes from $k+1$ out of these $2k$ keys.
- **Signature valid with respect to** avk^{j-1}**.**

The full $\mathsf{sc\_cert}_j$ is therefore a tuple $\left(j, \langle \mathsf{pending}_j \rangle, avk^j, \sigma_j\right)$, where σ_j is an ATMS signature on the preceding elements that verifies using avk^{j-1}.

The certificate $\mathsf{sc\_cert}_{j+1}$ is constructed as follows: Both the stake distribution $\overline{\mathsf{SD}}^*_{j+1}$ and the **SC**-randomness $\bar{\eta}_{j+1}$ (and hence also the slot leader schedule for **SC** in epoch $j+1$) are determined by the states of the blockchains **MC** and **SC** by the end of slot $10k$ of epoch j. Therefore, during the last $2k$ slots of epoch j, the $2k$ elected slot leaders for these slots can already include a (local) signature on (their proposal of) $\mathsf{sc\_cert}_{j+1}$ into the blocks they create. Given the deterministic construction of $\mathsf{sc\_cert}_{j+1}$, all valid blocks ending up in the part of **SC**-chain belonging to the last $2k$ slots of epoch j will contain a local signature on the same $\mathsf{sc\_cert}_{j+1}$, and by the chain growth property of the underlying blockchain, there will be at least $k+1$ of them. Therefore, any party observing **SC** can now combine these signatures into an ATMS that can be later verified using the ATMS key avk^j, it can hence create the complete certificate $\mathsf{sc\_cert}_{j+1}$ and serve it to the maintainers of **MC** for inclusion.

Transitioning trust As already outlined above, our construction uses ATMS to maintain the authenticity of the sidechain certificates from epoch to epoch. We now describe this inductive process in greater detail.

Initially, during the setup of the sidechain, $\mathcal{P} \leftarrow \mathsf{PGen}(1^\kappa)$ is ran. Stakeholders generate their keys by invoking $(sk_i, vk_i) \leftarrow \mathsf{Gen}(\mathcal{P})$. In case $\mathsf{Gen}(\cdot)$ is a probabilistic algorithm, it is run in a derandomized fashion with its coins fixed to the output of a PRNG that is seeded by $H(\mathsf{ats\_init}, \eta_{j_{\mathsf{start}}})$ where "ats_init" is a fixed label and H is a hash function. This ensures that $\mathcal{P}$ will be uniquely determined and will still be unpredictable. We note that this process is only suitable for ATMS that employ public-coin parameters; our ATMS constructions in Section 6.2 are only of this type.

For the induction base, $\mathcal{P}$ is published as part of the Genesis block $\overline{\mathsf{G}}$. Each time an **MC** stakeholder U_i posts the $\mathsf{sidechain\_support}$ message to **MC**, he also includes an ATMS key vk_i. Subsequently, when the **SC** is initialised, the stake distribution $\overline{\mathsf{SD}}^*_{j_{\mathsf{start}}}$ is known to the **MC** participants. Hence, based on $\overline{\mathsf{SD}}^*_{j_{\mathsf{start}}}$ and $\bar{\eta}_{j_{\mathsf{start}}}$, these can determine the last $2k$ slot leaders of epoch j_{start} in **SC**, we will refer to them as the j_{start}-th *trust committee.* (In general, the j-th *trust committee* for $j \geq j_{\mathsf{start}}$ will be the set of last $2k$ slot leaders in epoch j.) **SC**-maintainers (that also follow **MC**) can also determine the j_{start}-th trust committee and therefore create $avk^{j_{\mathsf{start}}}$ from their public keys and insert it into the genesis block $\overline{\mathsf{G}}$ of **SC**. They can also serve it as a special transaction to the **MC**-maintainers to include into the mainchain. The correctness of $avk^{j_{\mathsf{start}}}$ can be readily verified by anyone following the mainchain using the procedure ACheck of the used ATMS.

For the induction step, consider an epoch $j > j_{\mathsf{start}}$ and assume that there exists an ATMS key of the previous epoch avk^{j-1}, known to the mainchain maintainers. Every honest **SC** slot leader among the last $2k$ slot leaders of **SC** epoch $j-1$ will produce a local signature s_i^j on the message $m = (j, \langle \mathsf{pending}_j \rangle, avk_j)$ using their private key sk_i^{j-1} by running $\mathsf{Sig}(sk_i^{j-1}, m)$, and include this signature into the block they create. The rest of the **SC** maintainers will verify that the epoch index, avk^j and $\langle \mathsf{pending}_j \rangle$ are correct (by ensuring $\mathsf{ACheck}(\mathcal{VK}^j, avk^j)$ is *true* for $\mathcal{VK}$ denoting the public keys of the last $2k$ slot leaders on **SC** for epoch j, and by recomputing the Merkle tree commitment $\langle \mathsf{pending}_j \rangle$) and that s_i^j is valid by running $\mathsf{Ver}(m, vk_i^{j-1}, s_i^j)$, otherwise the block is considered invalid. Thanks to the chain growth property of the underlying Ouroboros protocol, after the last $2k$ slots of epoch $j-1$ the honest sidechain maintainers will all observe at least $k+1$ signatures among the $\{s_i^j : i \in [2k]\}$ desired ones. They then combine all of these local signatures into an aggregated ATMS signature $\sigma^j \leftarrow \mathsf{ASig}(m, \{(s_i^j, vk_i^{j-1})\}, \mathsf{keys}^j)$. This combined signature is then diffused as part of $\mathsf{sc\_cert}_j$ on the mainchain network. The mainchain maintainers verify that it has been signed by the sidechain maintainers by checking that $\mathsf{AVer}(m, avk^{j-1}, \sigma^j)$ evaluates to *true* and include it in a mainchain block. This effectively hands over control to the new committee.

7.1.3 Security of Sidechains

In this section we give a formal argument establishing that the construction from Section 7.1.2 achieves pegging security of Definition 79.

Assumptions

Let $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ denote the honest-majority assumption for an Ouroboros ledger $\mathbf{L}$. Namely, $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ postulates that in all slots $t' \leq t$, the majority of stake in the stake distribution used to sample the slot leader for slot t' in $\mathbf{L}$ is controlled by honest parties (note that the distribution in question is SD and $\overline{\mathsf{SD}}^*$ for $\mathbf{MC}$ and $\mathbf{SC}$, respectively). Specifically, the adversary is restricted to $(1-\epsilon)/2$ relative stake for some fixed $\epsilon > 0$.

The assumption $\mathbb{A}_{\mathbf{MC}}$ we consider for $\mathbf{MC}$ is precisely $\mathbb{A}_{\mathbf{MC}}[t] \triangleq \mathbb{A}_{\mathsf{hm}}(\mathbf{MC})[t]$, while the assumption $\mathbb{A}_{\mathbf{SC}}$ for $\mathbf{SC}$ is $\mathbb{A}_{\mathbf{SC}}[t] \triangleq \mathbb{A}_{\mathbf{MC}}[t] \wedge \mathbb{A}_{\mathsf{hm}}(\mathbf{SC})[t]$. The reason that $\mathbb{A}_{\mathbf{SC}}[t] \Rightarrow \mathbb{A}_{\mathbf{MC}}[t]$ is that $\mathbf{SC}$ uses merged staking and hence cannot provide any security guarantees if the stake records on $\mathbf{MC}$ get corrupted. It is worth noting that it is possible to program $\mathbf{SC}$ to wean off $\mathbf{MC}$ and switch to independent staking; in such case the assumption for $\mathbf{SC}$ will transition to $\mathbb{A}_{\mathsf{hm}}(\mathbf{SC})$ (now with respect to $\overline{\mathsf{SD}}$) after the weaning slot and the two chains will become sidechains of each other.

Remark 10. *We note that the assumption of honest majority in the distribution out of which leaders are* sampled *is one of two related ways of stating this requirement. The distribution from which sampling is performed corresponds to the actual stake distribution near the end of the previous epoch. Hence, the actual stake may have since shifted and may no longer be honest. Had we wanted to formulate this assumption in terms of the actual (current) stake distribution, we would have to state two different assumptions: (1) that the current actual stake has honest majority with some gap σ; and (2) that the rate of stake shifting is bounded by σ for the duration of (roughly) 2 epochs. From these two assumptions, one can conclude that the distribution from which leaders are elected is currently controlled by an honest majority. The latter approach was taken for example in [89].*

Proof Overview

Proving our construction secure requires some case analysis. We summarize the intuition behind this endeavour before we proceed with the formal treatment.

The proof of Theorem 69 that shows that our construction from Section 7.1.2 has pegging security with overwhelming probability will be established as follows. We will borrow the fact that our construction achieves persistence and liveness from the original analysis [89] and state them as Lemma 62. The main challenge will be to establish the firewall property, which is done in Lemma 68. These properties together establish pegging security as required by Definition 79.

To show that the firewall property holds, we perform a case analysis, looking at the two cases of interest: when both $\mathbf{MC}$ and $\mathbf{SC}$ are secure (i.e., when $\mathbb{A}_{\mathbf{MC}} \wedge \mathbb{A}_{\mathbf{SC}}$ holds), and when only $\mathbf{MC}$ is secure while the security assumption of $\mathbf{SC}$ has been violated. As discussed above, the case where $\mathbf{SC}$ is secure and the security of $\mathbf{MC}$ has been violated cannot occur per definition of $\mathbb{A}_{\mathbf{MC}}$ and $\mathbb{A}_{\mathbf{SC}}$, and so examining this case is not necessary.

First, we examine the case where both **MC** and **SC** are secure, but only concern ourselves with *direct observation* transactions, or transactions that can be verified without relying on sidechain certificates. We show that such transactions will always be correctly verified in this case.

Next, we establish that, when only **MC** is secure, it is impossible for the **MC** maintainers to accept a view inconsistent with the validity language, and hence the firewall property is maintained in the case of a sidechain failure.

Finally, the heart of the proof is a computational reduction (using the above partial results) showing how, given an adversary that breaks the firewall property, there must exist a receiving transaction on **MC** which breaks the validity of the scheme. Given such a transaction, we can construct an adversary against either the security of the underlying ATMS scheme or the collision resistance of the underlying hash function.

Liveness and Persistence

We begin by stating the persistence and liveness guarantees of our construction, they both follow directly from the guarantees shown for the standalone Ouroboros blockchain in [89].

Lemma 62 (Persistence and Liveness)**.** *Consider the construction of Section 7.1.2 with the assumptions* $\mathbb{A}_{\mathbf{SC}}, \mathbb{A}_{\mathbf{MC}}$. *For all slots* t*, if* $\mathbb{A}_{\mathbf{SC}}[t]$ *(resp.* $\mathbb{A}_{\mathbf{MC}}[t]$*) holds, then* **SC** *(resp.* **MC***) satisfies persistence and liveness up to slot* t *with overwhelming probability in* k*.*

We now restate the Common Prefix property of blockchains for future reference. If the Common Prefix property holds, then Persistence can be derived along the lines of [89].

Definition 80 (Common Prefix)**.** *For every honest party* P_1 *and* P_2 *both maintaining the same ledger (i.e., either both maintaining* **MC***, or both maintaining* **SC***) and for every slot* r_1 *and* r_2 *such that* $r_1 \leq r_2 \leq t$*, let* C_1 *be the adopted chain of* P_1 *at slot* r_1 *and* C_2 *be the adopted chain of* P_2 *at slot* r_2*. The* k-common prefix property *for slot* t *states that* $\mathsf{C}_2[: |\mathsf{C}_1[: -k]|] = \mathsf{C}_1[: -k]$.

The Firewall Property and MC-Receiving Transactions

Recall that the transactions in $\mathcal{T}_{\mathfrak{A}}$ can be partitioned into several classes with different validity-checking procedures. First, there are *local* transactions (where $\mathsf{send} = \mathsf{rec} = \mathsf{lid}$) and *sending* transactions (with $\mathsf{lid} = \mathsf{send} \neq \mathsf{rec}$). Then we have *receiving* transactions (with $\mathsf{send} \neq \mathsf{rec} = \mathsf{lid}$), which can be split into **SC**-*receiving* transactions ($\mathsf{send} \neq \mathsf{rec} = \mathsf{lid} = \mathbf{SC}$) and **MC**-*receiving* transactions ($\mathsf{send} \neq \mathsf{rec} = \mathsf{lid} = \mathbf{MC}$).

As the lemma below observes, if a transaction violates the firewall property in a certain situation, it must be an **MC**-receiving transaction.

Lemma 63. *Consider an execution of the protocol of Section 7.1.2 at slot* t *in which* **MC** *and* **SC** *satisfy persistence. Suppose*

$$\mathsf{L} = \mathsf{merge}\left(\{\mathsf{L}^{\cup}_{\mathbf{MC}}[t], \mathsf{L}^{\cup}_{\mathbf{SC}}[t]\}\right) \notin \mathbb{V}_{\mathfrak{A}}$$

and suppose that $\mathcal{S}_t = \{\mathbf{SC}, \mathbf{MC}\}$*. Let* L' *be the minimum prefix of* L *such that* $\mathsf{L}' \notin \mathbb{V}_{\mathfrak{A}}$*. Then* $\mathsf{L}' \neq \varepsilon$ *and* $\mathsf{tx} \triangleq \mathsf{L}'[-1]$ *is an* **MC**-*receiving transaction.*

Proof. The *base* property of the validity language implies $\mathsf{L}' \neq \varepsilon$, hence tx exists. Due to the minimality of L', Algorithm 40 returns *false* for L' but *true* for $\mathsf{L}'[:-1]$. Since it processes transactions sequentially, it must return *false* during the processing of tx. Suppose for contradiction that tx is not an **MC**-receiving transaction; let us call such a transaction *direct* in this proof.

Algorithm 40 can output *false* while processing a direct transaction in the following cases: (a) in Line 18 when there is a Conservation Law violation; (b) in Line 9 when there is a signature validation failure; (c) in Line 14 when tx is a replay of a previous transaction; (d) in Line 23 when tx is a replay, or (e) in Line 28 when the pre-image transaction has not yet been processed. Hence, tx falls under one of these violations.

Due to persistence and the definition of $\mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{\cup}_{\mathbf{SC}}[t]$, there exists an **MC** maintainer $P_{\mathbf{MC}}$ and an **SC** maintainer $P_{\mathbf{SC}}$, such that $\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t] = \mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{P_{\mathrm{SC}}}_{\mathbf{SC}}[t] = \mathsf{L}^{\cup}_{\mathbf{SC}}[t]$, respectively. Due to the *partitioning property* of merge, tx will be in $\mathsf{L}^{P_{\mathsf{lid(tx)}}}_{\mathsf{lid(tx)}}[t]$. We separately consider the two possibilities for $\mathsf{lid(tx)}$.

Case 1: $\mathsf{lid(tx)} = \mathbf{MC}$. In this case, the only violations that a direct tx can attain are (a), (b) and (c), as the cases (d) and (e) for $\mathsf{lid(tx)} = \mathbf{MC}$ do not pertain to a direct transaction. $P_{\mathbf{MC}}$ has reported $\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t]$ as its adopted state, hence $\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t]$ is a fixpoint of $\mathrm{verifytx}_{\mathbf{MC}}$ (as $\mathrm{verifytx}_{\mathbf{MC}}$ checks for a fixpoint). The execution of $\mathrm{verifytx}_{\mathbf{MC}}$ included every transaction in $\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t]$. Therefore, $\mathrm{verifytx}_{\mathbf{MC}}$ has accepted every transaction in every iteration until the last iteration, which processes tx. Consider, now, what happened in the last iteration of the execution of $\mathrm{verifytx}_{\mathbf{MC}}$. In that iteration, $\mathrm{verifytx}_{\mathbf{MC}}$ checks the validity of σ, the Conservation Law, and transaction replay. In all cases (a), (b) and (c), $\mathrm{verifytx}_{\mathbf{MC}}$ will reject tx. But this could not have happened, as $\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t]$ is a fixpoint, and we have a contradiction.

Case 2: $\mathsf{lid(tx)} = \mathbf{SC}$. Let C_{mc} and C_{sc} be the **MC** and respectively **SC** chain adopted by $P_{\mathbf{SC}}$ at slot t (and recall that $P_{\mathbf{SC}}$ maintains both chains). Let $\mathsf{C}_{\mathsf{mc}}'$ be the chain adopted by $P_{\mathbf{MC}}$ at slot t. As before, $\mathrm{annotatetx}_{\mathbf{SC}}(\mathsf{C}_{\mathsf{mc}}, \mathsf{C}_{\mathsf{sc}})$ must be a fixpoint of $\mathrm{verifytx}_{\mathbf{SC}}$ (as $\mathrm{verifytx}_{\mathbf{SC}}$ checks for a fixpoint). As in the previous case, tx cannot violate (a), (b), (c) and in this case nor (d), as this would constitute a fixpoint violation. Hence tx is an effect transaction and we will examine whether tx constitutes a violation of (e).

Let $\mathsf{tx}^{-1} \triangleq \mathsf{effect}^{-1}_{\mathbf{MC}\to\mathbf{SC}}(\mathsf{tx})$. Since tx is accepted by $\mathrm{verifytx}_{\mathbf{SC}}$ on input $\mathrm{annotatetx}_{\mathbf{SC}}(\mathsf{C}_{\mathsf{mc}}, \mathsf{C}_{\mathsf{sc}})$, we deduce that there exists some block $B \in \mathsf{C}_{\mathsf{mc}}[:-k]$ with $\mathsf{tx}^{-1} \in B$. But $\mathsf{C}_{\mathsf{mc}}'[:-k]$ is the longest stable chain among **MC** maintainers (due to $\mathsf{L}^{\cup}_{\mathbf{MC}}[t] = \mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t]$), hence $\mathsf{C}_{\mathsf{mc}}[:-k]$ is its prefix. Therefore $B \in \mathsf{C}_{\mathsf{mc}}'[:-k]$. Hence, $\mathsf{tx}^{-1} \in \mathbf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t]$. Due to the *partioning property* of merge, tx^{-1} must appear in the output of $\mathsf{merge}\left(\{\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t], \mathsf{L}^{P_{\mathrm{SC}}}_{\mathbf{SC}}[t]\}\right)$. Due to the *topological soundness* of merge, tx^{-1} must appear *before* tx in $\mathsf{merge}\left(\{\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t], \mathsf{L}^{P_{\mathrm{SC}}}_{\mathbf{SC}}[t]\}\right)$. Hence, it cannot be the case that (e) is violated, as the pre-image transaction exists. □

Firewall Property During Sidechain Failure

We now turn our attention to the case where the sidechain has suffered a "catastrophic failure" and so $\mathcal{S}_t = \{\mathbf{MC}\}$. We describe why a catastrophic failure in the sidechain does not violate the firewall property. To do this, we need to illustrate that, given a transaction sequence L which is accepted by the **MC** verifier, we can "fill in the gaps" with transactions from $\mathbf{SC}$ in order to produce a new transaction sequence $\vec{\mathsf{tx}}$ which is valid with respect to $\mathbb{V}_{\mathfrak{A}}$.

We prove this constructively in Lemma 65. The construction of such a sequence is described in Algorithm 51. The algorithm accepts a transaction sequence $\mathsf{L} \subseteq \mathcal{T}_{\mathbf{MC}}$ valid according to verifier$_{\mathbf{MC}}$ and produces a transaction sequence $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$ satisfying $\pi_{\mathbf{MC}}(\vec{\mathsf{tx}}) = \mathsf{L}$, as desired.

The algorithm works by mapping each $\mathsf{tx} \in \mathsf{L}$ to one or more transactions in $\vec{\mathsf{tx}}$. The mapping is done by calling $\mathsf{plausibility\text{-}map}(\mathsf{tx})$ for each transaction individually. Hence each transaction in $\vec{\mathsf{tx}}$ has a specific preimage transaction in L, which can be shared by other transactions in $\vec{\mathsf{tx}}$. The mapping is performed as follows. If tx is a local transaction, then it is simply copied over, otherwise some extra transactions are included. Specifically, if it's an sending transaction tx, then first tx is included, and subsequently the funds are recovered by a corresponding transaction tx_1 on $\mathbf{SC}$, the effect transaction of tx. The funds are afterwards moved to a pool address pool_{pk} by a transaction tx_2. (Note that for this, we assume that the receiving account public key has a correspnding private key, as this key is needed to sign tx_2. As we are only demonstrating the existence of $\vec{\mathsf{tx}}$, Algorithm 51 does not need to be efficient and so assuming the existence of the private key is sufficient.) On the other hand, if it is an ($\mathbf{MC}$-)receiving transaction tx, the reverse procedure is followed. First, the funds are collected by tx_2 from the pool address pool_{pk} and moved into the $\mathbf{SC}$ address which will be used for the upcoming remote transaction. Then tx_1 moves the funds out of $\mathbf{SC}$ so that they can be collected by the corresponding tx on $\mathbf{MC}$. In the first case, the transaction sequence is $(\mathsf{tx}, \mathsf{tx}_1, \mathsf{tx}_2)$ and in the second case the sequence is $(\mathsf{tx}_2, \mathsf{tx}_1, \mathsf{tx})$. Note that, in both cases, tx and tx_1 are identical, except for the fact that tx is recorded on $\mathbf{MC}$ while tx_1 is recorded on $\mathbf{SC}$; the latter is the effect (or pre-image, respectively) of the former.

The simple intuition behind this construction is that, in the plausible history $\vec{\mathsf{tx}}$ produced by Algorithm 51, the account pool_{pk} is holding all the money of the sidechain. More specifically, the balance that is maintained in the variable $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ is identical to the pool variable maintained by the $\mathbf{MC}$ verifier. This invariant is made formal in Lemma 64.

Lemma 64 (Plausible balances)**.** *Let* $\mathsf{L} \in \mathcal{T}^*_{\mathfrak{A},\mathbf{MC}}$ *and* $\vec{\mathsf{tx}} \leftarrow \mathsf{plausible}(\mathsf{L})$*. Consider an execution of Algorithm 40 on* $\vec{\mathsf{tx}}$ *and an execution of* verifier$_{\mathbf{MC}}$ *on* L*. Let* $\mathsf{tx} \in \mathsf{L}$*. Call* $\mathsf{pool}_{\mathsf{tx}}$ *the value of the* pool *variable maintained by* verifier$_{\mathbf{MC}}$ *prior to processing* tx *in its main* for *loop; call* $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}}$ *the value of the* $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ *variable prior to the iteration of its main* for *loop which processes the first item of* $\mathsf{plausibility\text{-}map}(\mathsf{tx})$*. For all* $\mathsf{tx} \in \mathsf{L}$*, the following invariant will hold:* $\mathsf{pool}_{\mathsf{tx}} = \mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}}$*.*

Proof. By direct inspection of the two algorithms, observe that $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]$ are updated by Algorithm 40 only when $\mathsf{send}(\mathsf{tx}_a) \neq \mathsf{rec}(\mathsf{tx}_a)$. The balances are increased when $\mathsf{send}(\mathsf{tx}_a) = \mathbf{MC}$ (due to $\mathsf{tx}_2 \in \mathsf{plausibility\text{-}map}(\mathsf{tx})$ at Line 17 of Algorithm 51) and decreased when $\mathsf{send}(\mathsf{tx}_a) = \mathbf{SC}$ (due to $\mathsf{tx}_2 \in \mathsf{plausibility\text{-}map}(\mathsf{tx})$ at Line 22 of Algorithm 51). Exactly the same accounting is performed by verifier$_{\mathbf{MC}}$ when the respective tx is processed. □

Algorithm 51 The plausible transaction sequence generator.

```
(pool$_{sk}$, pool$_{pk}$) $\leftarrow$ Gen($1^\lambda$)
function plausible($L$)
    $\vec{\mathsf{tx}} \leftarrow \varepsilon$
    for tx $\in L$ do
        $\vec{\mathsf{tx}} \leftarrow \vec{\mathsf{tx}}$ || plausibility-map(tx)
    end forreturn $\vec{\mathsf{tx}}$
end function
function plausibility-map(tx)
    ▷ Destructure tx into its constituents
    (txid, lid, (send, sAcc), (rec, rAcc), $v$, $\sigma$) $\leftarrow$ tx
    if send = rec then return (tx)
    end if
    if send = MC then
        tx$_1$ $\leftarrow$ effect$_{\mathbf{MC}\to\mathbf{SC}}$(tx)
        Construct a valid $\sigma_2$ using the private key corresponding to rAcc
        Generate a fresh txid$_2$
        tx$_2$ $\leftarrow$ (txid$_2$, SC, (SC, rAcc), (SC, pool$_{pk}$), $v$, $\sigma_2$)
      return (tx, tx$_1$, tx$_2$)
    end if
    if send = SC then
        Construct a valid $\sigma_2$ using pool$_{sk}$
        Generate a fresh txid$_2$
        tx$_2$ $\leftarrow$ (txid$_2$, SC, (SC, pool$_{pk}$), (SC, sAcc), $v$, $\sigma_2$)
        tx$_1$ $\leftarrow$ effect$^{-1}_{\mathbf{SC}\to\mathbf{MC}}$(tx) return (tx$_2$, tx$_1$, tx)
    end if
end function
```

We now prove the correctness of Algorithm 51 in Lemma 65.

Lemma 65 (Plausibility)**.** *For all* $\mathsf{L} \in \mathcal{T}^*_{\mathfrak{A},\mathbf{MC}}$*, if* $\text{verifytx}_{\mathbf{MC}}(\mathsf{L}) = \mathsf{L}$ *then* $\vec{\mathsf{tx}} \leftarrow$ *plausible*(L) *will satisfy* $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$*.*

Proof. Suppose for contradiction that $\vec{\mathsf{tx}} \notin \mathbb{V}_{\mathfrak{A}}$ and let $\vec{\mathsf{tx}}'$ be the minimum prefix of $\vec{\mathsf{tx}}$ such that $\vec{\mathsf{tx}}' \notin \mathbb{V}_{\mathfrak{A}}$. From the validity language *base* property we have that $\vec{\mathsf{tx}}' \neq \varepsilon$ and so it must have at least one element. Let $\mathsf{tx} \triangleq \vec{\mathsf{tx}}'[-1]$ and let $\mathsf{tx_L} \in \mathsf{L}$ be the input to plausibility-map which caused tx to be included in $\vec{\mathsf{tx}}$ in the execution of plausible in Algorithm 51. Since Algorithm 40 processes transactions sequentially, and by the minimality of $\vec{\mathsf{tx}}'$, it must return *false* when tx is processed.

We distinguish the following cases for $\mathsf{tx_L}$:

Case 1: Local transaction: $\mathsf{send}(\mathsf{tx_L}) = \mathsf{rec}(\mathsf{tx_L})$. Then $\mathsf{tx} = \mathsf{tx_L}$ and $\mathsf{send}(\mathsf{tx}) = \mathsf{lid}(\mathsf{tx})$. Since L is a fixpoint of $\text{verifytx}_{\mathbf{MC}}$, tx must (a) have a valid signature σ, (b) not be a replay transaction, and (c) respect the Conservation Law. As $\mathsf{tx_L}$ is a local transaction satisfying all of (a), (b) and (c), therefore $\vec{\mathsf{tx}}' \in \mathbb{V}_{\mathfrak{A}}$, which is a contradiction.

Case 2: Sending transaction: $\mathsf{send}(\mathsf{tx_L}) = \mathbf{MC}$ and $\mathsf{rec}(\mathsf{tx_L}) = \mathbf{SC}$. In this case, let $(\mathsf{tx_L}, \mathsf{tx}_1, \mathsf{tx}_2) = \text{plausibility-map}(\mathsf{tx_L})$. If $\mathsf{tx} = \mathsf{tx_L}$, then tx is a sending transaction and we can apply the same reasoning to argue that it will respect properties (a), (b) and (c). But those are the only violations for which Algorithm 40 can reject an sending transaction, and hence $\vec{\mathsf{tx}}' \in \mathbb{V}_{\mathfrak{A}}$, which is a contradiction.

If $\mathsf{tx} = \mathsf{tx}_1$, then Algorithm 40 must return *true*. To see this, consider the cases when Algorithm 40 returns *false*: (d) a replay failure in Line 23, which cannot occur as tx_L has been accepted by $\text{verifytx}_{\mathbf{MC}}$ and so $\text{verifytx}_{\mathbf{MC}}$ must have *seen* tx_L only once while Algorithm 40 must be seeing it for exactly the second time; or (e) a mismatch failure in Line 23 which cannot occur as tx_1 is constructed identical to tx_L.

If $\mathsf{tx} = \mathsf{tx}_2$ then $\mathsf{send}(\mathsf{tx}) = \mathsf{rec}(\mathsf{tx})$. This transaction cannot cause Algorithm 40 to return *false*. To see this, consider the cases when Algorithm 40 returns *false*: (a) a signature failure in Line 9 cannot occur because σ_2 was constructed correctly and the signature scheme is correct; (b) a replay failure in Line 14 cannot occur because txid_2 is fresh; (c) a conservation failure in Line 18 cannot occur because the immediately preceding transaction $\vec{\mathsf{tx}}'[-2]$ supplies sufficient balance.

Case 3: Receiving transaction: $\mathsf{send}(\mathsf{tx}_\mathsf{L}) = \mathbf{SC}$ and $\mathsf{rec}(\mathsf{tx}_\mathsf{L}) = \mathbf{MC}$. In this case, let $(\mathsf{tx}_2, \mathsf{tx}_1, \mathsf{tx}_\mathsf{L}) = \mathsf{plausibility\text{-}map}(\mathsf{tx}_\mathsf{L})$. The argument for $\mathsf{tx} = \mathsf{tx}_\mathsf{L}$ and $\mathsf{tx} = \mathsf{tx}_1$ is as in *Case 2*. For the case of $\mathsf{tx} = \mathsf{tx}_2$, the same argument as before holds for a signature validity and for replay protection. It suffices to show that the conservation law is not violated. This is established in Lemma 64 by the invariant that $\mathsf{pool}_{\mathsf{tx}_\mathsf{L}} = \mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}_\mathsf{L}}$ that holds prior to processing tx_2, as it is the first transaction of a triplet produced by $\mathsf{plausibility\text{-}map}$. As $\text{verifytx}_{\mathbf{MC}}(\mathsf{L}) = \mathsf{L}$ then therefore $\mathsf{pool}_{\mathsf{tx}_\mathsf{L}} - v \geq 0$ and so $\mathsf{balances}[\mathbf{SC}][\mathsf{pool}_{pk}]_{\mathsf{tx}_\mathsf{L}} - \mathit{amount} \geq 0$ and Algorithm 40 returns *true*.

All three cases result in a contradiction, concluding the proof. ☐

Lemma 66 (**SC** failure firewall)**.** *Consider any execution of the construction of Section 7.1.2 in which persistence holds for* **MC**. *For all slots t such that $\mathcal{S}_t = \{\mathbf{MC}\}$ we have that*

$$\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) \in \pi_{\{\mathbf{MC}\}}(\mathbb{V}_{\mathfrak{A}}) .$$

Proof. From the assumption that persistence holds, there exists some **MC** party P for which $\mathbf{L}^{P}_{\mathbf{MC}}[t] = \mathbf{L}^{\cup}_{\mathbf{MC}}[t]$. Additionally, $\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) = \mathbf{L}^{\cup}_{\mathbf{MC}}[t]$ due to the *partitioning* property. It suffices to show that there exists some $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$ such that $\pi_{\{\mathbf{MC}\}}(\vec{\mathsf{tx}}) = \mathbf{L}^{P}_{\mathbf{MC}}[t]$. Let $\vec{\mathsf{tx}} \leftarrow \mathsf{plausible}(\mathbf{L}^{P}_{\mathbf{MC}}[t])$. We have $\text{verifier}_{\mathbf{MC}}(\mathbf{L}^{P}_{\mathbf{MC}}[t]) = $ *true*, so apply Lemma 65 to obtain that $\vec{\mathsf{tx}} \in \mathbb{V}_{\mathfrak{A}}$.

To see that $\pi_{\{\mathbf{MC}\}}(\vec{\mathsf{tx}}) = \mathbf{L}^{P}_{\mathbf{MC}}[t]$, note that Algorithm 51 for input L includes all $\mathsf{tx} \in \mathsf{L}$ in the same order as in its input. Furthermore, all $\mathsf{tx} \in \vec{\mathsf{tx}}$ such that $\mathsf{tx} \notin \mathsf{L}$ have $\mathsf{lid}(\mathsf{tx}) = \mathbf{SC}$ and so are excluded from the projection. ☐

General Firewall Property

In preparation for establishing the full firewall property, we state the following simple technical lemma.

Lemma 67 (Honest subsequence)**.** *Consider any set S of $2k$ consecutive slots prior to slot t in an execution of an Ouroboros ledger* $\mathbf{L}$ *such that* $\mathbb{A}_{\mathsf{hm}}(\mathbf{L})[t]$ *holds. Then $k+1$ slots of S are honest, except with negligible probability.*

Sketch. If the adversary controlled at least k out of any $2k$ consecutive slots, he could use them to produce an alternative k-blocks long chain for this interval without any help from the honest parties, resulting in a violation of common prefix and hence persistence (cf. Lemma 62). ☐

We are now ready to prove our key lemma, showing that our construction satisfies the firewall property.

Lemma 68 (Firewall)**.** *For all PPT adversaries $\mathcal{A}$, the construction of Section 7.1.2 with a secure ATMS and a collision-resistant hash function satisfies the firewall property with respect to assumptions $\mathbb{A}_{\mathbf{MC}}, \mathbb{A}_{\mathbf{SC}}$ with overwhelming probability in k.*

Proof. Let $\mathcal{A}$ be an arbitrary PPT adversary against the firewall property, and $\mathcal{Z}$ be an arbitrary environment for the execution of $\mathcal{A}$. We will construct the following PPT adversaries:

1. $\mathcal{A}_1$ is an adversary against ATMS.

2. $\mathcal{A}_2$ is a collision adversary against the hash function.

We first describe the construction of these adversaries.

The adversary $\mathcal{A}_1$. $\mathcal{A}_1$ simulates the execution of $\mathcal{A}$ and $\mathcal{Z}$ and of two populations of maintainers for two blockchains, **MC** and **SC**, which run the protocol Π (either the **MC** or the **SC**-maintainer part respectively) and spawns parties according to the mandates of the environment $\mathcal{Z}$ as follows. For all parties that are spawned as **MC** maintainers, $\mathcal{A}_1$ generates keys internally by invoking the Gen algorithm of the ATMS scheme. For all parties that are spawned as **SC** maintainers, $\mathcal{A}_1$ uses the oracle $\mathcal{O}^{\mathsf{gen}}$ to produce the public keys vk_i.

Whenever $\mathcal{A}$ requests that a (block or transaction) signature in **SC** is created, $\mathcal{A}_1$ invokes its oracle $\mathcal{O}^{\mathsf{sig}}$ to obtain the respective signature to provide to $\mathcal{A}$. When $\mathcal{A}$ requests that a **MC** signature is created, $\mathcal{A}_1$ uses its own generated private key to sign by invoking the Sig algorithm of the ATMS scheme. If $\mathcal{A}$ requests the corruption of a certain party P^*, then $\mathcal{A}_1$ reveals P^*'s private key to $\mathcal{A}$ as follows: If P^* is a **MC** maintainer, then the secret key is directly available to $\mathcal{A}_1$, so it is immediately returned. Otherwise, if P^* is a **SC** maintainer, then $\mathcal{A}_1$ obtains the secret key of P^* by invoking the oracle $\mathcal{O}^{\mathsf{cor}}$.

For every time slot t of the execution, $\mathcal{A}_1$ inspects all pairs $(P_{\mathbf{MC}}, P_{\mathbf{SC}})$ of honest parties such that $P_{\mathbf{MC}}$ is a **MC** maintainer and $P_{\mathbf{SC}}$ is a **SC** maintainer such that $\mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t] = \mathsf{L}^{\cup}_{\mathbf{MC}}[t]$ and $\mathsf{L}^{P_{\mathrm{SC}}}_{\mathbf{SC}}[t] = \mathsf{L}^{\cup}_{\mathbf{SC}}[t]$ (if such parties exist). Let $\mathsf{L}_1 = \mathsf{L}^{P_{\mathrm{MC}}}_{\mathbf{MC}}[t]$ and $\mathsf{L}_2 = \mathsf{L}^{P_{\mathrm{SC}}}_{\mathbf{SC}}[t]$. The adversary obtains the stable portion of the honestly adopted chain, namely $\mathsf{C}_1 = \mathsf{C}^{P_{\mathrm{MC}}}[t][: -k]$ and the transactions included in C_1, namely L'_1 (note that $\mathsf{L}'_1 \neq \mathsf{L}_1$ if L'_1 contains certificate transactions). $\mathcal{A}_1$ examines whether $\mathsf{L} = \mathsf{merge}(\mathsf{L}_1, \mathsf{L}_2) \notin \mathbb{V}_{\mathfrak{A}}$, to deduce whether $\mathcal{A}$ has succeeded. Note that both the evaluation of merge on arbitrary states and the verification of inclusion in $\mathbb{V}_{\mathfrak{A}}$ are efficiently computable and hence $\mathcal{A}_1$ can execute them. If $\mathcal{A}_1$ is not able to find such a time slot t and parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$, it returns failure (in the latter part of this proof, we will argue that all $\mathcal{A}_1$ failures occur with negligible probability conditioned on the event that $\mathcal{A}$ is successful, unless $\mathcal{A}_2$ is successful).

Otherwise it obtains the minimum t for which this holds and the L for this t. Because of the *base property* of the validity language, we have that $\epsilon \in \mathbb{V}_{\mathfrak{A}}$ and therefore $\mathsf{L} \neq \epsilon$. Let L^* be the minimum prefix of L such that $\mathsf{L}^* \notin \mathbb{V}_{\mathfrak{A}}$ and let $\mathsf{tx} = \mathsf{L}^*[-1]$. If tx has $\mathsf{send}(\mathsf{tx}) \neq \mathbf{SC}$ or $\mathsf{lid}(\mathsf{tx}) \neq \mathbf{MC}$, then $\mathcal{A}_1$ returns failure. Now therefore $\mathsf{send}(\mathsf{tx}) = \mathbf{SC}$ and $\mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$ (and so $\mathsf{tx} \in \mathsf{L}_1$). Hence, tx references a certain certificate transaction, say tx'. Due to the algorithm executed by **MC** maintainers for validation, we will have that $\mathsf{tx}' \in \mathsf{L}'_1\{: \mathsf{tx}\}$.

Let $\vec{\mathsf{tx}}^*$ be the subsequence of L_1' containing all certificate transactions up to and including tx'. We will argue that there must exist some ATMS forgery among one of the certificate transactions in $\vec{\mathsf{tx}}^*$. $\mathcal{A}_1$ looks at every transaction $\mathsf{sc\_cert}_j \in \vec{\mathsf{tx}}^*$ (and note that it will correspond to a unique epoch e_j). $\mathsf{sc\_cert}_j$ contains a message $m = (j, \langle \mathsf{pending}_j \rangle, avk^j)$ and a signature σ_j. $\mathcal{A}_1$ extracts the epoch e_j in which $\mathsf{sc\_cert}_j$ was confirmed in C_1 (and note that we must have $j > 0$). $\mathcal{A}_1$ collects the public keys elected for the last $2k$ slots of epoch e_{j-1} according to the view of $P_{\mathbf{SC}}$ into a set keys_{j-1} and similarly for keys_j. $\mathcal{A}_1$ collects the pending cross-chain transactions of e_{j-1} according to the view of $P_{\mathbf{SC}}$ into $\mathsf{pending}_j'$, and creates the respective Merkle-tree commitment $\langle \mathsf{pending}_j' \rangle$. $\mathcal{A}_1$ checks whether the following *certificate violation* condition holds:

$$\begin{gathered}\mathsf{AVer}(m, avk^{j-1}, \sigma_j) \text{ and} \\ \mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1}) \text{ and} \\ \big(\neg\mathsf{ACheck}(\mathsf{keys}_j, avk^j) \vee \langle \mathsf{pending}_j \rangle \neq \langle \mathsf{pending}_j' \rangle\big)\end{gathered} \tag{7.1}$$

where avk^{j-1} is extracted from $\mathsf{sc\_cert}_{j-1}$ according to the view of $P_{\mathbf{SC}}$, unless $j = 1$ in which case avk^0 is known. If the condition (7.1) holds for no j then $\mathcal{A}_1$ returns failure, otherwise it denotes by j^* the minimum j for which (7.1) holds and outputs the tuple $(m, \sigma_{j^*}, avk^{j^*-1}, \mathsf{keys}^{j^*-1})$.

The adversary $\mathcal{A}_2$. Like $\mathcal{A}_1$, $\mathcal{A}_2$ simulates the execution of $\mathcal{A}$ including two populations of maintainers and spawns parties according to the mandates of the environment $\mathcal{Z}$. For *all* these parties, $\mathcal{A}_2$ generates keys internally. When $\mathcal{A}$ requests that a transaction is created, $\mathcal{A}_2$ provides the signature with its respective private key. If $\mathcal{A}$ requests the corruption of a certain party, say P^*, then $\mathcal{A}_2$ provides the respective private key to $\mathcal{A}$.

For every time slot t of the execution, $\mathcal{A}_2$ inspects all pairs of honest parties such that $P_{\mathbf{MC}}$ is a **MC** maintainer and $P_{\mathbf{SC}}$ is a **SC** maintainer such that $\mathsf{L}_{\mathbf{MC}}^{P_{\mathrm{MC}}}[t] = \mathsf{L}_{\mathbf{MC}}^{\cup}[t]$ and $\mathsf{L}_{\mathbf{SC}}^{P_{\mathrm{SC}}}[t] = \mathsf{L}_{\mathbf{SC}}^{\cup}[t]$ and obtains the variables $\mathsf{L}_1, \mathsf{L}_2, \mathsf{C}_1, \mathsf{L}_1'$ as before. $\mathcal{A}_2$ examines whether $\mathsf{L} = \mathsf{merge}(\mathsf{L}_1, \mathsf{L}_2) \notin \mathbb{V}_{\mathfrak{A}}$, to deduce whether $\mathcal{A}$ has succeeded. If $\mathcal{A}_2$ is not able to find such a time slot t and parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$, it returns failure. Let tx be as in $\mathcal{A}_1$. If $\mathsf{send}(\mathsf{tx}) \neq \mathbf{SC}$ or $\mathsf{lid}(\mathsf{tx}) \neq \mathbf{MC}$, then $\mathcal{A}_2$ returns failure. Then tx references a certain certificate transaction $\mathsf{sc\_cert}_j = (j, \langle \mathsf{pending}_j \rangle, avk^j, \sigma_j)$ and uses a Merkle tree proof π which proves the inclusion of tx in $\mathsf{pending}_j$. If $\mathsf{sc\_cert}_j \notin \mathsf{L}_1'$, then $\mathcal{A}_2$ returns failure. When $\mathsf{sc\_cert}_j$ was accepted by $P_{\mathbf{SC}}$, $\mathsf{pending}_j$ included a set of transactions $\vec{\mathsf{tx}}$ in the view of $P_{\mathbf{SC}}$. If $\mathsf{tx} \in \vec{\mathsf{tx}}$, then $\mathcal{A}_2$ returns failure. Otherwise, the Merkle tree $\langle \mathsf{pending}_j \rangle$ was constructed from $\vec{\mathsf{tx}}$, but a proof-of-inclusion π for $\mathsf{tx} \notin \vec{\mathsf{tx}}$ was created. From this proof, $\mathcal{A}_2$ extracts a hash collision and returns it.

Probability analysis. Define the following events:

- sc-forge$[t]$: $\mathcal{A}$ is successful at slot t, i.e., $\pi_{\mathfrak{A}}\left(\mathsf{merge}(\{\mathsf{L}_i^{\cup}[t] : i \in \mathcal{S}_t\})\right) \notin \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathfrak{A}})$.
- atms-forge: $\mathcal{A}_1$ finds an index j^* for which the condition (7.1) occurs.
- hash-collision: $\mathcal{A}_2$ finds a hash function collision.

Note that ledger states in the protocol only contain $\mathfrak{A}$-transactions, hence $\pi_{\mathfrak{A}}$ is the identity function and sc-forge$[t]$ is equivalent to $\mathsf{merge}\left(\{\mathbf{L}_i^{\cup}[t] \,:\, i \in \mathcal{S}_t\}\right) \notin \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathfrak{A}})$. We will now show that for every t, the probability Pr[sc-forge$[t]$] is negligible. We distinguish two cases:

Case 1: $\mathcal{S}_t = \{\mathbf{MC}, \mathbf{SC}\}$. In this case Persistence holds for both **MC** and **SC**, and $\pi_{\mathcal{S}_t}$ is the identity function. We deal with this case in two successive claims (both implicitly conditioning on being in Case 1). First we show that, if sc-forge$[t]$ occurs, then one of atms-forge, hash-collision occurs. Therefore applying a union bound, we will have that:

$$\Pr[\text{sc-forge}[t]] \leq \Pr[\text{atms-forge}] + \Pr[\text{hash-collision}]\ .$$

Second, we show that Pr[atms-forge] is negligible (and the negligibility of Pr[hash--collision] follows from our assumption that the hash function is collision resistant).

Claim 1a: sc-forge$[t] \Rightarrow$ atms-forge $\vee$ hash-collision.
Because persistence holds in both **MC** and **SC**, we know that there exist two parties $P_{\mathbf{MC}}, P_{\mathbf{SC}}$ such that at slot t we have that $\mathbf{L}_{\mathbf{MC}}^{P_{\mathrm{MC}}}[t] = \mathbf{L}_{\mathbf{MC}}^{\cup}[t]$ and $\mathbf{L}_{\mathbf{SC}}^{P_{\mathrm{SC}}}[t] = \mathbf{L}_{\mathbf{SC}}^{\cup}[t]$, respectively. Therefore sc-forge$[t]$ implies

$$\mathsf{merge}(\{\mathbf{L}_{\mathbf{MC}}^{P_{\mathrm{MC}}}[t], \mathbf{L}_{\mathbf{SC}}^{P_{\mathrm{SC}}}[t]\}) \notin \mathbb{V}_{\mathfrak{A}}\ .$$

Let $\mathsf{tx}, \mathsf{tx}'$ be as in the definition of $\mathcal{A}_1$. By Lemma 63 and using **MC** and **SC** persistence, tx will exist and be an **MC**-receiving transaction. Hence, $\mathsf{send}(\mathsf{tx}) = \mathbf{SC}$ and $\mathsf{rec}(\mathsf{tx}) = \mathsf{lid}(\mathsf{tx}) = \mathbf{MC}$. Therefore, tx' will also exist. If $\mathcal{A}_1$ finds the index j^* for which (7.1) is satisfied, then atms-forge has occured and the claim is established, so let us assume otherwise. Hence, for each certificate $\mathsf{sc\_cert}_j$ containing a message $m = (j, \langle\mathsf{pending}_j\rangle, avk^j)$, it holds that

$$\begin{gathered}\left(\mathsf{AVer}(m, avk^{j-1}, \sigma_j) \wedge \mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})\right) \\ \Rightarrow \\ \left(\mathsf{ACheck}(\mathsf{keys}_j, avk^j) \wedge \langle\mathsf{pending}_j\rangle = \langle\mathsf{pending}'_j\rangle\right)\ .\end{gathered} \tag{7.2}$$

Therefore, we have a chain of certificates, each of which is signed with a valid key avk^{j-1} and attests to the validity of the next key avk^j. For all of these certificates, $\mathsf{AVer}(m, avk^{j-1}, \sigma_j)$ holds, as it has been verified by $P_{\mathbf{MC}}$. Furthermore, by an induction argument (where the base case comes from the construction of avk^0 and the induction step follows from (7.2)) we have $\mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})$ as well.

As tx' is a certificate transaction which appears last in the above chain (with some index $\mathsf{sc\_cert}_k$), the above implication also holds for tx', and so does its premise $\mathsf{AVer}(m, avk^{k-1}, \sigma_k) \wedge \mathsf{ACheck}(\mathsf{keys}_{k-1}, avk^{k-1})$. Therefore, the conclusion of the implication $\langle\mathsf{pending}_k\rangle = \langle\mathsf{pending}'_k\rangle$ holds. However, the sending transaction corresponding to tx has been proven to belong to the Merkle Tree $\langle\mathsf{pending}_k\rangle$ (as verified by $P_{\mathbf{MC}}$), but does not belong to $\mathsf{pending}'_k$ (by the selection of tx). This constitutes a Merkle Tree collision, which translates to a hash collision. The construction of $\mathcal{A}_2$ outputs exactly this collision, and in this case we deduce that $\mathcal{A}_2$ is successful and hash-collision follows.

Claim 1b: Pr[atms-forge] is negligible.
Suppose that atms-forge occurs. We will argue that, in this case, $\mathcal{A}_1$ will have computed an ATMS forgery, which is a negligible event by the assumption that the used ATMS is secure.

From the assumption that atms-forge has occurred, at epoch e_j we have that $\mathsf{AVer}(m, avk^{j-1}, \sigma_j)$ and $\mathsf{ACheck}(\mathsf{keys}_{j-1}, avk^{j-1})$, but $\neg\mathsf{ACheck}(\mathsf{keys}_j, avk^j)$ or $\langle \mathsf{pending}_j \rangle \neq \langle \mathsf{pending}'_j \rangle$. From Lemma 67 and using $\mathbb{A}_{\mathsf{hm}}(\mathbf{SC})[t]$, we deduce that in the last $2k$ slots of epoch e_{j-1}, at least $k+1$ must be honest. Since e_j is the earliest epoch in which this occurs, this means that keys_{j-1} corresponds to the last $2k$ slot leaders of epoch e_{j-1}, and all honest parties agree on the same $2k$ slot leaders. Hence, in the ATMS game, the number of keys in keys corrupted by the adversary through the use of the oracle $\mathcal{O}^{\mathsf{cor}}(\cdot)$ is less than k. Furthermore, since $\neg\mathsf{ACheck}(\mathsf{keys}_j, avk^j)$ or $\langle \mathsf{pending}_j \rangle \neq \langle \mathsf{pending}'_j \rangle$, the message m contains either an invalid future aggregate key, an invalid Merkle Tree root of outgoing cross-chain transactions, or both. Hence, no honest party will sign the message m for this epoch and therefore $|Q^{\mathsf{sig}}[m]| = 0$. Hence $q < k$, and $\mathcal{A}_1$ wins the ATMS security game.

Case 2: $\mathcal{S}_t \neq \{\mathbf{MC}, \mathbf{SC}\}$. If $\mathbf{MC} \notin \mathcal{S}_t$ then, since $\mathbb{A}_{\mathbf{MC}}[t] \Rightarrow \mathbb{A}_{\mathbf{SC}}[t]$, we have $\mathcal{S}_t = \emptyset$ and $\neg$sc-forge$[t]$, as $\epsilon \in \mathbb{V}_{\mathfrak{A}}$ by the *base property*. It remains to consider the case $\mathcal{S}_t = \{\mathbf{MC}\}$. Using $\mathbf{MC}$ persistence, by Lemma 66 we obtain $\mathsf{merge}(\{\mathbf{L}^{\cup}_{\mathbf{MC}}[t]\}) \in \pi_{\{\mathbf{MC}\}}(\mathbb{V}_{\mathfrak{A}})$, and hence sc-forge$[t]$ did not occur.

From the two above cases, we conclude that for every t, $\Pr[\text{sc-forge}[t]] \leq \mathsf{negl}$. As the total number of slots is polynomial, we have shown that with overwhelming probability, we have that for all slots t and for all $\mathsf{A} \in \bigcup_{i \in \mathcal{S}_t} \mathsf{Assets}(\mathbf{L}_i)$, $\pi_{\mathsf{A}}(\mathsf{merge}(\{\mathbf{L}^{\cup}_i[t] : i \in \mathcal{S}_t\})) \in \pi_{\mathcal{S}_t}(\mathbb{V}_{\mathsf{A}})$, concluding the proof. □

Lemmas 62 and 68 together directly imply the following theorem.

Theorem 69 (Pegging Security). *In the synchronous setting with $2R$-semiadaptive corruptions, the construction of Section 7.1.2 using a secure ATMS and a collision resistant hash function is pegging secure with liveness parameter $u = 2k$ with respect to assumptions $\mathbb{A}_{\mathbf{MC}}$ and $\mathbb{A}_{\mathbf{SC}}$ defined above, and* merge*,* effect *and $\mathbb{V}_{\mathfrak{A}}$ defined in Section 7.1.2.*

7.1.4 The Diffuse Functionality

In the model described in Section 3.1 we employ the "Delayed Diffuse" functionality of [45], which we now describe in detail for completeness. The functionality is parameterized by $\Delta \in \mathbb{N}$ and denoted $\mathsf{DDiffuse}_\Delta$. It keeps rounds, executing one round per slot. $\mathsf{DDiffuse}_\Delta$ interacts with the environment $\mathcal{Z}$, stakeholders $U_1, \ldots, U_n$ and adversary $\mathcal{A}$, working as follows for each round: $\mathsf{DDiffuse}_\Delta$ maintains an incoming string for each party P_i that participates. A party, if activated, can fetch the contents of its incoming string, hence it behaves as a mailbox. Furthermore, parties can give an instruction to the functionality to diffuse a message. Activated parties can diffuse once per round.

When the adversary $\mathcal{A}$ is activated, it can: (a) read all inboxes and all diffuse requests and deliver messages to the inboxes in any order; (b) for any message m obtained via a diffuse request and any party P_i, $\mathcal{A}$ may move m into a special string $\mathsf{delayed}_i$ instead of the inbox of P_i. $\mathcal{A}$ can decide this individually for each message and each party; (c) for any party P_i, $\mathcal{A}$ can move any message from the string $\mathsf{delayed}_i$ to the inbox of P_i.

At the end of each round, the functionality ensures that every message that was either (a) diffused in this round and not put to the string $\mathsf{delayed}_i$ or (b) removed from the string $\mathsf{delayed}_i$ in this round is delivered to the inbox of party P_i. If a message currently present in $\mathsf{delayed}_i$ was originally diffused Δ slots ago, the functionality removes it from $\mathsf{delayed}_i$ and appends it to the inbox of party P_i.

Upon receiving $(\mathsf{Create}, U, \mathcal{C})$ from the environment, the functionality spawns a new stakeholder with chain $\mathcal{C}$ as its initial local chain (as in [89, 45]).

7.1.5 Adaptation to Other Proof-of-Stake Blockchains

Our construction can be adapted to work with other provably secure proof-of-stake blockchains: Ouroboros Praos [45], Ouroboros Genesis [13], Snow White [22], and Algorand [113]. Here we assume some familiarity with the considered protocols and refer the interested reader to the original papers for details.

Ouroboros Praos and Ouroboros Genesis

These protocols [45, 13] are strongly related and differ from each other only in the chain-selection rule they use, which is irrelevant for our discussion here, hence we consider both of the protocols simultaneously. Ouroboros Praos was shown secure in the semi-synchronous model with fully adaptive corruptions (cf. Section 3.1) and this result extends to Ouroboros Genesis. Despite sharing the basic structure with Ouroboros, they differ in several significant points which we now outline.

The slot leaders are elected differently: Namely, each party for each slot evaluates a verifiable random function (VRF, [48]) using the secret key associated with their stake, and providing as inputs to the VRF both the slot index and the epoch randomness. If the VRF output is below a certain threshold that depends on the party's stake, then the party is an eligible slot leader for that slot, with the same consequences as in Ouroboros. Each leader then includes into the block it creates the VRF output and a proof of its validity to certify her eligibility to act as slot leader. The probability of becoming a slot leader is roughly proportional to the amount of stake the party controls, however now it is independent for each slot and each party, as it is evaluated locally by each stakeholder for herself. This local nature of the leader election implies that there will inevitably be some slots with no, or several, slot leaders. In each epoch j, the stake distribution used in Praos and Genesis for slot leader election corresponds to the distribution recorded in the ledger up to the last block of epoch $j-2$. Additionally, the *epoch randomness* η_j for epoch j is derived as a hash of additional VRF-values included into blocks from the first two thirds of epoch $j-1$ for this purpose by the respective slot leaders. Finally, the protocols use *key-evolving signatures* for block signing, and in each slot the honest parties are mandated to update their private key, contributing to their resilience to adaptive corruptions.

Ouroboros Praos was shown [45] to achieve persistence and liveness under weaker assumptions than Ouroboros, namely:

1. Δ-semi-synchronous communication (where Δ affects the security bounds but is unknown to the protocol);
2. the majority of the stake is always controlled by honest parties;
3. the stake shift per epoch is limited.

In particular, Ouroboros Praos is secure in face of fully adaptive corruptions without any corruption delay. Ouroboros Genesis provides the same guarantees as Praos, as well as several other features that will not be relevant for our present discusion.

Construction of Pegged Ledgers. The main difference compared to our treatment of Ouroboros would be in the construction of the sidechain certificate (cf. Section 7.1.2). The need for a modification is caused by the private, local leader selection using VRFs in these protocols, which makes it impossible to identify the set of slot leaders for the suffix of an epoch at the beginning of this epoch, as done for Ouroboros.

The sidechain certificate included in **MC** at the beginning of epoch j would hence contain the following, for parameters Q and T specified below:

1. the epoch index;
2. a Merkle commitment to the list of withdrawals as in the case of Ouroboros;
3. a Merkle commitment to the **SC** stake distribution $\overline{\mathsf{SD}}_j$;
4. a list of Q public keys;
5. Q inclusion proofs (with respect to $\overline{\mathsf{SD}}_{j-1}$ contained in the previous certificate) and Q VRF-proofs certifying that these Q keys belong to slot leaders of Q out of the last T slots in epoch $j-1$;
6. Q signatures from the above Q public keys on the above; these can be replaced by a single aggregate signature to save space on **MC**.

The parameters Q and T have to be chosen in such a way that with overwhelming probability, there will be a chain growth of at least Q blocks during the last T slots of epoch $j-1$, but the adversary controls Q slots in this period only with negligible probability (and hence at least one of the signatures will have to come from an honest slot leader). The existence of such constants for $T = \Theta(k)$ was shown in [13].

While the above sidechain certificate is larger (and hence takes more space on **MC**) than the one we propose for Ouroboros, a switch to Ouroboros Praos or Genesis would also bring several advantages. First off, both constructions would give us security in the semi-synchronous model with fully adaptive corruptions (as shown in [45, 13]), and the use of Ouroboros Genesis would allow newly joining players to bootstrap from the mainchain genesis block only—without the need for a trusted checkpoint—as discussed extensively in [13].

Snow White

The high-level structure of Snow White execution is similar to the protocols we have already discussed: it contains epochs, committees that are sampled for each epoch based on the stake distribution recorded in the blockchain prior to that epoch, and randomness used for this sampling produced by hashing special nonce values included in previous blocks. Hence, our construction can be adapted to work with Snow White-based blockchains in a straightforward manner.

Algorand

Algorand does not aim for the so-called eventual consensus. Instead it runs a full Byzantine Agreement protocol for each block before moving to the next block, hence blocks are immediately finalized. Consider a setting with **MC** and **SC** both running Algorand. The main difficulty to address when constructing pegged ledgers is the continuous authentication of the sidechain certificate constructed by **SC**-maintainers for **MC** (other aspects, such as deposits from **MC** to **SC** work analogously to what we described above). As Algorand does not have epochs, and creating and processing a sidechain certificate for each block is overly demanding, a natural choice is to introduce a parameter R and execute this process only once every R blocks. Namely, every R blocks, the **SC**-maintainers produce a certificate that the **MC**-maintainers insert into the mainchain. This certificate most importantly contains:

1. a Merkle commitment to the list of withdrawals in the most recent R-block period;
2. a Merkle commitment to the full, most recent stake distribution $\overline{\mathsf{SD}}_j$ on **SC**;
3. a sufficient number of signatures from a separate committee certifying the above information, together with proofs justifying the membership of the signature's creators in the committee.

This additional committee is sampled from $\overline{\mathsf{SD}}_{j-1}$ (the stake distribution committed to in the previous sidechain certificate) via Algorand's private sortition mechanism such that the expected size of the committee is large enough to ensure honest supermajority (required for Algorand's security) translates into a strong honest majority within the committee. Note that the sortition mechanism also allows for a succinct proof of membership in the committee. The members of the committee then insert their individual signatures (signing the first two items in the certificate above) into the **SC** blockchain during the period of R blocks preceding the construction of the certificate. All the remaining mechanics of the pegged ledgers are a direct analogy of our construction above.

7.2 Bidirectional Sidechains with Work Sources

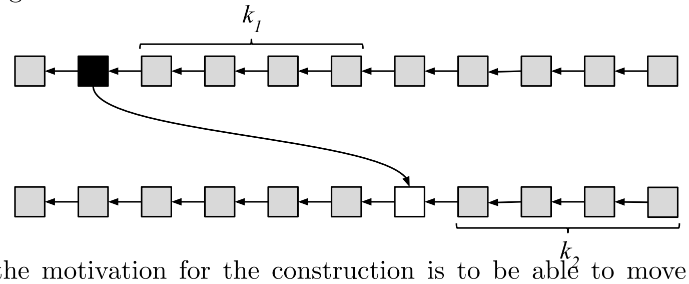
7.2.1 Sidechains as Smart Contracts

In this section, we introduce the first trustless construction for proof-of-work sidechains. We describe how to build generic communication between blockchains. As one application, we give the construction of a *two-way pegged* asset which can be moved from one blockchain to another while retaining its nature. We provide a high-level construction in Solidity. Our construction works across a broad range of blockchains requiring only two underlying properties. First, that the *source* blockchain is a proof-of-work blockchain supporting NIPoPoWs. Second, that the *target* blockchain is able to validate such proofs through smart contracts such as, e.g., Ethereum or Ethereum Classic. We give a formal proof of security of our construction via reduction to NIPoPoW security under the assumption that the interoperating blockchains are secure individually. To our knowledge, we are the first to provide such a construction in full and prove its security.

7.2.2 Smart Contract Workflow

We wish to transfer assets from one blockchain to another and then back. When assets can be transferred from one blockchain to another but not back, we call it a *one-way peg*. If assets can also be moved back, we call it a *two-way peg*. In each individual transfer of an asset, we have a particular *source blockchain*, from which the asset is moved, and a particular *target blockchain*, to which the asset is moved. In a sidechain setting of two blockchains that are two-way pegged, both blockchains can function as a source and a target blockchain for different transfers.

Figure 7.3: Basic information transfer between two blockchains



While the motivation for the construction is to be able to move assets from one blockchain to another, we generalize the notion of sidechains from this strict setting. In general, we would like the target blockchain to be able to react to any *event* that occurs on the source blockchain. Such events can be the fact that a transaction with a particular txid took place, that a certain account was paid a certain amount of money, or that a particular smart contract was instantiated. Our sidechain construction allows the target blockchain to react to events that took place on the source blockchain. This reaction can be implemented in its target blockchain smart contracts. We describe our construction in pseudocode similar to Ethereum' *Solidity*. In Solidity, *events* can be fired arbitrarily from within a smart contract and do not have a semantic interpretation. In this setting, events are defined by Solidity using the `event` type and have an *event name*, a *contract address* which fired them, as well as certain parameter values. A contract can elect to fire an event with any name and any parameters of its choice by invoking the emit command.

A high-level overview of cross-chain event transmission is shown in Figure 7.3. The process is as follows. First, an event is fired in the source blockchain, shown at the top. This could be any event that can be emitted using Ethereum's emit command. This event firing is caused by a certain transaction which is included at a certain block, indicated in black at the top. This block is then buried under k_1 subsequent blocks within the source blockchain, where the k_1 parameter is a security parameter of the scheme depending on the specific parameters of the source blockchain [60]. As soon as this confirmation occurs, the target blockchain can react to the event, shown at the bottom. This reaction occurs in a transaction which is included in a block within the target blockchain, illustrated in white. As usual, the block needs to be confirmed by waiting for k_2 blocks to be mined on top of it. It is possible that $k_1 \neq k_2$ because of different blockchain parameters such as a difference in block generation time or network synchrony. In this figure, arrows between blocks of the same blockchain indicate authenticated ancestry. The arrow between the two blockchains indicates the data transfer needed for the event.

Using this basic functionality of event information exchange between blockchains, we can construct two-way pegged sidechains. In such a construction, an asset that exists on one blockchain will gain the ability to be *moved* to a different blockchain and back. We will use the example of moving ether, the native asset of the Ethereum blockchain, from the Ethereum blockchain into the Ethereum Classic blockchain and back. Such an action is different from *exchanging* ether (ETH), the native token of the Ethereum blockchain, with ether classic (ETC), the native token of the Ethereum Classic blockchain. Instead, the asset retains its nature; it maintains its price and its ability to be used for the same purposes, while being governed by the rules of the new blockchain, such as different performance, fees, features, or security guarantees. Furthermore, no counterparty or market is required to perform the exchange; the transfer is something a party can do on its own.

7.2.3 Smart Contract Construction

Cross-chain certificates

For our construction, we use a primitive called Non-Interactive Proofs of Proof-of-Work recently introduced in [87]. Non-Interactive Proofs of Proofs-of-Work are cryptographic protocols which implement a *prover* and a *verifier*. The prover is a *full node* on the *source blockchain*. The verifier does not have access to that blockchain, but knows the source genesis block $\mathcal{G}$. The prover wants to convince the verifier that an *event* took place in the source blockchain; for instance, a smart contract method was called with certain parameters or that a payment was made into a particular address. Whether such an event took place can easily be determined if one inspects the whole blockchain. However, the prover wishes to convince the verifier by only sending a *succinct proof*, a short string which does not grow linearly with the size of the source blockchain, but, rather, *polylogarithmically*. The verifier must not be fooled by *adversarial provers* who provide incorrect proofs claiming that an event happened while in fact it didn't, or that it didn't while in fact it did. These adversaries can also mine blocks, but the honest parties are assumed to control the majority of computational power on both the source and the target blockchain networks. To withstand such attacks, the verifier accepts multiple proofs, at least one of which is assumed to have been honestly generated (this assumption is necessary in standard blockchain protocols in general [67, 153]). Comparing these proofs against each other, the verifier extracts a reliable truth value corresponding to the same value it would deduce if it were to be running a full node on the blockchain itself. This property is the *security* of NIPoPoWs proven in [87].

The NIPoPoWs construction talks about *predicates* evaluated on blockchains, but we are interested in *events*. We can translate from events to predicates provable with NIPoPoWs. Specifically, given a genesis block $\mathcal{G}$, a smart contract address addr, an event name Event, and a series of event parameter values $(\mathsf{param}_1, \mathsf{param}_2, \cdots, \mathsf{param}_n)$, the predicate e we wish to check for truth is the following: *Has the event named* Event *been fired with parameters* $(\mathsf{param}_1, \mathsf{param}_2, \cdots, \mathsf{param}_n)$ *by the smart contract residing in address* addr *on the blockchain with genesis block* $\mathcal{G}$ *at least* k *blocks ago?* This predicate is (1) *monotonic*, meaning that it starts with the value false and, if it ever becomes true, it cannot ever change its value back as the blockchain grows; (2) *infix-sensitive*, meaning that its truth value can be deduced by inspecting a polylogarithmically-bound number of blocks on the blockchain (in our case one block, within which the event firing was confirmed); and (3) *stable*, meaning that, if one party deduces that its value is true, then soon enough *all* parties will deduce that its value is true. This last property stems from the requirement that the event be buried under k blocks ensuring a blockchain reorganization up to k blocks ago cannot affect the predicate's value.

In order to determine whether an event took place, the NIPoPoW verifier function $\mathsf{verify}^{\mathcal{G},e}_{k,m}(\mathcal{P})$ accepts the event description in the form of a blockchain predicate e, which we gave above, the genesis block of the remote chain $\mathcal{G}$, as well as two security parameters k and m. These security parameters can be constants specified when the sidechain system is created (concrete values for these are given in [87]). Subsequently, the NIPoPoW verifier accepts a set of *proofs* $\mathcal{P} = \{\pi_1, \pi_2, \cdots, \pi_n\}$ which it compares and extracts a truth value for the predicate: Whether the event has taken place in the remote blockchain or not. As long as at least one *honestly generated* proof π_i is provided, the verifier's security ensures that the output will correspond to whether the event actually occurred.

Our protocol works as follows. Whenever an event of interest occurs on the source blockchain, the occurence of this event is observed by a source blockchain honest node, who generates a NIPoPoW about it. The target blockchain contains a smart contract with a method to accept and verify the veracity of this proof. The node can then submit the proof to the smart contract by broadcasting a transaction on the target blockchain. As soon as the proof is validated by the smart contract, the target blockchain can elect to react to the event as desired.

Adoption considerations. Our construction has certain prerequisites for both the source and the target blockchain before it can be adopted. In the case of bidirectionally connected blockchains, both of them must satisfy the source and the target blockchain prerequisites.

- **The source blockchain** needs to support *proofs* about it, which requires augmenting it with an *interlink* vector, the details of which can be found in [83]. This interlink vector can be added to a blockchain using a *user-activated velvet fork* [87, 158], which is performed without miner awareness and does not require a hard or soft fork. However, only events occuring *after* the velvet fork can be proven. New blockchains can adopt this from genesis.
- **The target blockchain** needs to be able to run the above verify function. This function can be programmed in a Turing-complete language such as Solidity. If the source blockchain proof-of-work hash function is available as an opcode or pre-compiled smart contract within the target blockchain's VM the way, e.g., Bitcoin's SHA256 hash function is available in Solidity, the implementation can be more gas-efficient.

Blockchain agnosticism. We underline the remarkable property that miners and full nodes of the target blockchain do not need to be aware of the source blockchain at all. To them, all information about the source blockchain is simply a string which is passed as a parameter to a smart contract and can remain *agnostic* to its semantics as a proof. Additionally, miners and full nodes of the source blockchain do not need to be aware of the target blockchain. Only the parties interested in facilitating cross-chain events must be aware of both. Those untrusted facilitators need to maintain an SPV node on the source blockchain about which they generate their NIPoPoW. To broadcast their proof on the target blockchain, they connect to target blockchain nodes and send the transaction containing the NIPoPoW. Blockchain agnosticism allows users to initiate cross-chain relationships between different blockchains *dynamically*, as long as the blockchains in question satisfy the above prerequisites.

Cross-chain events

We give our crosschain construction in Algorithm 52. Initially, our communication will be unidirectional. In the next section, we use two unidirectional channels to establish bidirectional communication. This smart contract runs on the target blockchain and informs it about events that took place in the source blockchain. It is parameterized by three parameters: k and m are the underlying security parameters of the NIPoPoW protocol. The value z is a *collateral* parameter, denominated in ether (or the native currency of the blockchain in which the execution takes place) and is used to incentivize honest participants to intervene in cases of false claims. The contract utilizes the NIPoPoW verify function parameterized by the event e, the remote genesis block $\mathcal{G}$ and the security parameters k and m. We do not give an explicit implementation of verify, as it can be implemented in a straightforward manner by translating the pseudocode listing of [87]. For our purposes, it suffices to treat it as a black box which, given a set of proofs, at least one of which is honestly generated, returns the truth value of the respective predicate.

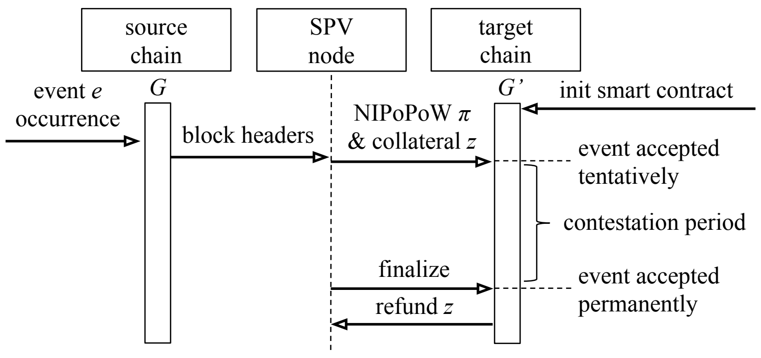
Algorithm 52 The smart contract skeleton that enables checking cross-chain proofs about events.

```
contract $\textsf{crosschain}_{k,m,z}$
    $\textsf{finalized-events} \leftarrow \emptyset$; $\textsf{events} \leftarrow \emptyset$
    internal function $\textsf{initialize}(\mathcal{G}_{\textsf{remote}})$
        $\mathcal{G} \leftarrow \mathcal{G}_{\textsf{remote}}$
    end function
    payable function $\textsf{submit-event-proof}(\pi, e)$
        if $\textsf{msg.value} < z$ then                          ▷ Ensure sufficient collateral
            return $\perp$
        end if
        if $\textsf{events}[e] = \perp \wedge \textsf{verify}^{e,\mathcal{G}}_{k,m}(\pi)$ then
            $\textsf{events}[e] \leftarrow \{\textsf{expire: block.number} + k, \textsf{proof: } \pi, \textsf{author: msg.sender}\}$
        end if
    end function
    function $\textsf{finalize-event(e)}$
        if $\textsf{events}[e] = \perp \vee \textsf{block.number} < \textsf{events}[e]\textsf{.expire}$ then
            return $\perp$
        end if
        $\textsf{finalized-events} \leftarrow \textsf{finalized-events} \cup \{e\}$
        $\textsf{author} \leftarrow \textsf{events}[e]\textsf{.author}$
        $\textsf{events}[e] \leftarrow \perp$
        $\textsf{author.send(z)}$                                  ▷ Return collateral
    end function
    function $\textsf{submit-contesting-proof}(\pi^*, e)$
        if $\textsf{events}[e] = \perp \vee \textsf{block.number} \geq \textsf{events}[e]\textsf{.expire}$ then
            return $\perp$
        end if
        if $\neg\textsf{verify}^{e,\mathcal{G}}_{k,m}(\textsf{events}[e]\textsf{.proof}, \pi^*)$ then     ▷ Original proof was fraudulent
            $\textsf{events}[e] \leftarrow \perp$
            $\textsf{msg.sender.send(z)}$                          ▷ Pay collateral to contester
        end if
    end function
    function $\textsf{event-exists(e)}$
        return $e \in \textsf{finalized-events}$
    end function
end contract
```

The contract allows detecting remote blockchain events and can be *inherited* by other contracts that wish to adopt its functionality. It works as follows. First, the initialize method is called exactly once to configure the contract, passing the *hash* of the genesis block of the remote chain which this contract will handle. This method is internal and can only be called by the contract inheriting from it. Users of the contract can check it has been configured with the correct genesis block prior to using it. We note that, while our algorithm does not reflect this to keep complexity low, it is possible to have a contract interact with *multiple* remote chains by extending it to include multiple geneses.

The lifecycle of an event submission is illustrated in Figure 7.4. When an event has taken place in the source blockchain, any source blockchain SPV node, the *author*, can inform the crosschain contract about this fact by generating a NIPoPoW π claiming that the event took place based on their current view of the source blockchain. This proof can then be submitted to the target blockchain by calling the submit-event-proof function and passing it the proof π and the event predicate e. The submission is accompanied by a collateral payment z. If the author is honest, this collateral will be returned to her later. The submit-event-proof function runs the NIPoPoW verify algorithm to check that the proof π is well-formed and that it claims that the predicate is true. It then stores the proof for later use. It also stores the address of the *author* and an *expiration block number*.

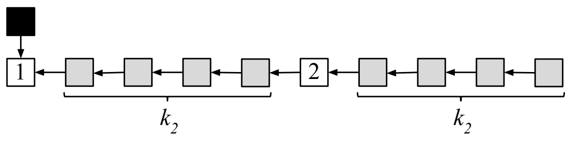
Figure 7.4: A sequence diagram showing the actions of the untrusted SPV node when communicating with both blockchain networks and the lifecycle of an event submission



Upon submission of a proof to the submit-event-proof function, the event is *tentatively accepted* for a *contestation period* of k blocks, during which any other party, the *contester*, can provide a counter-proof showing that the original proof was fraudulent. The contester can call the submit-contesting-proof function passing it the contesting proof π^* and the event predicate e. The function runs the NIPoPoW verify algorithm to compare the original proof events[e].proof against the contesting proof π^*. If the verification algorithm concludes that the original proof was fraudulent, the tentatively accepted event is abandoned and the collateral is paid to the contester.

Otherwise, when the contestation period has expired without any valid contestations, the author can call the finalize-event function. This function changes the acceptance of the event from tentative to *permanent* by including it in the finalized-events set and returns the collateral to the author. Finally, the event-exists function can be used by the inheriting contract to check if an event has been permanently accepted. The target blockchain state during this execution is shown in Figure 7.5. The source blockchain's event included in the black box, upon sufficient confirmation by k_1 blocks (not shown), is transmitted to the target blockchain at the bottom. The target blockchain includes the event *tentatively* in block 1 until a contestation period of k_2 has passed; the event is included *permanently* in block 2; subsequently, permanent inclusion needs to be confirmed with k_2 further blocks.

Figure 7.5: The target blockchain state during event submission



Two-way pegged sidechains

Having created the generic crosschain contract, we now build two-way pegged sidechains on top. For concreteness, we use the example of transferring ether (ETH), the native currency of the Ethereum blockchain, to the Ethereum Classic blockchain, and back. We note that this example is arbitrary and for illustration. Our construction can be used between any work-based blockchains satisfying the prerequisites detailed above.

When ether is moved to the Ethereum Classic blockchain, it will be represented as an ERC20 token[6] within Ethereum Classic. Let this custom token be called ETH20. The asset retains its nature as it moves from one blockchain to another if it is always possible to move ETH into ETH20 and back at a one-to-one rate. The economic reason is that the price of ETH and ETH20 on the market will necessarily be the same. If the price of ETH were to ever be significantly above the price of ETH20 in the market, then a rational participant would exchange their ETH20 for ETH using sidechains and sell their ETH on the market instead, and vice versa. There can be a small discrepancy in the two prices which stems from two different factors: First, the fees needed for a cross-chain transfer; and second, the temporary market fluctuations that can occur during the limited time needed to perform the cross chain transfer ($k_1 + 2k_2$). If we assume the price fluctuation (of ETH20 denominated in ETH) per unit of time is bounded, then the market price difference between ETH and ETH20 at any moment in time can be bounded by the sum of these two factors.

The sidechain smart contracts are presented in Algorithm 53. These smart contracts both extend the $\mathsf{crosschain}$ smart contract of Algorithm 52. Furthermore, $\mathsf{sidechain}_2$ also inherits basic $\mathsf{ERC20}$ functionality which allows token owners to transfer the token [138]. The $\mathsf{sidechain}_1$ contract will be instantiated on Ethereum, while the $\mathsf{sidechain}_2$ contract will be instantiated on Ethereum Classic. Suppose the genesis block hash of Ethereum is $\mathcal{G}_1$ and of Ethereum Classic is $\mathcal{G}_2$. We will use the genesis block hash of each blockchain as its unique identifier.

The two smart contracts both contain an $\mathsf{initialize}$ method which accepts the hash of the remote blockchain as well as the address of the remote smart contract it will interface with. Note that, while the two genesis hashes can be hard-coded into the respective smart contract code itself, the remote contract address cannot be built-in as a constant into the smart contract, but must be later specified by calling the $\mathsf{initialize}$ function. The reason is that, if $\mathsf{sidechain}_1$ were to be created on $\mathcal{G}_1$, it would require the address of $\mathsf{sidechain}_2$ to exist prior to its creation, and vice versa in a circular dependency. Therefore, the two contracts must first be created on their respective blockchain to obtain addresses, and then their $\mathsf{initialize}$ methods can be called to inform each contract about the address of the other.

[6]The ERC20 standard [150] defines an interface implementable by smart contracts that enables holding and transferring custom fungible tokens such as ICO tokens.

Specifically, first the contract $\mathsf{sidechain}_1$ is created on $\mathcal{G}_1$ to obtain its instance address which we also denote $\mathsf{sidechain}_1$. Then the second contract, $\mathsf{sidechain}_2$, is created on $\mathcal{G}_2$ to obtain its address $\mathsf{sidechain}_2$. Subsequently, the $\mathsf{initialize}$ function of $\mathsf{sidechain}_1$ is called, passing it $\mathcal{G}_2$ and the address $\mathsf{sidechain}_2$. Finally, $\mathsf{initialize}$ is called on $\mathsf{sidechain}_2$, passing it $\mathcal{G}_1$ and the address $\mathsf{sidechain}_1$. These initialization parameters are stored by the respective smart contracts for future use. As the $\mathsf{crosschain}$ contract requires, the $\mathsf{initialize}$ method can only be called once. Any user wishing to utilize this sidechain is expected to validate that the contracts have been set up correctly and that $\mathsf{initialize}$ has been called with the appropriate parameters.

Algorithm 53 An asset transferring contract between $\mathcal{G}_1$ and $\mathcal{G}_2$

```
contract sidechain_1 extends crosschain_{k,m,z}
    initialized ← false; ctr ← 0; claimed-events ← ∅
    function initialize(G_2, sidechain_2)
        if ¬initialized then
            ▷ Initialize with the remote chain genesis block
            crosschain.initialize(G_2)
            this.sidechain_2 ← sidechain_2; initialized ← true
        end if
    end function
    payable function deposit(target)
        ▷ Emit an event to be picked up by remote contract
        emit Deposited_1(target, msg.value, ctr++)
    end function
    function withdraw(amount, target, ctr)
        ▷ Validate that event took place on remote chain
        e ← (sidechain_2, Deposited_2, (amount, target, ctr))
        if e ∈ claimed-events ∨ ¬event-exists(e) then
            return ⊥
        end if
        claimed-events ← claimed-events ∪ {e}
        target.send(amount)
    end function
end contract
contract sidechain_2 extends crosschain_{k,m,z}; ERC20
    mapping(address ⇒ int) balances
    initialized ← false; ctr ← 0; claimed-events ← ∅
    function initialize(G_1, sidechain_1)
        if ¬initialized then
            crosschain.initialize(G_1)
            this.sidechain_1 ← sidechain_1; initialized ← true
        end if
    end function
    function deposit(target, amount)
        if balances[msg.sender] < amount then
            return ⊥
        end if
        balances[msg.sender] −= amount                ▷ Charge account of sender
        emit Deposited_2(target, amount, ctr++)
    end function
    function withdraw(amount, target, ctr)
        e ← (sidechain_1, Deposited_1, (amount, target, ctr))
        if e ∈ claimed-events ∨ ¬event-exists(e) then
            return ⊥
        end if
        claimed-events ← claimed-events ∪ {e}
        balances[target] += amount                     ▷ Credit target account
    end function
end contract
```

$\mathsf{sidechain}_1$ contains a $\mathsf{deposit}$ function which is *payable* in the native asset of Ethereum, ETH. When a user pays ETH into the $\mathsf{deposit}$ function, the funds are held by the smart contract and can later be used to pay parties who wish to *withdraw*, an operation performed by calling the $\mathsf{withdraw}$ function. $\mathsf{sidechain}_2$ contains similar $\mathsf{deposit}$ and $\mathsf{withdraw}$ functions which, however, do not pay in the native currency of Ethereum Classic, but instead maintain a $\mathsf{balance}$ mapping akin to a typical ERC20 implementation. The balance is updated when a user deposits or withdraws.

Moving funds from the Ethereum blockchain into the Ethereum Classic blockchain works as follows. First, the user pays with ETH to call the $\mathsf{deposit}$ function of $\mathsf{sidechain}_1$ which resides on $\mathcal{G}_1$, passing the target parameter which indicates their address in the Ethereum Classic blockchain that they wish to receive the money into. This call emits an event, $\mathsf{Deposited}_1$ which contains the necessary data: the target, the amount paid, as well as a nonce ctr to allow for future payments of the same amount to the same target. When the event has been emitted and buried under k_1 blocks within the Ethereum blockchain, the user produces an Ethereum NIPoPoW π_1 about the predicate e_1 which claims that the event $\mathsf{Deposited}_1$ has been emitted in blockchain $\mathcal{G}_1$ with the particular parameters by the contract residing at address $\mathsf{sidechain}_1$.

Subsequently, the user calls the $\mathsf{submit\text{-}event\text{-}proof}$ function of $\mathsf{sidechain}_2$ (which is inherited from the $\mathsf{crosschain}$ contract), passing the NIPoPoW π_1 and the event predicate e_1 and paying collateral z, which registers e_1 on $\mathsf{sidechain}_2$ as tentative. Because the user is honest, no adversary can produce a π_1^* which disproves their claim during the dispute period, and therefore the user waits for k_2 blocks for the contestation period to expire without any successful contestations. She then calls the $\mathsf{finalize\text{-}event}$ function for e_1 and receives back the collateral z, marking the event permanent. Finally, she calls the function $\mathsf{withdraw}$ of $\mathsf{sidechain}_2$, passing it the same parameters that e_1 was issued with. The $\mathsf{withdraw}$ function checks that e_1 exists using the $\mathsf{event\text{-}exists}$ method, which will return true. The user is then credited with amount in their ETH20 balance stored in $\mathsf{balances}[\mathsf{target}]$. This increment in balance creates brand new ETH20 tokens. The $\mathsf{withdraw}$ function also stores the signature of the event parameters that have been spent to avoid replay attacks, which is not shown here for algorithm brevity.

The user can then transfer their ETH20 tokens by utilizing the functionality inherited from the $\mathsf{ERC20}$ contract. When some (not necessarily the same) user is ready to move some (not necessarily the same) amount of ETH20 from the Ethereum Classic blockchain back into ETH on the Ethereum blockchain, they follow the reverse procedure: They call the $\mathsf{withdraw}$ function of $\mathsf{sidechain}_2$ which ensures their ERC20 balance is sufficient, deduces the requested amount, and fires an event e_2 as before. At this point, these particular ETH20 tokens are destroyed by the balance deduction. Once e_2 is confirmed in $\mathcal{G}_2$, the user produces the NIPoPoW π_2 about e_2 which claims a payment was made within $\mathcal{G}_2$. That proof is then submitted to $\mathsf{sidechain}_1$ by calling the $\mathsf{submit\text{-}event\text{-}proof}$ and $\mathsf{finalize\text{-}event}$ functions as before. Last, the user calls the $\mathsf{withdraw}$ function of $\mathsf{sidechain}_1$, which uses the $\mathsf{event\text{-}exists}$ function which will return true, finally paying back the user the respective amount of ETH. Because the only way to create ETH20 tokens in $\mathsf{sidechain}_2$ is by depositing ETH into $\mathsf{sidechain}_1$, there will always exist a sufficient balance of ETH owned by the $\mathsf{sidechains}_1$ smart contract to pay for any requested withdrawals.

Suppose now that an adversarial user makes a false claim that an event e took place in $\mathcal{G}_1$ and posts a relevant NIPoPoW π in $\mathcal{G}_2$. If an honest party is monitoring the chain $\mathcal{G}_2$ for the appearance of NIPoPoWs and the chain $\mathcal{G}_1$ for the firing of events, the fraudulence of π will be immediately obvious to them. They can subsequently generate a contesting NIPoPoW π^* providing a counter-claim that e did not occur. The honest party will broadcast this transaction at the beginning of the contestation period. Due to the *liveness* property of $\mathcal{G}_2$, the honest party will manage to include this transaction into $\mathcal{G}_2$ within one of the blocks before the end of the contestation period. The collateral z must be sufficient to incentivize an honest party to monitor $\mathcal{G}_1$ and $\mathcal{G}_2$ simultaneously, pay for transaction fees and ensure the time needed to generate a NIPoPoW π^* is small as compared to block generation time. The argument for $\mathcal{G}_2$ is analogous. We make this security argument formal in the next section.

7.2.4 Security of Smart Contract Implementation

We now formalize our protocol and provide a cryptographic analysis of its security. As NIPoPoWs security is modelled in the Bitcoin Backbone Protocol [60], we work in the same model (and note that the same mathematical model also captures Ethereum). See Chapter 2 for more details.

We assume that the standard results of the backbone protocol are attained, namely blockchain *persistence* and *liveness*. Persistence and liveness can be proved to hold with overwhelming probability under the honest mining majority assumption. For the details of that result, consult the Bitcoin Backbone paper [60].

We show sidechains security by illustrating that the definition from Section 79 is satisfied.

We will show that proving, to the maintainers of a chain $\mathcal{G}_2$, that an event e took place in chain $\mathcal{G}_1$ without it actually happening, can only occur if the underlying NIPoPoWs protocol is insecure. Therefore, our proof strategy follows the standard form of a cryptographic computational reduction. In our assumptions, we will make use of the persistence and liveness of $\mathcal{G}_2$, but only the persistence of $\mathcal{G}_1$.

Theorem 70 (Proof-of-Work Sidechains Security). *Assume a secure NIPoPoWs construction. Then, under the honest majority assumption for both $\mathcal{G}_1$ and $\mathcal{G}_2$, for all PPT adversaries $\mathcal{A}$ and for all environments $\mathcal{Z}$, the proof-of-work sidechains construction between $\mathcal{G}_1$ and $\mathcal{G}_2$ with contestation period $2k$ is secure, except with negligible probability in k.*

Proof. Let $\mathcal{A}$ be an arbitrary PPT adversary against the proof-of-work sidechains construction and $\mathcal{Z}$ be an arbitrary environment. We will construct an adversary $\mathcal{A}^*$ against NIPoPoWs and an environment $\mathcal{Z}^*$ in which it will operate.

Suppose, without loss of generality, that $\mathcal{A}$ can break the security of proof-of-work sidechains during a cross-chain transfer from $\mathcal{G}_1$ to $\mathcal{G}_2$. (Because the construction is symmetric, if the adversary is not able to do that, then they will be able to break the security of a cross-chain transfer from $\mathcal{G}_2$ to $\mathcal{G}_1$ and the proof follows in the same manner.)

Note that $\mathcal{A}$ works in an environment with two blockchains, $\mathcal{G}_1$ and $\mathcal{G}_2$, while $\mathcal{A}^*$ must work in the environment of one blockchain, namely $\mathcal{G}_1$.

$\mathcal{A}^*$ works as follows. First, it simulates the execution of the blockchain civilization $\mathcal{G}_2$. That is, it creates a new random oracle for $\mathcal{G}_2$ which is independent of its external random oracle used with $\mathcal{G}_1$. For any random oracle queries of $\mathcal{A}$ pertaining to $\mathcal{G}_1$, $\mathcal{A}^*$ forwards the queries to its external random oracle. For random oracles queries of $\mathcal{A}$ pertaining to $\mathcal{G}_2$, $\mathcal{A}^*$ answers its queries with its simulated and independent random oracle. Because $\mathcal{A}$ is subject to honest majority limitations in both $\mathcal{G}_1$ and $\mathcal{G}_2$, it follows that $\mathcal{A}^*$ will respect honest majority with regards to its external random oracle. For any environment instructions requested by $\mathcal{Z}$ pertaining to $\mathcal{G}_1$ (namely, the creation of new parties), the instructions are mirrored by $\mathcal{Z}^*$. Intructions of $\mathcal{Z}$ pertaining to $\mathcal{G}_2$ are simulated by $\mathcal{A}^*$. All diffusions of blocks in $\mathcal{G}_1$ by $\mathcal{A}$ are also diffused by $\mathcal{A}^*$, while diffusions in $\mathcal{G}_2$ by $\mathcal{A}$ are held private.

$\mathcal{A}^*$ monitors the chains adopted by honest parties and for every round r observes the state of all honest parties. $\mathcal{A}^*$ looks for a round r, an event e, a $\mathcal{G}_1$ maintainer p_1 and a $\mathcal{G}_2$ maintainer p_2 for which the following properties hold:

1. p_1 has not included e in their state

2. p_2 has included e in their finalized-events state

Because of the construction of p_2, finalized-events can contain e only if an issuance of submit-event-proof is included at least $2k$ blocks deep and contains the respective NIPoPoW π stored in events[e].proof. $\mathcal{A}^*$ now returns the proof π.

We will now analyze the probability of success of $\mathcal{A}$. Consider the following (probabilistic) events:

1. SC-Brk that $\mathcal{A}$ is successful

2. Cert-Brk that $\mathcal{A}^*$ is successful

3. Per_1 that persistence is maintained in $\mathcal{G}_1$

4. Per_2 that persistence is maintained in $\mathcal{G}_2$

5. Live_2 that liveness is maintained in $\mathcal{G}_2$

6. BC the union of $\text{Per}_1 \land \text{Per}_2 \land \text{Live}_2$

From total probability we obtain:

$$\Pr[\text{SC-Brk}] = \Pr[\text{SC-Brk}|\text{BC}]\Pr[\text{BC}] + \Pr[\text{SC-Brk}|\neg\text{BC}]\Pr[\neg\text{BC}]$$

From the honest majority assumption of $\mathcal{G}_1$, we deduce that $\Pr[\neg\text{Per}_1]$ and $\Pr[\neg\text{Live}_1]$ are negligible, and similarly from the honest majority assumption of $\mathcal{G}_2$ we deduce that $\Pr[\neg\text{Per}_2]$ is negligible, therefore $\Pr[\neg\text{BC}]$ is negligible. It now suffices to show that the probability $\Pr[\text{SC-Brk}|\text{BC}]\Pr[\text{BC}]$ is negligible.

Suppose that SC-Brk occurs. It follows that a (blockchain) event e must have been adopted by p_2 with some NIPoPoW π, but not by p_1, as detailed above. Suppose now that BC occurs.

Because of the *persistence* of $\mathcal{G}_2$, when π was burried under k blocks in the adopted chain of p_2, all honest parties in $\mathcal{G}_2$ must have seen π (this warrants the oldest k of the $2k$ blocks in the contestation period). Because of the *liveness* of $\mathcal{G}_2$, at least one honest block must have been included in the last k blocks after π had been received by all honest parties (this warrants the latest k of the $2k$ blocks in the contestation period).

Because of the *persistence* of $\mathcal{G}_1$, if e is not included in the state of p_1 at round r, then therefore it cannot have been included in the state of any $\mathcal{G}_1$ party during round $r - \eta k$. It follows that an honest party will attempt and succeed in generating a $\mathcal{G}_2$ block containing a contesting proof π^* attesting to the fraudulence of event e by invoking $\mathsf{submit\text{-}contesting\text{-}proof}(\pi^*, e)$ and this block will be adopted by p_2. As p_2 has finalized e, then therefore it must be such that $\mathsf{verify}_{k,m}^{e,\mathcal{G}}(\{\pi, \pi^*\})$, and therefore Cert-Brk has occurred.

Putting the above together, we obtain that:

$$\Pr[\text{Cert-Brk}] \geq \Pr[\text{SC-Brk}|\text{BC}] \Pr[\text{BC}]$$

From the NIPoPoW security assumption, we have that Pr[Cert-Brk] is negligible. Therefore, Pr[SC-Brk] is negligible. □

7.3 Unidirectionality with Proof-of-Burn

7.3.1 Burning Money

Since the dawn of history, humans have entertained the defiant thought of money burning, sometimes literally, for purposes ranging from artistic effect to protest, or to prevent it from falling into the hands of pirates [29, 98, 96, 42]. People did not shy away from the practice in the era of cryptocurrencies. Acts of money burning immediately followed the inception of Bitcoin [116] in 2009, with the first recorded instance of intentional cryptocurrency destruction taking place on August 2010 [140], a short three months after the first real-world transaction involving cryptocurrency in May 2010 [28]. For the first time, however, cryptocurrencies exhibit the unique ability for money burning to be provable retroactively in a so-called *proof-of-burn*.

First proposed by Iain Stewart in 2012 [139], proof-of-burn constitutes a mechanism for the destruction of cryptocurrency irrevocably and provably. The ability to create convincing proofs changed the practice of money burning from a fringe act to a rational and potentially useful endeavour. It has since been discovered that metadata of the user's choice can be uniquely ascribed to an act of burning, allowing each burn to become tailored to a particular purpose. Such protocols have been used as a consensus mechanism similar to proof-of-stake (Slimcoin [121]), as a mechanism for establishing identity (OpenBazaar [123, 160]), and for notarization (Carbon dating [43] and OpenTimestamps [145]). A particularly apt use case is the destruction of one type of cryptocurrency to create another. In one prolific case, users destroyed more than 2,130.87 BTC ($1.7M at the time, $21.6M in today's prices) for the bootstrapping of the Counterparty cryptocurrency [1].

While its adoption is undeniable, there has not been a formal treatment for proof-of-burn. Using the methods outlined in the previous sections and chapters, one can utilize proofs-of-burn to create a one-way peg: Money is burned on one chain to be created on another.

This section addresses the following:

(i) **Primitive definition.** Our definitional contribution introduces proof-of-burn as a cryptographic primitive for the first time. We define it as a protocol which consists of two algorithms, a burn address *generator* and a burn address *verifier*. We put forth the foundational properties which make for secure burn protocols, namely *unspendability*, *binding*, and *uncensorability*. One of the critical features of our formalization is that a tag has to be bound cryptographically with any proof-of-burn operation.

(ii) **Novel construction.** We propose a novel and simple construction which is flexible and can be adapted for use in existing cryptocurrencies, as long as they use public key hashes for address generation. To our knowledge, all popular cryptocurrencies are compatible with our scheme. We prove our construction secure in the Random Oracle model [19].

(iii) **Bootstrapping mechanism.** We propose a cryptocurrency proof-of-burn bootstrapping mechanism which does not require miners to connect to external blockchain networks. Our mechanism in principle allows burning from any proof-of-work-based cryptocurrency. This is what gives rise to one-way pegs.

(iv) **Experimental results.** We provide a compehensively tested production grade implementation of the bootstrapping mechanism in Ethereum written in Solidity, which we release as open source software. Our implementation can be used to consume proofs of burn of a source blockchain within a target blockchain. We provide experimental measurements for the cost of burn verification and find that, in current Ethereum prices, burn verification costs $0.28 per transaction. This allows coins burned on one blockchain to be consumed on another for the purposes of, for example, ERC-20 tokens creation [150].

Workflow. A user who wishes to burn her coins generates an address which we call a *burn address*. This address encodes some user-chosen metadata called the *tag*. She then proceeds to send any amount of cryptocurrency to the burn address. After burning her cryptocurrency, she proves to any interested party that she irrevocably destroyed the cryptocurrency in question.
Properties. We define the following properties for a proof-of-burn protocol:

- **Unspendability.** No one can spend the burned cryptocurrency.
- **Binding.** The burn commits only to a single tag.
- **Uncensorability.** Miners who do not agree with the scheme cannot censor burn transactions.

Finally, we consider the *usability* of a proof-of-burn protocol important: whether a user is able to create a burn transaction using her regular cryptocurrency wallet.

7.3.2 Defining Proof-of-Burn

Let κ be the security parameter.

Definition 81 (Burn protocol)**.** *A* burn *protocol* Π *consists of two functions* $\mathsf{GenBurnAddr}(1^\kappa, t)$ *and* $\mathsf{BurnVerify}(1^\kappa, t, \mathsf{burnAddr})$ *which work as follows:*

- $\mathsf{GenBurnAddr}(1^\kappa, t)$*: Given a tag t, generate a* burn address.
- $\mathsf{BurnVerify}(1^\kappa, t, \mathsf{burnAddr})$*: Given a tag t and an address* $\mathsf{burnAddr}$*, return* true *if and only if* $\mathsf{burnAddr}$ *is a burn address and correctly encodes t.*

The protocol works as follows. Alice first generates an address $\mathsf{burnAddr}$ to which she sends some cryptocurrency. The address encodes information contained in a tag t and is generated by invoking $\mathsf{GenBurnAddr}(1^\kappa, t)$. When the transaction is completed, she gives the transaction and tag to Bob who invokes $\mathsf{BurnVerify}(1^\kappa, t,$

burnAddr) to verify she irrevocably destroyed the cryptocurrency while committing to the provided tag.

We require that the burn scheme is *correct*.

Definition 82 (Correctness)**.** *A burn protocol* Π *is* correct *if for all* $t \in \{0,1\}^*$ *and for all* $\kappa \in \mathbb{N}$ *it holds that* $\mathsf{BurnVerify}(1^\kappa, t, \mathsf{GenBurnAddr}(1^\kappa, t)) = \mathit{true}$.

With foresight, we remark that the implementation of GenBurnAddr and BurnVerify will typically be deterministic, which alleviates the need for a probabilistic correctness definition.

Naturally, for GenBurnAddr to generate addresses that "look" valid but are unspendable according to the blockchain protocol requires that the burn protocol respects its format. We abstract the address generation and spending verification of the given system into a *blockchain address protocol*:

Definition 83 (Blockchain address protocol)**.** *A* blockchain address protocol Π_α *consists of two functions* GenAddr *and* SpendVerify*:*

- $\mathsf{GenAddr}(1^\kappa)$*: Returns a tuple* $(\mathsf{pk}, \mathsf{sk})$*, denoting the cryptocurrency address* pk *(a public key) used to receive money and its respective secret key* sk *which allows spending from that address.*
- $\mathsf{SpendVerify}(\mathsf{m}, \sigma, \mathsf{pk})$*: Returns* true *if the transaction m spending from receiving address pk has been authorized by the signature* σ *(by being signed by the respective private key).*

We note that, while the blockchain address protocol is not part of the burn protocol, the *security* properties of a burn protocol Π will be defined *with respect to* a blockchain address protocol Π_α.

These two functionalities are typically implemented using a public key signature scheme and accompanied by a respective signing algorithm. The signing algorithm is irrelevant for our burn purposes, as burning entails the inability to spend. As the format of m is cryptocurrency-specific, we intentionally leave it undefined. In both Bitcoin and Ethereum, m corresponds to transaction data. When a new candidate transaction is received from the network, the blockchain node calls SpendVerify, passing the public key pk, which is the address spending money incoming to the new transaction m, together with a signature σ, which signs m and should be produced using the respective secret key.

To state that the protocol generates addresses which cannot be spent from, we introduce a game-based security definition. The unspendability game spend-attack is illustrated in Algorithm 54.

Algorithm 54 The challenger for the burn protocol game-based security.

```
1: function spend-attack$_{\mathcal{A},\Pi}(\kappa)$
2:     $(t, m, \sigma, pk) \leftarrow \mathcal{A}(1^\kappa)$
3:     return $(\mathsf{BurnVerify}(1^\kappa, t, pk) \wedge \mathsf{SpendVerify}(m, \sigma, pk))$
4: end function
```

Definition 84 (Unspendability)**.** *A burn protocol* Π *is* unspendable *with respect to a blockchain address protocol* Π_α *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there exists a negligible function* negl *such that* $\Pr[\text{spend-attack}_{\mathcal{A},\Pi}(\kappa) = \mathit{true}] \leq \mathsf{negl}$.

Algorithm 55 The challenger for the burn protocol game-based security.

```
1: function bind-attack$_{\mathcal{A},\Pi}(\kappa)$
2:     $(t, t', \mathsf{burnAddr}) \leftarrow \mathcal{A}(1^\kappa)$
3:     return $(t \neq t' \wedge \mathsf{BurnVerify}(1^\kappa, t, \mathsf{burnAddr}) \wedge \mathsf{BurnVerify}(1^\kappa, t', \mathsf{burnAddr}))$
4: end function
```

It is desired that a burn address encodes one and only one tag. Concretely, given a burn address $\mathsf{burnAddr}$, $\mathsf{BurnVerify}(1^\kappa, t, \mathsf{burnAddr})$ should only evaluate to true for a single tag t. The game bind-attack in Algorithm 55 captures this property.

Definition 85 (Binding)**.** *A burn protocol* Π *is* binding *if for all probabilistic polynomial-time adversaries* $\mathcal{A}$ *there is a negligible function* negl *such that* $\Pr[\text{bind-attack}_{\mathcal{A},\Pi}(\kappa)] \leq \mathsf{negl}$.

We note here that the correctness and binding properties of a burn protocol are irrespective of the blockchain address protocol it was designed for.

We are now ready to define what constitutes a *secure proof-of-burn protocol.*

Definition 86 (Security)**.** *Let* Π *be a correct burn protocol. We say* Π *is* secure *with respect to a blockchain address protocol* Π_α *if it is* unspendable *and* binding *with respect to* Π_α.

The aforementioned properties form a good basis for a burn protocol. We observe that it may be possible to detect whether an address is a burn address. While this is desirable in certain circumstances, it allows miners to censor burn transactions. To mitigate this, we propose *uncensorability*, a property which mandates that a burn address is indistinguishable from a regular address if its tag is not known. During the execution of protocols which satisfy this property, when the burn transaction appears on the network, only the user who performed the burn knows that it constitutes a burn transaction prior to revealing the tag. Naturally, as soon as the tag is revealed, *correctness* mandates that the burn transaction becomes verifiable.

Definition 87 (Uncensorability)**.** *Let* $\mathcal{T}$ *be a distribution of tags. A burn protocol* Π *is* uncensorable *if the distribution ensembles* $\{(pk, sk) \leftarrow \mathsf{GenAddr}(1^\kappa); pk\}_\kappa$ *and* $\{t \leftarrow \mathcal{T}; pk \leftarrow \mathsf{GenBurnAddr}(1^\kappa, t); pk\}_\kappa$ *are computationally indistinguishable.*

7.3.3 Construction

We now present our construction for an uncensorable proof-of-burn protocol. To generate a burn address, the tag t is hashed and a perturbation is performed on the hash by toggling the last bit. Verifying a burn address $\mathsf{burnAddr}$ encodes a certain tag t is achieved by invoking $\mathsf{GenBurnAddr}$ with tag t and checking whether the result matches $\mathsf{burnAddr}$. If it matches, the $\mathsf{burnAddr}$ correctly encodes t. Our construction is illustrated in Algorithm 56.

Algorithm 56 Our uncensorable proof-of-burn protocol for Bitcoin P2PKH.

```
1: function GenBurnAddr_H(1^κ, t)
2:     $th \leftarrow H(t)$
3:     $th' \leftarrow th \oplus 1$                    ▷ Key perturbation
4:     return th'
5: end function
6: function BurnVerify_H(1^κ, t, th')
7:     return $(\mathsf{GenBurnAddr}_H(1^\kappa, t) = th')$
8: end function
```

We outline the blockchain address protocol for Bitcoin Pay to Public Key Hash (P2PKH) [2], with respect to which we prove our construction secure and uncensorable in Section 7.3.5. It is parametrized by a secure signature scheme S and a hash function H (for completeness, we give a construction which includes the concrete hash functions and checksums of Bitcoin in Appendix 7.3.8). GenAddr uses S to generate a keypair and hashes the public key to generate the public key hash. A tuple consisting of the public key hash and the secret key is returned. SpendVerify takes a spending transaction m, a scriptSig σ and a public key hash pkh. The scriptSig should contain the public key pk corresponding to pkh such that $H(pk) = pkh$ and a valid signature σ' for the spending transaction m [2]. If these conditions are met, the function returns true, otherwise it returns false. The blockchain address protocol is illustrated in Algorithm 57.

Algorithm 57 The Bitcoin P2PKH algorithm, parameterized by a signature scheme $S = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$.

```
1: function GenAddr_{S,H}(1^κ)
2:     $(pk, sk) \leftarrow \mathsf{Gen}(1^\kappa)$
3:     $pkh \leftarrow H(pk)$
4:     return (pkh, sk)
5: end function
6: function SpendVerify_{S,H}(m, σ, pkh)
7:     $(pk, \sigma') \leftarrow \sigma$
8:     return $(H(pk) = pkh \land \mathsf{Ver}(m, \sigma', pk))$
9: end function
```

7.3.4 Comparison

We now compare alternatives for proof-of-burn proposed in previous work. Our burn primitive captures all of these schemes.

`OP_RETURN`. Bitcoin provides a native opcode called `OP_RETURN` [18] which can be used for burning. Unfortunately, standard wallets do not provide a user friendly interface for creating `OP_RETURN` transactions. However, it benefits the Bitcoin network by allowing the UTXO to be pruned, at the cost of not being uncensorable. Similarly to `OP_RETURN`, any provably failing Bitcoin script can be used for burning [139].

P2SH `OP_RETURN`. An `OP_RETURN` or other provably failing script can also be used as the redeemScript for a Pay to Script Hash (P2SH) [4] address. It is unspendable since there is no scriptSig that could make the script succeed. Additionally, it is uncensorable if the tag is not revealed. Finally, the scheme is user friendly since any regular wallet can create a burn transaction.

Nothing-up-my-sleeve. An address is manually crafted, so that it is clear it was not generated from a regular keypair. For example, the all-zeros address is considered nothing-up-my-sleeve[7]. It is hard to obtain a public key hashing to this address, thus funds sent to it are unspendable. Because no metadata can be associated with such a burn, this scheme is not binding.

We compare the aforementioned schemes on whether they satisfy the burn protocol properties we define: Binding, unspendability and uncensorability. Additionally, we compare them based on how easily they translate to multiple cryptocurrencies. For instance, `OP_RETURN` and P2SH `OP_RETURN` rely on Bitcoin Script semantics and do not directly apply to any non-Bitcoin based cryptocurrencies like Monero [149], thus we say they are not *flexible*. The comparison is illustrated on Table 7.1.

Table 7.1: Comparison between proof-of-burn schemes.

	`OP_RETURN`	P2SH `OP_RETURN`	Nothing up my sleeve	$a \oplus 1$ **(this work)**
Binding	●	●		●
Flexible			●	●
Unspendable	●	●	●	●
Uncensorable		●	●	●
User friendly		●	●	●

7.3.5 Security of Burn

We now move on to the analysis of our scheme. As the scheme is deterministic, its correctness is straightforward to show.

Theorem 71 (Correctness)**.** *The proof-of-burn protocol* Π *of Section 7.3.3 is* correct.

Proof. Based on Algorithm 56, $\mathsf{BurnVerify}(1^\kappa, t, \mathsf{GenBurnAddr}(1^\kappa, t)) = \mathsf{true}$ if and only if $\mathsf{GenBurnAddr}(1^\kappa, t) = \mathsf{GenBurnAddr}(1^\kappa, t)$, which always holds as GenBurn-Addr is deterministic. □

We now state a simple lemma pertaining to the distribution of Random Oracle outputs.

[Perturbation] Let $p(\kappa)$ be a polynomial and $F : \{0,1\}^\kappa \longrightarrow \{0,1\}^\kappa$ be a permutation. Consider the process which samples $p(\kappa)$ strings $s_1, s_2, \ldots, s_{p(\kappa)}$ uniformly at random from the set $\{0,1\}^\kappa$. The probability that there exists $i \neq j$ such that $s_i = F(s_j)$ is negligible in κ.

We will now apply the above lemma to show that our scheme is unspenable.

Theorem 72 (Unspendability)**.** *If H is a* Random Oracle*, then the protocol* Π *of Section 7.3.3 is* unspendable.

[7]The Bitcoin address `1111111111111111111114oLvT2` encodes the all-zeros string and has received more than 50,000 transactions dating back to Aug 2010.

Proof. Let $\mathcal{A}$ be an arbitrary probabilistic polynomial time spend-attack adversary. $\mathcal{A}$ makes at most a polynomial number of queries $p(\kappa)$ to the Random Oracle. Let Match denote the event that there exist $i \neq j$ with $s_i = F(s_j)$ where $F(s) = s \oplus 1$.

If the adversary is successful then it has presented t, pk, pkh such that $H(pk) = pkh$ and $H(t) \oplus 1 = pkh$. Observe that spend-attack$_{\mathcal{A},\Pi}(\kappa) = \mathsf{true} \Rightarrow$ Match. Therefore Pr[spend-attack$_{\mathcal{A},\Pi}(\kappa)$] $\leq Pr$[Match]. Apply Lemma 7.3.5 on F to obtain Pr[spend-attack$_{\mathcal{A},\Pi}(\kappa)$] $\leq \mathsf{negl}$. □

We note that the security of the signature scheme is not needed to prove unspendability. Were the signature scheme of the underlying cryptocurrency ever found to be *forgeable*, the coins burned through our scheme would remain unspendable. We additionally remark that the choice of the permutation $F(x) = x \oplus 1$ is arbitrary. Any one-to-one function beyond the identity function would work equally well.

Preventing proof-of-burn. It is possible for a cryptocurrency to prevent proof-of-burn by requiring every address to be accompanied by a proof of possession [129]. To the best of our knowledge, no cryptocurrency features this.

Next, our binding theorem only requires that the hash function used is collision resistant and is in the standard model.

Algorithm 58 The collision adversary $\mathcal{A}^*$ against H using a proof-of-burn bind--attack adversary $\mathcal{A}$.

```
1: function $\mathcal{A}^*_{\mathcal{A}}(1^\kappa)$
2:     $(t, t', \_) \leftarrow \mathcal{A}(1^\kappa)$
3:     return (t, t')
4: end function
```

Theorem 73 (Binding)**.** *If H is a* collision resistant *hash function then the protocol of Section 7.3.3 is* binding*.*

Proof. Let $\mathcal{A}$ be an arbitrary adversary against Π. We will construct the Collision Resistance adversary $\mathcal{A}^*$ against H.

The collision resistance adversary, illustrated in Algorithm 58, calls $\mathcal{A}$ and obtains two outputs, t and t'. If $\mathcal{A}$ is successful then $t \neq t'$ and $H(t) \oplus 1 = H(t') \oplus 1$. Therefore $H(t) = H(t')$.

We thus conclude that $\mathcal{A}^*$ is successful in the collision game if and only if $\mathcal{A}$ is successful in the bind-attack game.

$$\Pr[\text{bind-attack}_{\mathcal{A},\Pi}(\kappa) = \mathsf{true}] = \Pr[\text{collision}_{\mathcal{A}^*,H}(\kappa) = \mathsf{true}]$$

From the collision resistance of H it follows that Pr[collision$_{\mathcal{A}^*,H} = \mathsf{true}$] $< \mathsf{negl}$. Therefore, Pr[bind-attack$_{\mathcal{A},\Pi} = \mathsf{true}$] $< \mathsf{negl}$, so the protocol Π is binding. □

We now posit that no adversary can predict the public key of a secure signature scheme, except with negligible probability. We call a distribution *unpredictable* if no probabilistic polynomial-time adversary can predict its sampling. We give the formal definition, with some of its statistical properties, in Appendix 7.3.9.

[Public key unpredictability] Let $S = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ be a secure signature scheme. Then the distribution ensemble $X_\kappa = \{(sk, pk) \leftarrow \mathsf{Gen}(1^\kappa); pk\}$ is unpredictable.

The following lemma shows that the output of the random oracle is indistinguishable from random if the input is unpredictable (for the complete proofs see

Appendix 7.3.9). For reference, the definition of computational indistinguishability is included in Appendix 7.3.9.

[Random Oracle unpredictability] Let $\mathcal{T}$ be an unpredictable distribution ensemble and H be a Random Oracle. The distribution ensemble $X = \{t \leftarrow \mathcal{T}; H(t)\}$ is indistinguishable from the uniform distribution ensemble $\mathcal{U}(\{0,1\}^\kappa)$.

Theorem 74 (Uncensorability)**.** *Let* $S = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ *be a* secure signature scheme*,* H *be a* Random Oracle*, and* $\mathcal{T}$ *be an unpredictable tag distribution. Then the protocol of Section 7.3.3 instantiated with* $H, S, \mathcal{T}$ *is* uncensorable*.*

Proof. Let X be the distribution ensemble of public keys generated using $\mathsf{GenAddr}$ and Y that of keys generated using $\mathsf{GenBurnAddr}$.

From Lemma 7.3.5 the distribution of public keys generated from S is unpredictable. The function $\mathsf{GenAddr}$ samples a public key from S and applies the random oracle H to it. Applying Lemma 7.3.5, we obtain that $X \approx_c \mathcal{U}(\{0,1\}^\kappa)$.

The function $H'(x) = H(x) \oplus 1$ is a random oracle (despite not being independent from the random oracle H). Since $\mathcal{T}$ is unpredictable, and applying Lemma 7.3.5 with random oracle H', we obtain that $Y \approx_c \mathcal{U}(\{0,1\}^\kappa)$.

By transitivity, X and Y are computationally indistinguishable. $\square$

From the above, we conclude that the tags used during the burn process must be unpredictable. If the tag is chosen to contain a randomly generated public key from a secure signature scheme, or its hash, Lemmas 7.3.5 and 7.3.5 show that sufficient entropy exists to ensure uncensorability. Our cross-chain application makes use of this fact.

7.3.6 Consumption

Over the last 5 years there has been an explosion of new cryptocurrencies. Unfortunately, it is hard for a new cryptocurrency to gain traction. Without traction, no market depth ensues and a cryptocurrency has difficulty getting listed in exchanges. But without being listed in exchanges, a cryptocurrency cannot gain traction.

This chicken-and-egg situation presents the need for a solution that circumvents exchanges and allows users to acquire the cryptocurrency directly. We propose utilizing proof-of-burn to allow users to obtain capital on a new cryptocurrency by burning a legacy cryptocurrency. In line with previous chapters, we call the legacy one the *source* and the new one the *target cryptocurrency*, and their blockchains the *source* and *target blockchain* respectively. The target blockchain may support burning from multiple source blockchains.

Workflow. A user who wishes to acquire a target cryptocurrency first forms a burn address valid in the source blockchain which encodes her receiving address on the target blockchain by using it as a tag. She then sends an amount of source cryptocurrency to that address. She submits a proof of this burn to a smart contract [35] on the target blockchain, where it is verified and she is credited an equivalent amount of target cryptocurrency on her receiving address. Proof-of-burn verification happens using NIPoPoWs in the work setting, or ATMs in the stake setting.

We define an event as a simple value transfer described by a transaction id txid, a receiving address addr and an amount amount. Simple value transfers are supported by all cryptocurrencies, allowing a verifier to process burns from a wide range of source blockchains. Note that this event type does not yet distinguish between burn and non-burn addresses.

For a verifier to be convinced that an event occurred on a source blockchain, they ensure its transaction is contained in a stable block in the best source chain. Specifically, the following data are supplied to the smart contract as a proof:

- tx: The transaction which contains the burn on the source blockchain.
- b: The block header for the block which contains tx.
- τ_{tx}: An inclusion proof showing tx $\in b$.
- τ_{b}: A proof that b is contained in the best (i.e., most proof-of-work) source blockchain and is stable.

We assume the source blockchain provides a function verify-tx(addr, amount, b, tx, τ_{tx}) which can be written in the smart contract language of the target blockchain and verifies the validity of a source blockchain transaction. It takes a source blockchain address addr, an amount of source cryptocurrency amount, a block b, a transaction tx and a proof τ_{tx} for the inclusion of tx in b. It returns true if tx contains a transfer of amount to addr and the proof τ_{tx} is valid for b.

The proof τ_{tx} is usually a Merkle Tree inclusion proof. More concretely, in Bitcoin, each block header contains a commitment to the set of transaction ids in the block in the form of a Merkle Tree root. Ethereum stores a similar commitment in its header — the root of a Merkle–Patricia Trie [151].

For verifying that a provided block b belongs to the best source blockchain and is stable, we assume the existence of a function in-best-chain(b). We explore how it can be implemented in the "Verifying block connection" paragraph below.

Bootstraping mechanism. Being able to verify events, we can grant target cryptocurrency to users who burn source cryptocurrency. After burning on the source blockchain, the user calls the claim function with the aforementioned event and a proof for it. This function ensures that the event provided is valid and has not been claimed before (i.e. no one has been granted target cryptocurrency for this specific event in the past), that it corresponds to the transaction tx and that the block b is stable, belongs to the best source chain and contains tx. Then, after verifying by invoking BurnVerify that the receiving address of the event is a burn address where the tag is the function caller's address, it releases the amount of coins burned in the form of an ERC-20 token. We present the contract burn-verifier with this capability in Algorithm 59.

Algorithm 59 A contract for verifying burns from the source chain. This smart contract runs within the target blockchain.

```
contract burn-verifier extends crosschain; ERC20
    mapping(address ⇒ uint256) balances
    claimed-events ← $\emptyset$
    function claim($e, b, \tau_{\text{tx}}$)
        block-ok ← in-best-chain($b$)
        tx-ok ← verify-tx($e$.addr, $e$.amount, $b$, $e$.tx, $\tau_{\text{tx}}$)
        event-ok ← $e \notin$ claimed-events
        if block-ok $\wedge$ tx-ok $\wedge$ event-ok $\wedge$ BurnVerify(msg.sender, $e$.addr) then
            claimed-events ← claimed-events $\cup \{e\}$
            balances[msg.sender] += $e$.amount
        end if
    end function
end contract
```

In the interest of keeping this implementation generic we assume that the user receives a token in return for his burn. However, instead of minting a token, the target cryptocurrency could allow the burn verifier contract to mint native cryptocurrency for any user who successfully claims an event. This would allow the target cryptocurrency to be bootstrapped entirely though burning as desired.

The problem of verifying a block belongs in the best source chain has been extensively addressed in the previous chapters. Here, we remark that there are multiple ways of implementing the aforementioned in-best-chain method.

Direct observation. Miners connect to the source blockchain network and have access to the best source chain. A miner can thus evaluate if a block is included in that chain and is stable. This mechanism does not provide miner-isolation. It is adopted by Counterparty.

NIPoPoWs. Verifying block connection can be achieved through NIPoPoWs. We remark that, as discussed in the previous chapter, with this setup a block connection proof may be considered valid provisionally, but there needs to be a period in which the proof can be disputed for the smart contract to be certain for the validity of the proof. Specifically, when a user performs a claim, they have to put down some collateral. If they have provided a valid NIPoPoW, a contestation period begins. Within that period a challenger can dispute the provided proof which – provided that the dispute is successful – would turn the result of in-best-chain to false, abort the claim and grant the challenger the user's collateral. If the contestation period ends with the proof undisputed, then in-best-chain evaluates to true, the collateral gets returned to the user and the claim is performed successfully.

Federation. A simpler approach is to allow a federation of n nodes monitoring the source chain to vote for their view of the best source chain.

Algorithm 60 A in-best-chain implementation which verifies that a block b is included in the best source chain using the federation mechanism. $\mathcal{M}$ denotes the latest MMR approved by the federation majority.

```
 1: votes ← ∅
 2: best-idx ← 0
 3: $\mathcal{M} \leftarrow \epsilon$
 4: function vote_fed($m, \sigma, pk$)
 5:     if $pk \in \text{fed} \wedge \mathsf{Ver}(m, \sigma, pk)$ then        ▷ Check that $pk$ is a valid federation member
 6:         $(\mathcal{M}^*, \mathsf{idx}) \leftarrow m$
 7:         $\mathsf{votes}[m] \leftarrow \mathsf{votes}[m] \cup \{pk\}$
 8:         if $|\mathsf{votes}[m]| \geq \lfloor \frac{|\text{fed}|}{2} \rfloor + 1 \wedge \mathsf{idx} > \mathsf{best\text{-}idx}$ then
 9:             $\mathcal{M} \leftarrow \mathcal{M}^*$                    ▷ Update accepted MMR
10:             best-idx ← idx
11:         end if
12:     end if
13: end function
14: function in-best-chain_$\mathcal{M}$($b, \tau_b$)
15:     return $\mathsf{Ver}\mathcal{MT}(\mathcal{M}, b, \tau_b)$
16: end function
```

The best source chain is expressed as the root $\mathcal{M}$ of a Merkle Tree containing the chain's stable blocks as leaves. Each federation node connects to both blockchain networks, calculates $\mathcal{M}$ and submits their vote for it every time a new source chain block is found. When a majority of $\lfloor \frac{n}{2} \rfloor + 1$ nodes agrees on the same $\mathcal{M}$, it is considered valid.

Having a valid $\mathcal{M}$, a verifier verifies a Merkle Tree inclusion proof τ_b for $b \in \mathcal{M}$ and is certain the block provided is part of the best source chain and is stable. This approach is illustrated in Algorithm 60.

The more suitable Merkle Mountain Range [32] data structure can be used to store $\mathcal{M}$ in place of regular Merkle Trees, as they constitute a more efficient append-only structure.

7.3.7 Empirical Results

In order to evaluate our consumption mechanisms, we implement the federated consumption mechanism in Solidity. We provide a concrete implementation of the burn-verifier contract described in Algorithm 59. We implement the crosschain parent contract from [90]. We verify transaction data by making use of the open source bitcoin-spv library [3]. Finally, the federation mechanism for verifying block connection is employed. The members of the federation can vote on their computed checkpoints using the vote function.

We release our implementation as open source software under the MIT license[8]. The implementation is production-ready and fully tested with 100% code coverage.

At the time of writing we obtain the median gas price of 6.9 gwei and the price of Ethereum in US Dollars at \$170.07. The cost of gas in USD is calculated by the formula $gas * 1.173483 * 10^{-6}$ rounded to two decimal places.

[8]`https://github.com/decrypto-org/burn-paper/tree/master/experiment`

Method	Gas cost	Equivalent in USD
vote	50103 gas	\$0.06
submit-event-proof	157932 gas	\$0.19
claim	78267 gas	\$0.09
Total claim cost	262817 gas	\$0.28

For the end user to prove an event and claim her burn, the cost is thus \$0.28. Comparatively, for a Bitcoin transaction to be included in the next block at the time of writing a user has to spend \$0.77.

7.3.8 Deployment to Bitcoin

Algorithm 61 The Bitcoin blockchain address protocol, including the engineering details of checksums and practical hash implementation.

```
1: function GenAddr()
2:     $(pk, sk) \leftarrow \mathsf{Gen}()$
3:     $pkh \leftarrow \texttt{RIPEMD160}(\texttt{SHA256}(\texttt{0x04} \| pk)$
4:     $\mathsf{addr} \leftarrow \texttt{0x00} \| pkh'$                ▷ Magic byte indicating mainnet
5:     $\mathsf{checksum} \leftarrow \texttt{SHA256}(\texttt{SHA256}(\mathsf{addr}))[:4]$     ▷ Keep the first 4 bytes
6:     return $\texttt{base58}(\mathsf{addr} \| \mathsf{checksum})$
7: end function
```

The scheme described above works for a generic P2PKH cryptocurrency and can be adapted to any cryptocurrency. We illustrate its suitability by giving a precise construction for Bitcoin, taking into account the engineering details that are behind the generation of a Bitcoin P2PKH address. A comparable approach can be used to generate Ethereum addresses or others.

The way Bitcoin generates P2PKH addresses is illustrated in Algorithm 61. Here, Gen generates an elliptic curve public key (of fixed key size $\kappa = 256$). After the elliptic curve public key is generated, it is marked by a magic number and subsequently hashed by the so-called HASH160 algorithm, which consists of evaluating RIPEMD160 on the SHA256 of the public key. The resulting hash is additionally prefixed by a magic number indicating that the execution is taking place on the main net (and not the test net), and the final address, together with a checksum, are encoded using base58 to obtain the final address.

Our burn algorithm follows the same structure for address generation, ensuring that the magic numbers and checksums validate correctly. In this construction, the hash function which is modelled as a random oracle is the HASH160 algorithm. The algorithm is illustrated in Algorithm 62 and works as follows. Given a tag t, the user derives a 160-byte hash $th = \texttt{RIPEMD160}(\texttt{SHA256}(t))$ which looks like a public key hash. The least significant bit of th is then flipped to achieve unspendability. This produces the 20-byte *perturbated hash th'*. The perturbated hash is then prefixed with 0x00 to designate that we're working on the Bitcoin mainnet as usual. The checksum is calculated and appended to it, and the result is base58 encoded into a Bitcoin address which correctly validates.

Algorithm 62 The key perturbation algorithm which generates a provable proof-of-burn address which validates under Bitcoin.

```
1: function GenBurnAddr($t$)
2:     $th \leftarrow$ RIPEMD160(SHA256($t$))
3:     $th' \leftarrow th \oplus$ 0x01                              ▷ Key perturbation
4:     addr $\leftarrow$ 0x00 $\|$ $th'$                        ▷ Magic byte indicating *mainnet*
5:     checksum $\leftarrow$ SHA256(SHA256(addr))[: 4]          ▷ Keep the first 4 bytes
6:     return base58(addr $\|$ checksum)
7: end function
```

7.3.9 Full proofs

In this section, we give the full proofs of our claims. Section 7.3.9 proves some facts about computationally indistinguishable distributions. In Section 7.3.9, we introduce unpredictable distributions and show that public keys are unpredictable. In Section 7.3.9, we prove some facts about random oracles, including Lemma 7.3.5 from which the unspendability of our scheme follows and Lemma 7.3.5 from which the uncensorability of our scheme follows.

Computational indistinguishability

Algorithm 63 The challenger for computational indistinguishability.

```
1: function dist-game$_{\mathcal{A},D_0,D_1}(\kappa)$
2:     $b \overset{\$}{\leftarrow} \{0,1\}$
3:     $z \leftarrow D_b$
4:     $b^* \leftarrow \mathcal{A}(z, 1^\kappa)$
5:     return $(b = b^*)$
6: end function
```

We review the definition of computational indistinguishability between two distributions X and Y. Define the cryptographic game illustrated in Algorithm 63. Computational indistinguishability mandates that no adversary can win the game, except with negligible probability.

Definition 88 (Computational indistinguishability)**.** *Two distribution ensembles* $\{X_\kappa\}_{\kappa\in\mathbb{N}}$ *and* $\{Y_\kappa\}_{\kappa\in\mathbb{N}}$ *are* computationally indistinguishable *if for every probabilistic polynomial-time adversary* $\mathcal{A}$*, there exists a negligible function* negl *such that* $\Pr[\text{dist-game}_{\mathcal{A},X,Y}(\kappa) = \textit{true}] < \mathsf{negl}$.

It is clear that applying an efficiently computable function to indistinguishable distributions preserves indistinguishability.

Algorithm 64 The distinguisher $\mathcal{A}^*$ between distributions X, Y which makes use of a distinguisher $\mathcal{A}$ between X' and Y'.

```
1: function $\mathcal{A}^*_{X,Y,f}(z, 1^\kappa)$
2:     return $\mathcal{A}(f(z))$
3: end function
```

Lemma 75 (Indistinguishability preservation)**.** *Given two computationally indistinguishable distribution ensembles $\{X_\kappa\}_{\kappa\in\mathbb{N}}$ and $\{Y_\kappa\}_{\kappa\in\mathbb{N}}$, let $\{f_\kappa\}_{\kappa\in\mathbb{N}}$, be an efficiently computable family of functions $X_\kappa \longrightarrow Y_\kappa$. Then the distribution ensembles $X' = \{f_\kappa(X_\kappa)\}_{\kappa\in\mathbb{N}}$ and $Y' = \{f_\kappa(Y_\kappa)\}_{\kappa\in\mathbb{N}}$ are computationally indistinguishable.*

Proof. Let $\mathcal{A}$ be a probabilistic polynomial-time distinguisher between X' and Y'. Consider the probabilistic polynomial-time distinguisher $\mathcal{A}^*$ between X and Y illustrated in Algorithm 64. Then $\Pr[\text{dist-game}_{\mathcal{A}^*}(\kappa) = \mathsf{true}] = \Pr[\text{dist-game}_{\mathcal{A}}(\kappa) = \mathsf{true}]$. As $\Pr[\text{dist-game}_{\mathcal{A}^*}(\kappa) = \mathsf{true}] \leq \frac{1}{2} + \mathsf{negl}$, therefore $\Pr[\text{dist-game}_{\mathcal{A}}(\kappa) = \mathsf{true}] \leq \frac{1}{2} + \mathsf{negl}$. □

Unpredictable distributions

We call a distribution ensemble *unpredictable* if no polynomial-time adversary can guess its output. The cryptographic predictability game is illustrated in Algorithm 65 and the security definition is given below.

Algorithm 65 The challenger for the distribution predictor.

```
1: function predict$_{\mathcal{A},X}(\kappa)$
2:     $x \leftarrow X$
3:     $x^* \leftarrow \mathcal{A}(1^\kappa)$
4:     return $(x = x^*)$
5: end function
```

Definition 89 (Unpredictable distribution)**.** *A distribution ensemble $\{X_\kappa\}_{\kappa\in\mathbb{N}}$ is* unpredictable *if for all probabilistic polynomial-time adversaries $\mathcal{A}$ there is a negligible function* negl *such that*

$$\Pr[\text{predict}_{\mathcal{A},X}(\kappa) = \textit{true}] < \mathsf{negl}\,.$$

We observe that, if each element of a distribution appears with negligible probability, then the distribution must be unpredictable.

Lemma 76 (Negligible unpredictability)**.** *Consider a distribution ensemble $\{X_\kappa\}_{\kappa\in\mathbb{N}}$ and a negligible function* negl. *If*

$$\max_{x\in[X_\kappa]} \Pr_{x^*\leftarrow X_\kappa}[x^* = x] \leq \mathsf{negl}\,,$$

then X is unpredictable.

Proof. Consider a probabilistic polynomial-time adversary $\mathcal{A}$ which predicts X_κ. The adversary is not given any input beyond 1^κ, hence the distribution of its output is independent from the choice of the challenger. Therefore

$$\Pr[\text{predict}_{\mathcal{A},X}(\kappa) = \mathsf{true}] = \sum_{x'\in[X]} \Pr_{x\leftarrow X}[\mathcal{A}(\kappa) = x' \wedge x = x'] =$$

$$\sum_{x'\in[X]} \Pr[\mathcal{A}(\kappa) = x'] \Pr_{x\leftarrow X}[x = x'] \leq \mathsf{negl} \sum_{x'\in[X]} \Pr[\mathcal{A}(\kappa) = x'] \leq \mathsf{negl}\,.$$

□

Finally, we observe that public keys generated from secure signature schemes must be unpredictable.

Algorithm 66 The existential forgery $\mathcal{A}$ which tries to guess the secret key through sampling.

```
1: function $\mathcal{A}_S(1^\kappa, pk)$
2:     $(pk, sk) \leftarrow \mathsf{Gen}(1^\kappa)$
3:     return $(\epsilon, \mathsf{Sig}(sk, \epsilon))$
4: end function
```

[Public key unpredictability] Let $S = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$ be a secure signature scheme. Then the distribution ensemble $X_\kappa = \{(sk, pk) \leftarrow \mathsf{Gen}(1^\kappa); pk\}$ is unpredictable.

Proof. Let $p = \max_{\widehat{pk} \in [X_\kappa]} \Pr_{pk \leftarrow X_\kappa}[pk = \widehat{pk}]$. Consider the existential forgery adversary $\mathcal{A}$ illustrated in Algorithm 66 which works as follows. It receives pk as its input from the challenger, but ignores it and generates a new key pair $(pk', sk') \leftarrow \mathsf{Gen}(1^\kappa)$. Since the two invocations of Gen are independent,

$$\begin{aligned} \Pr[pk = pk'] &\geq \max_{\widehat{pk} \in [X_\kappa]} \Pr[pk = \widehat{pk} \wedge pk' = \widehat{pk}] \\ &= \max_{\widehat{pk} \in [X_\kappa]} \Pr[pk = \widehat{pk}] \Pr[pk' = \widehat{pk}] \\ &= \max_{\widehat{pk} \in [X_\kappa]} \left(\Pr[pk = \widehat{pk}]\right)^2 = p^2 \,. \end{aligned}$$

The adversary checks whether $pk = pk'$. If not, it aborts. Otherwise, it uses sk' to sign the message $m = \epsilon$ and returns the forgery $\sigma = \mathsf{Sig}(sk, m)$. From the correctness of the signature scheme, if $pk = pk'$, then $\mathsf{Ver}(pk, \mathsf{Sig}(sk, m)) = \mathsf{true}$ and the adversary is successful. Since the signature scheme is secure, $\Pr[\mathsf{Sig\text{-}forge}^{cma}_{\mathcal{A},S}] = \mathsf{negl}$. But $\Pr[pk = pk'] \leq \Pr[\mathsf{Sig\text{-}forge}^{cma}_{\mathcal{A},S}]$ and therefore $p \leq \sqrt{\Pr[pk = pk']} \leq \mathsf{negl}$. Applying Lemma 76, we deduce that the distribution ensemble X_κ is unpredictable. □

Random Oracle properties

In this section, we state some statistical properties of the Random Oracle, which are useful for the proofs of our main results.

[Perturbation] Let $p(\kappa)$ be a polynomial and $F : \{0,1\}^\kappa \longrightarrow \{0,1\}^\kappa$ be a permutation. Consider the process which samples $p(\kappa)$ strings $s_1, s_2, \ldots, s_{p(\kappa)}$ uniformly at random from the set $\{0,1\}^\kappa$. The probability that there exists $i \neq j$ such that $s_i = F(s_j)$ is negligible in κ.

Proof. Let Match denote the event that there exist $1 \leq i \neq j \leq p(\kappa)$ such that $s_i = F(s_j)$. Let $\text{Match}_{i,j}$ denote the event that $s_i = F(s_j)$. Apply a union bound to obtain $\Pr[\bigcup_{i \neq j} \text{Match}_{i,j}] \leq \sum_{i \neq j} \Pr[\text{Match}_{i,j}]$. But $\Pr[\text{Match}_{i,j}] = 2^{-p(\kappa)}$ and therefore $\Pr[\text{Match}] \leq \sum_{i \neq j} 2^{-p(\kappa)} \leq p^2(\kappa) 2^{-p(\kappa)}$. □

Algorithm 67 The predictor $\mathcal{A}^*$ of the distribution X which makes use of a distinguisher $\mathcal{A}$ between X and $U(\{0,1\}^\kappa)$.

```
$i \leftarrow 0$
$Q \leftarrow \emptyset$                                ▷ Record of all random oracle queries
function $H'_H(x)$
    $i \leftarrow i + 1$
    $Q[i] \leftarrow H(x)$
    return $Q[i]$
end function
function $\mathcal{A}^*_{X,\mathcal{A}}(1^\kappa)$
    $b \overset{\$}{\leftarrow} \{0,1\}$
    if $b = 0$ then
        $z \leftarrow X$
        $j \overset{\$}{\leftarrow} [r]$
    else
        $z \leftarrow U(\{0,1\}^\kappa)$
    end if
    $b^* \leftarrow \mathcal{A}^{H'}(z)$
    if $b = 1 \vee j > i$ then
        return failure
    end if
    return Q[j]
end function
```

[Random Oracle unpredictability] Let $\mathcal{T}$ be an unpredictable distribution ensemble and H be a Random Oracle. The distribution ensemble $X = \{t \leftarrow \mathcal{T}; H(t)\}$ is indistinguishable from the uniform distribution ensemble $\mathcal{U}(\{0,1\}^\kappa)$.

Proof. Let $\mathcal{A}$ be an arbitrary polynomial distinguisher between X and $\mathcal{U}(\{0,1\}^\kappa)$. We construct an adversary $\mathcal{A}^*$ against predict$_\mathcal{T}$. Let r denote the (polynomial) maximum number of random oracle queries of $\mathcal{A}$. The adversary $\mathcal{A}^*$ is illustrated in Algorithm 67 and works as follows. Initially, it chooses a random bit $b \overset{\$}{\leftarrow} \{0,1\}$ and sets $Z = X$ if $b = 0$, otherwise sets $Z = \mathcal{U}(\{0,1\}^\kappa)$. It samples $z \leftarrow Z$. If $b = 0$, then z is chosen by applying GenAddr which involves calling the random oracle H with some input pk. It then chooses one of $\mathcal{A}$'s queries $j \overset{\$}{\leftarrow} [r]$ uniformly at random. Finally, it outputs the input received by the random oracle during the j^{th} query of $\mathcal{A}$.

We will consider two cases. Either $\mathcal{A}$ makes a random oracle query containing pk, or it does not. We will argue that, if $\mathcal{A}$ makes a random oracle query containing pk with non-negligible probability, then $\mathcal{A}^*$ will be successful with non-negligible probability. However, we will argue that, if $\mathcal{A}$ does not make the particular random oracle query, it will be unable to distinguish X from $\mathcal{U}(\{0,1\}^\kappa)$.

Let qry denote the event that $b = 0$ and $\mathcal{A}$ asks a random oracle query with input pk. Let x denote the random variable sampled by the challenger in the predictability game of $\mathcal{A}^*$. Let exqry denote the event that $b = 0$ and $\mathcal{A}$ asks a random oracle query with input equal to x. Observe that, since the input to $\mathcal{A}$ does not depend on x, we have that $\Pr[\text{exqry}] = \Pr[\text{qry}]$. As j is chosen independently of the execution of $\mathcal{A}$, conditioned on exqry the probability that $\mathcal{A}^*$ is able to correctly guess which query caused exqry will be $\frac{1}{r}$. Therefore we obtain that $\Pr[\text{predict}_{\mathcal{A}^*,\mathcal{T}}(\kappa) = \mathsf{true}] = \frac{1}{r}\Pr[\text{exqry}] = \frac{1}{r}\Pr[\text{qry}]$. As $\Pr[\text{predict}_{\mathcal{A}^*,\mathcal{T}}(\kappa) = \mathsf{true}] \leq \mathsf{negl}$ and r is polynomial in κ, we deduce that $\Pr[\text{qry}] \leq \mathsf{negl}$.

Consider the computational indistinguishability game depicted in Algorithm 63 in which the distinguisher gives a guess b^* attempting to identify the origin b of its input. If $b = 0$, then the distinguisher $\mathcal{A}$ receives a truly random input $pkh = H(pk)$. If the distinguisher does not query the random oracle with input pk, the input of the distinguisher is truly random and therefore $\Pr[b^* = 0|b = 0|\neg\text{qry}] = \Pr[b^* = 0|b = 1]$.

Consider the case where $b = 0$ and apply total probability to obtain

$$\begin{aligned}
&\Pr[b^* = 0|b = 0] = \\
&\Pr[b^* = 0|\text{qry}]\Pr[\text{qry}] + \Pr[b^* = 0|b = 0|\neg\text{qry}]\Pr[\neg\text{qry}] \\
\leq &\Pr[b^* = 0|\text{qry}]\Pr[\text{qry}] + \Pr[b^* = 0|b = 0|\neg\text{qry}] \\
\leq &\Pr[\text{qry}] + \Pr[b^* = 0|b = 0|\neg\text{qry}]
\end{aligned}$$

Then $\Pr[\text{dist-game}_{\mathcal{A},X,\mathcal{U}(\{0,1\}^\kappa)} = \mathsf{true}] = \Pr[b = b^*]$ is the probability of success of the distinguisher. Applying total probability we obtain

$$\begin{aligned}
\Pr[b = b^*] &= \Pr[b = b^*|b = 0]\Pr[b = 0] + \Pr[b = b^*|b = 1]\Pr[b = 1] \\
&= \frac{1}{2}(\Pr[b^* = 0|b = 0] + \Pr[b^* = 1|b = 1]) \\
&\leq \frac{1}{2}(\Pr[\text{qry}] + \Pr[b^* = 0|b = 0|\neg\text{qry}] + \Pr[b^* = 1|b = 1]) \\
&= \frac{1}{2}(\Pr[\text{qry}] + \Pr[b^* = 0|b = 1] + \Pr[b^* = 1|b = 1]) \\
&= \frac{1}{2}(\Pr[\text{qry}] + \Pr[b^* = 0|b = 1] + (1 - \Pr[b^* = 0|b = 1])) \\
&= \frac{1}{2}(1 + \Pr[\text{qry}]) \leq \frac{1}{2} + \mathsf{negl}
\end{aligned}$$

□

7.3.10 Relaxing the Random Oracle assumption

The construction presented above works for P2PKH and achieves its unspendability and uncensorability in the Random Oracle model. In this section, we discuss alternative constructions which work without requiring the Random Oracle model.

The simplest blockchain address protocol is the Pay to Public Key (P2PK) protocol which, in contrast to P2PKH does not hash the public key to generate an address. Instead, the address is literally the public key and spending verification simply checks the validity of a signature. This protocol is illustrated in Algorithm 68.

Algorithm 68 The blockchain address P2PK algorithm, parameterized by a signature scheme $S = (\mathsf{Gen}, \mathsf{Sig}, \mathsf{Ver})$.

```
1: function $\mathsf{GenAddr}_S(1^\kappa)$
2:     $(pk, sk) \leftarrow \mathsf{Gen}(1^\kappa)$
3:     return (pk, sk)
4: end function
5: function $\mathsf{SpendVerify}_S(m, \sigma, pk)$
6:     return $\mathsf{Ver}(m, \sigma', pk)$
7: end function
```

Without the Random Oracle model, our construction must be tailored to the signature scheme in order to ensure uncensorability, as our addresses must look similar to public keys generated by the scheme. We describe a burn scheme which can work for (EC)DSA signatures, as used in most cryptocurrencies today. Our scheme is unconditionally correct and binding in the standard model. We provide evidence of uncensorability in the Common Random String model, assuming the DLOG problem is hard and a collision resistant hash function exists. Additionally, we provide evidence that our scheme is unspendable in the Common Random String model and that no generic unspendable construction is possible in the standard model.

Initially, a κ-order multiplicative group $\mathbb{G}$ of order q and a generator g are selected and let the Common Random String be a random group element $h = g^y$ for some $y \in [q]$. Due to the self-reducibility of the DLOG problem, if DLOG is difficult in the group, an adversary will not be able to find the logarithm y of the random group element, except with negligible probability.

Our scheme is illustrated in Algorithm 69. $\mathsf{GenBurnAddr}$ hashes the tag t and treats $H(t)$ as the exponent, calculates the public key $g^{H(t)}$ and blinds it using the factor h. As before, $\mathsf{BurnVerify}$ regenerates the burn address from t and ensures it has been calculated correctly.

Algorithm 69 Our proof-of-burn protocol for P2PK using a Common Random String h representing a group element in which DLOG is difficult and parameterized by a collision resistant hash function H.

```
1: function $\mathsf{GenBurnAddr}_H(1^\kappa, t)$
2:     return $hg^{H(t)}$
3: end function
4: function $\mathsf{BurnVerify}_H(1^\kappa, t, th)$
5:     return $(\mathsf{GenBurnAddr}(1^\kappa, t) = th)$
6: end function
```

Correctness holds unconditionally.

Theorem 77 (Correctness)**.** *The proof-of-burn protocol* Π *of Algorithm 69 is* correct.

Proof. Based on Algorithm 69, $\mathsf{BurnVerify}(1^\kappa, t, \mathsf{GenBurnAddr}(1^\kappa, t)) = \mathsf{true}$ if and only if $\mathsf{GenBurnAddr}(1^\kappa, t) = \mathsf{GenBurnAddr}(1^\kappa, t)$, which always holds as GenBurnAddr is deterministic. □

As evidence towards unspendability, we now remark that it is difficult for an adversary to obtain the secret key corresponding to the public key $hg^{H(t)}$ needed to produce signatures. We therefore conjecture that our scheme is unspendable.

Algorithm 70 The random discrete log solver $\mathcal{A}^*$ which makes use of an adversary $\mathcal{A}$ which recovers the spending key corresponding to $hg^{H(t)}$.

```
1: function $\mathcal{A}^*_{\mathcal{A},H}(h)$
2:     $(t, z) \leftarrow \mathcal{A}(h)$
3:     return $z - H(t)$
4: end function
```

Lemma 78 (Logarithm ignorance)**.** *If h is a* Common Random String *and assuming the DLOG problem is hard, no probabilistic polynomial-time adversary can produce (t, z) such that $g^z = hg^{H(t)}$, except with negligible probability in κ.*

Proof. Suppose $\mathcal{A}$ is a probabilistic polynomial-time adversary which produces (t, z) with probability of success $p = \Pr[g^z = hg^{H(t)}]$. We construct the adversary $\mathcal{A}^*$ which invokes $\mathcal{A}$ illustrated in Algorithm 70 and finds the logarithm of h. Conditioned on the event that $\mathcal{A}$ is successful, we have that $g^z = hg^{H(t)} \Rightarrow g^z = g^{y+H(t)} \Rightarrow y \equiv z - H(t) \pmod{q}$, so $\mathcal{A}^*$ is successful. Therefore $\Pr[\mathcal{A}^*(h) = y] = p$. But $\Pr[\mathcal{A}^*(h) = y]$ is negligible. □

This observation illustrates the useful fact that, if a single group element with unknown logarithm is provided, an arbitrary number of such group elements can be found and logarithm ignorance can be proven.

Proofs-of-ignorance. There are other constructions which can give similar results. In fact, recent work on *proofs-of-ignorance* [46] has shown that any NP language can support proofs-of-ignorance, which are a prerequisite for our need of unspendability (as inability to produce signatures mandates ignorance of the private key). Therefore, we conjecture that such constructions are possible using any secure signature scheme in which the secret key constitutes a witness for the fact that the public key is an element of an NP language. Additionally, they argue that such constructions are not possible in the standard model given non-uniform probabilistic polynomial-time adversaries, supporting our construction in the Common Random String model. Whether burn constructions in the Standard Model exist against uniform probabilistic polynomial-time adversaries remains a question for future work.

Algorithm 71 The collision adversary $\mathcal{A}^*$ against H using a proof-of-burn bind--attack adversary $\mathcal{A}$.

```
1: function $\mathcal{A}^*_{\mathcal{A}}(1^\kappa)$
2:     $(t, t', \_) \leftarrow \mathcal{A}(1^\kappa)$
3:     return (t, t')
4: end function
```

Theorem 79 (Binding)**.** *If the hash function H is collision resistant and its range lies in $[q]$ where q denotes the group order of $\mathbb{G}$, then the proof-of-burn protocol Π of Algorithm 69 is* binding.

Proof. Let $\mathcal{A}$ be a probabilistic polynomial-time binding adversary against the protocol Π. We construct the probabilistic polynomial-time collision adversary $\mathcal{A}^*$ against the hash function H. The adversary $\mathcal{A}^*$ is illustrated in Algorithm 71 and works as follows. It invokes $\mathcal{A}$ which returns a triplet $(t, t', \mathsf{burnAddr})$, then returns the collision (t, t'). Let p denote the probability that $\mathcal{A}$ is successful.

Conditioned on the event that $\mathcal{A}$ is successful, it holds that $hg^{H(t)} = hg^{H(t')}$ and $t \neq t'$. This implies that $g^{H(t)} = g^{H(t')}$, which in turns yields $H(t) \equiv H(t') \pmod{q}$. Since the range of H lies in $[q]$, this constitutes a collision and $\mathcal{A}^*$ is successful.

We thus conclude that $\mathcal{A}^*$ is successful in the collision game if and only if $\mathcal{A}$ is successful in the bind-attack game.

$$\Pr[\text{bind-attack}_{\mathcal{A},\Pi} = \mathsf{true}] = \Pr[\text{collision}_{\mathcal{A}^*,H} = \mathsf{true}]$$

From the collision resistance of H it follows that $\Pr[\text{collision}_{\mathcal{A}^*,H} = 1] < \mathsf{negl}$. Therefore, $\Pr[\text{bind-attack}_{\mathcal{A},\Pi} = \mathsf{true}] < \mathsf{negl}$, so the protocol Π is binding. $\square$

We now give some evidence towards the uncensorability of our scheme. The following lemma expands on the results of Lemma 7.3.5 without making use of the Random Oracle model.

Algorithm 72 The collision adversary $\mathcal{A}$ against H which samples from an unpredictable distribution $\mathcal{T}$.

```
1: function $\mathcal{A}_{H,\mathcal{T}}(1^\kappa)$
2:     $t_1 \leftarrow \mathcal{T}$
3:     $t_2 \leftarrow \mathcal{T}$
4:     return $(t_1, t_2)$
5: end function
```

Lemma 80 (Collision resistant unpredictability). *Let H be a collision resistant hash function and $\{\mathcal{T}\}_{\kappa\in\mathbb{N}}$ be an efficiently samplable unpredictable distribution ensemble. Then the distribution ensemble $X_\kappa = \{t \leftarrow \mathcal{T}; H(t)\}$ is unpredictable.*

Proof. Consider the collision adversary $\mathcal{A}$ against the hash function H illustrated in Algorithm 72 which samples t_1 and t_2 independently from $\mathcal{T}_\kappa$ and hopes for a collision. Let Coll_{h^*} denote the event that $H(t_1) = H(t_2) = h^*$. Applying total probability

$$\begin{aligned}
&\max_{h^*\in[X_\kappa]} \Pr[\text{Coll}_{h^*}]\\
&= \max_{h^*\in[X_\kappa]} (\Pr[\text{Coll}_{h^*} | t_1 = t_2] \Pr[t_1 = t_2] + \Pr[\text{Coll}_{h^*} | t_1 \neq t_2] \Pr[t_1 \neq t_2])\\
&\leq \max_{h^*\in[X_\kappa]} \Pr[\text{Coll}_{h^*} | t_1 = t_2] \Pr[t_1 = t_2]\\
&\quad + \max_{h^*\in[X_\kappa]} \Pr[\text{Coll}_{h^*} | t_1 \neq t_2] \Pr[t_1 \neq t_2]\\
&\leq \max_{h^*\in[X_\kappa]} \Pr[t_1 = t_2] + \max_{h^*\in[X_\kappa]} \Pr[\text{Coll}_{h^*} | t_1 \neq t_2] \Pr[t_1 \neq t_2]\\
&= \Pr[t_1 = t_2] + \max_{h^*\in[X_\kappa]} \Pr[\text{Coll}_{h^*} \wedge t_1 \neq t_2]\,.
\end{aligned}$$

Therefore $\max_{h^* \in [X_\kappa]} \Pr[\mathrm{Coll}_{h^*} \wedge t_1 \neq t_2] \geq \max_{h^* \in [X_\kappa]} \Pr[\mathrm{Coll}_{h^*}] - \Pr[t_1 = t_2]$. We have that

$$\Pr[\mathrm{collision}_{\mathcal{A}}(\kappa) = \mathsf{true}] = \sum_{h^* \in [X_\kappa]} \Pr[\mathrm{Coll}_{h^*} \wedge t_1 \neq t_2]$$
$$\geq \max_{h^* \in [X_\kappa]} \Pr[\mathrm{Coll}_{h^*} \wedge t_1 \neq t_2] \geq \max_{h^* \in [X_\kappa]} \Pr[\mathrm{Coll}_{h^*}] - \Pr[t_1 = t_2]\,.$$

Since $\Pr[\mathrm{collision}_{\mathcal{A}}(\kappa) = \mathsf{true}] \leq \mathsf{negl}$ and $\Pr[t_1 = t_2] \leq \mathsf{negl}$, therefore

$$\max_{h^* \in [X_\kappa]} \Pr[\mathrm{Coll}_{h^*}] \leq \mathsf{negl}\,.$$

Because $H(t_1)$ and $H(t_2)$ are chosen independently,

$$\max_{h^* \in [X_\kappa]} \Pr_{x \leftarrow X_\kappa}[x = h^*] = \sqrt{\max_{h^* \in [X_\kappa]} \Pr[\mathrm{Coll}_{h^*}]} \leq \mathsf{negl}\,.$$

Applying Lemma 76, we deduce that the distribution ensemble X_κ is unpredictable. □

Unfortunately, a merely unpredictable distribution on the exponent does not allow us to prove uncensorability. However, we can prove uncensorability if we assume the hash function maps the tag distribution to the uniform distribution $\mathcal{U}([q])$ of the exponents of $\mathbb{G}$, which is an assumption closely related to the Random Oracle. We leave the relaxation of this additional assumption for future work.

Theorem 81 (Uncensorability)**.** *Let $\mathcal{T}$ be an efficiently samplable unpredictable tag distribution and H be a hash function such that $\{t \leftarrow \mathcal{T}_\kappa; H(t)\} \approx_c \mathcal{U}([q])$ where q denotes the order of the group $\mathbb{G}$. Then the proof-of-burn protocol Π of Algorithm 69 is* uncensorable *with respect to blockchain address protocol Π_α of Algorithm 68.*

Proof. Apply Lemma 75 to the computationally indistinguishable distribution ensembles $X = \{t \leftarrow \mathcal{T}_\kappa; H(t)\}$ and $Y = \mathcal{U}([q])$ mapped through the function $f(x) = g^x$. The resulting distributions, g^X and g^Y are indistinguishable. The distribution g^Y is the distribution of public keys generated by $\mathsf{GenAddr}$. As multiplication by h constitutes a permutation of the group, the distribution g^Y is identical to the distribution hg^Y. Hence g^X and hg^Y are indistinguishable. □

Trusted setup. We remark here that we *do not* require a trusted setup. In particular, for the selection of the protocol parameters, we do not generate a Common Reference String g^y by selecting a random y and computing g^y, as this would require ensuring y is destroyed. Instead, we select a random group element h directly, which is possible in many finite groups. As an example of such a construction in practice, a point can be selected on the secp256k1 elliptic curve by starting with an X coordinate corresponding to a well-known number such as $X = \texttt{SHA256}$("Whereof one cannot speak, thereof one must be silent") and incremented until a solution of the elliptic curve equation exists for Y, then taking the positive such Y and using the point $h = (X, Y)$.

Perturbation of group element labels. Yet another scheme that can potentially realize the above properties is the burn address generation by evaluating $(g^{H(t)}) + 1$, where the $+1$ does not pertain to the group operation, but operates on the label of the group element. For example, in the primed order group $\mathbb{Z}_p^*$, the $+1$ operation can be taken to be literally the next integer. Such a scheme is clearly correct and binding. Its uncensorability is comparable to our above scheme. Lastly, its unspendability, given appropriate restrictions ($t \neq 0$) seems to intuitively hold: It is hard to know the logarithm of both a group element and its next. Whether this is provable in the Generic Group Model or using appropriate hardness assumptions is left for future work.

Chapter 8

Conclusion

Throughout this thesis, we have presented effective consensus compression mechanisms for all decentralized blockchain protocols, in particular for both proof-of-work and proof-of-stake. We introduced the NIPoPoWs primitive for proof-of-work and the ATMS primitive for proof-of-stake. For proof-of-work, we presented the following variants:

- *Charity with goodness* in the static synchronous model, which achieves security against a $\frac{1}{2}$ adversary, but succinctness only optimistically.
- *Charity without goodness* which achieves both security and succinctness against a $\frac{1}{3}$ adversary in both the static and variable synchronous models and achieves both security and succinctness against a $\frac{1}{4}$ adversary in both the static and variable Δ-bounded delay model. Succinctness is achieved when difficulty does not decrease exponentially.
- *Distill*, which achieves comparable results to the above, with the additional assumption that difficulty is non-decreasing.

For the first among the above, we gave concrete security parameters obtained through experimental analysis and simulations.

We gave three important applications of our primitives:

- *Superlight clients*, which allow the construction of wallets that can synchronize faster than standard SPV wallets. The improvement is exponential for proof-of-work and constant for proof-of-stake.
- *Logarithmic space mining*, which allows the replacement of all proof-of-work miners of the protocol with logarithmic-space equivalents under the assumption that honest $\frac{1}{4}$ majority holds always. The improvement is exponential compared to standard miners for both state as well as communication complexity.
- *Blockchain interoperability*, which allows any variant of blockchains to communicate, namely work/work, work/stake and stake/stake. We gave constructions using native support (for stake) as well as smart contract-based constructions (for work) and showed how they can be used for generic information transfer. Finally, we leveraged them to construct one-way and two-way pegs.

In addition to improving efficiency of existing solutions, our constructions hint towards two avenues which are in need of improvement in the blockchain space more generally. The first avenue concerns *scalability*, a major current topic of research in the field. While most solutions have been focusing on layer-2 solutions such as Lightning [126], our solution of allowing multiple chains to interoperate and in particular our proposal to separate the notion of a *cryptocurrency* from its native *blockchain* allows sidechains to be used to offload transaction traffic off of a main chain and into multiple sidechains. As long as the majority of transactions remain within one chain and are not cross-chain transactions, sidechain solutions can improve the scalability of the main chain. One example means of ensuring sharded transaction traffic is to create one sidechain per particular industry or wide geographical location. The second avenue concerns *upgradability* and the trial of new features. While soft forks and hard forks require consensus change and may face opposition, sidechains can be used to trial out new features without requiring all of the main chain to upgrade to these new features. This can be useful for beta-testing, but also for adopting features that are considered more risky by the majority. The portion of the population willing to take the risk can move their capital to a novel sidechain, while the risk-averse majority can leverage the firewall property to protect their capital on the main chain.

Overall, our proposals have given rise to vibrant new research directions and have inspired solutions which are seeing practical adoption across the cryptocurrency space. Multiple production cryptocurrencies have adopted our protocols, among others ERGO, nimiq, and WebDollar. Lastly, our primitives have been implemented and extended by researchers in peer reviewed papers. One prominent example is FlyClient [32], which provides an alternative implementation to our NIPoPoW primitive.

8.1 Future Work

As we have seen, the consensus compression primitives have numerous important applications which can have significant impact in the space. For these to be deployable in practice with confidence, more research is needed around the primitives. Around the topic of proof-of-stake sidechains, the central question that remains is whether it is possible to do so succinctly, i.e., in $\mathcal{O}(polylog(|\mathsf{C}|))$. For NIPoPoWs, more research is needed to establish how much the model in which they achieve security can be relaxed.

We identify three major directions for future work, which we summarize below.

Velvet NIPoPoWs. In Chapter 3, we discussed various deployment strategies for NIPoPoWs and we pointed out that deployment can be made using a hard fork, a soft fork, or a *velvet fork*, and we described some algorithmic modifications to our protocols which make velvet forking possible. However, no proof of security was provided. In fact, velvet fork security is more complicated than what it seems on the surface. At first glance it seems that the adversary cannot benefit from including incorrect pointers: She can only include either altogether *wrong* pointers, or pointers that point to the correct superblock level but are not the most recent superblock. The first ones can be filtered out by the verified who can check the hash of the superblock pointer. As for the wrong ones, it seems that the adversary can only submit pointers to older blocks in the same chain, which would only harm their success rate. However, upon closer inspection one observes the following interesting attack: The adversary is able to include pointers not only to older superblocks of the correct level *on the same chain*, but also on different chains. These blocks, although they have necessary been created prior to the block in which their pointer is included, may not be ancestors of the block in question. This allows the adversary to play a game of *chain sewing* by "cutting" and "pasting" chunks of the honestly generated chain into their dishonest fork. This cutting and pasting allows the velvet adversary to "borrow" proof-of-work from an honest chain which does not belong to its adversarial fork, making it potentially win when compared against an honest fork. In short, the tree containing all interlink pointers does not look like a tree at all in the velvet case, but it forms a Directed Acyclic Graph. Patching the protocol and proving security under these settings is challenging and is explored in follow-up work [88].

NIPoPoWs under dishonest majority. While our analysis has used the assumption that honest majority holds for *all* rounds, it is a known result that full nodes in proof-of-work settings can withstand temporary dishonest majority [9]. More specifically, consider an execution in which honest majority holds *most* of the time, but a spike of dishonest majority occurs for a limited number of rounds. The short period of dishonest majority is then followed by a much longer period of honest majority. While the ledger property of persistence can be lost for the duration of the dishonest majority spike and for an additional period thereafter, the protocol is *self-healing* in that, given sufficient time during which honest majority holds, the protocol can recover and persistence will be restored. This stems from the fact that the *common prefix* property of chains is self-healing. A natural question that arises is whether NIPoPoWs are also self-healing. More specifically, in a temporary dishonest execution, a NIPoPoW verifier can be convinced to choose a chain that does not correspond to the chain that a full node honest party would choose. For example, that chain could be a short chain (such that the full node would reject it) with many superblocks (such that the NIPoPoW verifier would accept it). However, it seems that such proofs will become superseded by honestly generated proofs that correspond to a chain adopted by an honest full node if sufficient time time with honest majority is allowed to pass. A full proof of this security claim in the temporary dishonest model would significantly increase our confidence in superblock-based NIPoPoWs.

Bribing-resilient NIPoPoWs. One criticism of using superblocks to construct NIPoPoWs has been the susceptibility of these proofs to *bribing* [32]. The argument goes like this: While under the honest majority assumption the distribution of superblocks within chains will be as expected, the *real* protocol does not have honest majority. Instead, parties behave *rationally* and would be happy to deviate from the honest protocol if appropriate incentives are provided. The idea then is for the adversary to *bribe* miners in exchange for keeping high-level superblocks withheld. Because high-level blocks are rare, the money needed to bribe these miners will be small. More specifically, suppressing μ-level superblocks in expectation has a cost equal to the block reward multiplied by $|C|2^{-\mu}$ which is the number of blocks that the adversary wishes suppressed. In fact, the cost can be even less if not *all* μ superblocks are suppressed. In the case of our *charity with goodness* construction, such bribes can only harm the succinctness of the proofs, not their security. However, a harm in succinctness can translate to a harm in security depending on the application. For example, a cross-chain smart contract as disucussed in Chapter 7 has very limited gas available, and proofs that are $\omega(polylog(|\mathsf{C}|))$ would easily exhaust those limits. A failure of the smart contract to receive proofs would not only cause a denial of service, but can easily have financial cost incured by the victims. That cost can be sufficient to provide capital to an adversary for bribing miners. On the other hand, our *charity without goodness* and our *distill* constructions directly rely on good distribution of superblocks within the chains even for security, not just for succinctness. As such, a bribe can directly harm security, and it is not hard to imagine a rational adversary who makes sufficient money from their attack to be able to use some of it for bribing purposes. To fix this issue, miner rewards have to be rebalanced according to superblock levels. As such, a μ-level superblock must receive a reward equal to 2^{μ} times the reward of a 0-level block. This makes bribing for the purposes of block suppression as costly as bribing for suppressing the whole chain. Therefore, if the participants believe the chain itself to be secure and not bribable, the superblocks will also survive for the same reasons. In fact, it may be possible to allocate such rewards with a soft fork using a smart contract beneficiary. The exact mechanics needed for reward allocation to make NIPoPoWs bribing-resilient will be explored in future work.

8.2 Epilogue

Computer science is a data-driven science in which optimization according to some measurable metric or another always remains the main goal. In our case, we have optimized the space and communication complexity of blockchain consensus protocols, and this has given rise to important applications on top. In focusing on a narrow optimization problem, it is often easy to forget that our work has moral impact, and one has to keep in mind the moral character of cryptographic work [130]. In addition to the moral dilemmas of secrecy and transparency faced by our predecessor cryptographers who worked on secure messaging and digital signatures, as blockchain scientists we are facing broader ethical questions which stem from the fact that the protocols we design have the potential for enormous economic and political impact if they are ever to become mainstream. When I began this thesis four years ago, I was, perhaps naïvely, extremely excited about the democratization that blockchain protocols can bring to the world, from their promise to *bank the unbanked* to the elimination of the extravagant fees charged by private financial institutions.

Throughout the duration of this work, after studying and understanding the topics in depth and develping new protocols, some of that initial excitement faded and turned to partial disillusionment. This came especially through numerous discussions and research conducted together with my colleague Dimitris Karakostas and our findings on lack of blockchain egalitarianism [74] (which does not form part of the present work). As a new scientist, naturally it is often easy to dismiss legacy systems such as the existing monetary and banking system by focusing on their shortcomings instead of their advantages which one often overlooks. Despite more sober, I am still excited about the future that blockchains and decentralized protocols can bring if we make good use of them. We shall keep working on them with ethics in mind. Some big picture questions will keep coming up: Are our new protocols better than the legacy system, and in which ways? Do they lack in others? Most importantly, are we building systems which will form a net benefit for humankind and the less fortunate in our society? Do they preserve or improve upon egalitarianism and democracy, and in which ways exactly? These are not exact science questions. While everyone's answers might be different, it is imperative that we consider the questions and each of us makes their own judgement. For, in solving our mathematical equations and proving our theorems, we must not forget the real people that our work will impact.

Bibliography

[1] Counterparty. Available at: https://counterparty.io/. URL: https://counterparty.io/.

[2] Developer guide - bitcoin. Available at: https://bitcoin.org/en/developer-guide. URL: https://bitcoin.org/en/developer-guide.

[3] summa-tx/bitcoin-spv: utilities for bitcoin spv proof verification on other chains. Available at: https://github.com/summa-tx/bitcoin-spv/. URL: https://github.com/summa-tx/bitcoin-spv/.

[4] Gavin Andresen. BIP 0016: Pay to Script Hash. Available at: https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki, Jan 2012.

[5] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 375–392. IEEE, 2017.

[6] Georgia Avarikioti, Roman Brunner, Aggelos Kiayias, Roger Wattenhofer, and Dionysis Zindros. Structure and content of the visible Darknet. *arXiv preprint arXiv:1811.01348*, 2018.

[7] Georgia Avarikioti, Lukas Käppeli, Yuyi Wang, and Roger Wattenhofer. Bitcoin security under temporary dishonest majority. In *23rd Financial Cryptography and Data Security (FC)*. Springer, 2019.

[8] Georgia Avarikioti, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. Divide and scale: Formalization of distributed ledger sharding protocols. *arXiv preprint arXiv:1910.10434*, 2019.

[9] Zet Avarikioti, Lukas Käppeli, Yuyi Wang, and Roger Wattenhofer. Bitcoin security under temporary dishonest majority. In *International Conference on Financial Cryptography and Data Security*, pages 466–483. Springer, 2019.

[10] Zeta Avarikioti, EK Kogias, Roger Wattenhofer, and Dionysis Zindros. Brick: Asynchronous incentive-compatible payment channels. In *International Conference on Financial Cryptography and Data Security*. Springer, 2021.

[11] Zeta Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. Cerberus Channels: Incentivizing Watchtowers for Bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 346–366. Springer, 2020.

[12] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. 2014. `https://blockstream.com/sidechains.pdf`.

[13] Christian Badertscher, Peter Gaži, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 913–930, 2018.

[14] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Consensus redux: distributed ledgers in the face of adversarial supremacy. Technical report, Cryptology ePrint Archive, Report 2020/1021, 2020.

[15] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 324–356, 2017. `doi:10.1007/978-3-319-63688-7\_11`.

[16] Lear Bahack. Theoretical Bitcoin Attacks with less than Half of the Computational Power. *arXiv preprint arXiv:1312.7013*, 2013.

[17] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[18] Massimo Bartoletti and Livio Pompianu. An analysis of Bitcoin `OP_RETURN` metadata. In *International Conference on Financial Cryptography and Data Security*, pages 218–230. Springer, 2017.

[19] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[20] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *IACR Cryptol. ePrint Arch.*, 2018:46, 2018.

[21] Eli Ben-Sasson, Iddo Bentov, Ynon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Manuscript.(2017). Slides at https://people.eecs.berkeley.edu/~alexch/docs/pcpip_bensasson.pdf*, 2017.

[22] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. *IACR Cryptology ePrint Archive*, 2016:919, 2016. URL: `http://eprint.iacr.org/2016/919`.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 181–197. Springer, 2008.

[24] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 326–349, 2012. `doi:10.1145/2090236.2090263`.

[25] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 111–120, 2013. `doi:10.1145/2488608.2488623`.

[26] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[27] Joseph Bonneau. Hostile blockchain takeovers (short paper). In *International Conference on Financial Cryptography and Data Security*, pages 92–100. Springer, 2018.

[28] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 104–121. IEEE, 2015.

[29] Chris Brook. *K Foundation Burn a Million Quid*. Ellipsis Books, 1997.

[30] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[31] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[32] Benedikt Bünz, Lucianna Kiffer, Loi Luu, and Mahdi Zamani. Flyclient: Super-light clients for cryptocurrencies. 2020.

[33] Vitalik Buterin. Hard forks, soft forks, defaults and coercion, 2017. URL: `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`.

[34] Vitalik Buterin and Alex Van de Sande. Mixed-case checksum address encoding. jan. 2016. *URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-55.md*, 2016.

[35] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[36] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct 2001.

[37] Alexander Chepurnoy, Mario Larangeira, and Alexander Ojiganov. Rollerchain, a blockchain with safely pruneable full blocks. *arXiv preprint arXiv:1603.07926*, 2016.

[38] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A Cryptocurrency with Stateless Transaction Validation. *IACR Cryptology ePrint Archive*, 2018:968, 2018.

[39] Herman Chernoff. A career in statistics. *Past, Present, and Future of Statistical Science*, page 29, 2014.

[40] Herman Chernoff et al. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[41] Joseph Chow. BTC Relay. Available at: `https://github.com/ethereum/btcrelay`, Dec 2014. URL: `https://github.com/ethereum/btcrelay`.

[42] Cicero. *De Inventione*. 85 BC.

[43] Jeremy Clark and Aleksander Essex. Commitcoin: Carbon dating commitments with bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 390–398. Springer, 2012.

[44] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees. In *Nordic Conference on Secure IT Systems*, pages 199–215. Springer, 2016.

[45] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 10821 of *LNCS*, pages 66–98. Springer, Apr–May 2018. `doi:10.1007/978-3-319-78375-8_3`.

[46] Apoorvaa Deshpande and Yael Kalai. Proofs of ignorance and applications to 2-message witness hiding. *IACR Cryptology ePrint Archive*, 2018:896, 2018.

[47] Johnny Dilley, Andrew Poelstra, Jonathan Wilkins, Marta Piekarska, Ben Gorlick, and Mark Friedenbach. Strong federations: An interoperable blockchain solution to centralized third party risks. *CoRR*, abs/1612.05491, 2016. URL: `http://arxiv.org/abs/1612.05491`, `arXiv:1612.05491`.

[48] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *International Workshop on Public Key Cryptography*, pages 416–431. Springer, 2005.

[49] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[50] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure logging schemes and certificate transparency. In *European Symposium on Research in Computer Security*, pages 140–158. Springer, 2016.

[51] Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set. *IACR Cryptol. ePrint Arch.*, 2019:611, 2019.

[52] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Annual International Cryptology Conference*, pages 139–147. Springer, 1992.

[53] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, pages 436–454. Springer, 2014.

[54] Giulia Fanti, Leonid Kogan, Sewoong Oh, Kathleen Ruan, Pramod Viswanath, and Gerui Wang. Compounding of wealth in proof-of-stake cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*, pages 42–61. Springer, 2019.

[55] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[56] Marc Fischlin, Anja Lehmann, Thomas Ristenpart, Thomas Shrimpton, Martijn Stam, and Stefano Tessaro. Random oracles with (out) programmability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 303–320. Springer, 2010.

[57] Mark Friedenbach. Compact SPV proofs via block header commitments. Available at: `https://sourceforge.net/p/bitcoin/mailman/message/32111357/`, 2014. URL: `https://sourceforge.net/p/bitcoin/mailman/message/32111357/`.

[58] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications (revised 2019). Cryptology ePrint Archive, Report 2014/765, 2014. `https://eprint.iacr.org/2014/765`.

[59] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin Backbone Protocol with Chains of Variable Difficulty (revised 2020). Unpublished Manuscript, 2020.

[60] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, volume 9057 of *LNCS*, pages 281–310. Springer, Apr 2015. `doi:10.1007/978-3-662-46803-6_10`.

[61] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Annual International Cryptology Conference*, volume 10401 of *LNCS*, pages 291–323. Springer, Aug 2017.

[62] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools.* Cambridge university press, 2007.

[63] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge university press, 2009.

[64] David Graeber. *Debt: The First 5,000 Years.* Melville House, 2014.

[65] Sandra M Hedetniemi, Stephen T Hedetniemi, and Arthur L Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18(4):319–349, 1988.

[66] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *Network and Distributed System Security Symposium*, 2017.

[67] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, pages 129–144, 2015.

[68] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

[69] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub*, 2016.

[70] Julian Hosp, Toby Hoenisch, and Paul Kittiwongsunthorn. COMIT: Cryptographically-secure Off-chain Multi-asset Instant Transaction network. Available at: `https://www.comit.network/doc/COMIT%20white%20paper%20v1.0.2.pdf`, 2017. URL: `https://www.comit.network/doc/COMIT%20white%20paper%20v1.0.2.pdf`.

[71] Geoffrey Ingham. Money is a social relation. In Steve Fleetwood, editor, *Critical realism in economics: Development and debate*, pages 104–105. Routledge, 1998.

[72] Hudson Jameson. Renaming suicide opcode. *URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-6.md*, 2015.

[73] William Stanley Jevons. *Money and the Mechanism of Exchange*. H.S. King & Co., 1875.

[74] Dimitris Karakostas, Aggelos Kiayias, Christos Nasikas, and Dionysis Zindros. Cryptocurrency egalitarianism: a quantitative approach. 2019.

[75] Kostis Karantias. Enabling NIPoPoW Applications on Bitcoin Cash. Master's thesis, University of Ioannina, Ioannina, Greece, 2019.

[76] Kostis Karantias. Sok: A taxonomy of cryptocurrency wallets. Technical report, IACR Cryptology ePrint Archive, 2020: 868, 2020.

[77] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. Compact storage of superblocks for nipopow applications. In *The 1st International Conference on Mathematical Research for Blockchain Economy*. Springer Nature, 2019.

[78] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. Proof-of-burn. In *International Conference on Financial Cryptography and Data Security*, 2019.

[79] Kostis Karantias, Aggelos Kiayias, and Dionysis Zindros. Smart contract derivatives. In *The 2nd International Conference on Mathematical Research for Blockchain Economy*. Springer Nature, 2020.

[80] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography.* CRC press, 1996.

[81] Rami Khalil and Arthur Gervais. Nocust–a non-custodial 2 nd-layer financial intermediary. Technical report, Cryptology ePrint Archive, Report 2018/642. `https://eprint.iacr.org/2018/642`, 2018.

[82] Aggelos Kiayias, Peter Gaži, and Dionysis Zindros. Proof-of-stake sidechains. In *IEEE Symposium on Security and Privacy.* IEEE, IEEE, 2019.

[83] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-Panagiota Stouka. Proofs of proofs of work with sublinear complexity. In *International Conference on Financial Cryptography and Data Security*, pages 61–78. Springer, Springer, 2016.

[84] Aggelos Kiayias, Nikos Leonardos, and Dionysis Zindros. Mining in Logarithmic Space. Unpublished Manuscript, 2020.

[85] Aggelos Kiayias, Nikos Leonardos, and Dionysis Zindros. Variable Difficulty Non-Interactive Proofs of Proof-of-Work. Unpublished Manuscript, 2020.

[86] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A composable security treatment of the lightning network. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 334–349. IEEE, IEEE, 2020.

[87] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-Interactive Proofs of Proof-of-Work. In *International Conference on Financial Cryptography and Data Security.* Springer, 2020.

[88] Aggelos Kiayias, Andrianna Polydouri, and Dionysis Zindros. The Velvet Path to Superlight Blockchain Clients. 2020.

[89] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In Jonathan Katz and Hovav Shacham, editors, *Annual International Cryptology Conference*, volume 10401 of *LNCS*, pages 357–388. Springer, Springer, Aug 2017.

[90] Aggelos Kiayias and Dionysis Zindros. Proof-of-work sidechains. In *International Conference on Financial Cryptography and Data Security: Workshop on Trusted Smart Contracts.* Springer, Springer, 2019.

[91] Neal Koblitz. CM-curves with good cryptographic properties. In *Annual international cryptology conference*, pages 279–287. Springer, 1991.

[92] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[93] Georgios Konstantopoulos. Plasma Cash: Towards more efficient Plasma constructions. *arXiv preprint arXiv:1911.12095*, abs/1612.05491, 2019. URL: `https://arxiv.org/pdf/1911.12095.pdf`.

[94] 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), Washington DC*, June 10-11 2013.

[95] Yujin Kwon, Hyoungshick Kim, Jinwoo Shin, and Yongdae Kim. Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash? In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 935–951. IEEE, 2019.

[96] Diogenes Laertius. *Lives of the Eminent Philosophers, Book II*, chapter 8. 3rd Century AD.

[97] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019.

[98] Steven E Landsburg. *The Armchair Economist (revised and updated May 2012): Economics & Everyday Life*. Simon and Schuster, 2007.

[99] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, page 33, 2012.

[100] Ben Laurie, Adam Langley, and Emilia Kasper. Rfc6962: Certificate transparency. *Request for Comments. IETF*, 2013.

[101] Charlie Lee. Litecoin. Available at: `https://litecoin.org`, 2011. URL: `https://litecoin.org`.

[102] Sergio Demian Lerner. Drivechains, sidechains and hybrid 2-way peg designs, 2016. `https://docs.rsk.co/Drivechains_Sidechains_and_Hybrid_2-way_peg_Designs_R9.pdf`.

[103] Yehuda Lindell and Jonathan Katz. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2014.

[104] Orfeas Stefanos Thyfronitis Litos and Dionysis Zindros. Trust is Risk: A Decentralized Financial Trust Platform. In *International Conference on Financial Cryptography and Data Security*, pages 340–356. Springer, 2017.

[105] Andreas Loibl and J Naab. Namecoin. Available at: `https://namecoin.org`, 2011. URL: `https://namecoin.org`.

[106] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 17–30, 2016. URL: `http://doi.acm.org/10.1145/2976749.2978389`, `doi:10.1145/2976749.2978389`.

[107] Billy Markus and Jackson Palmer. Dogecoin. Available at: `https://dogecoin.com`, 2013. URL: `https://dogecoin.com`.

[108] Roman Matzutt, Benedikt Kalde, Jan Pennekamp, Arthur Drichel, Martin Henze, and Klaus Wehrle. How to securely prune bitcoin's blockchain. In *2020 IFIP Networking Conference (Networking)*, pages 298–306. IEEE, 2020.

[109] Colin McDiarmid. *Probabilistic Methods for Algorithmic Discrete Mathematics*, chapter Concentration, pages 195–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. URL: http://dx.doi.org/10.1007/978-3-662-12788-9_6, doi:10.1007/978-3-662-12788-9_6.

[110] Izaak Meckler and Evan Shapiro. Coda: Decentralized cryptocurrency at scale. 2018.

[111] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140, 2013.

[112] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378. Springer, 1987.

[113] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016. URL: http://arxiv.org/abs/1607.01341.

[114] 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. IEEE, 2013.

[115] Andrew Miller. The high-value-hash highway. bitcoin forum post, 2012.

[116] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Available at: https://bitcoin.org/bitcoin.pdf, 2008. URL: https://bitcoin.org/bitcoin.pdf.

[117] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *Annual International Cryptology Conference*, pages 111–126. Springer, 2002.

[118] Tier Nolan. Alt chains and atomic transfers. bitcointalk.org, May 2013.

[119] National Institute of Standards and US Department of Commerce Technology. Processing standards publication fips 180-2, secure hash standard. Aug 2002.

[120] Michael Okun. *Distributed computing among unacquainted processors in the presence of Byzantine failures*. PhD thesis, 2005.

[121] P4Titan. Slimcoin a peer-to-peer crypto-currency with proof-of-burn. Available at: https://www.doc.ic.ac.uk/~ids/realdotdot/crypto_papers_etc_worth_reading/proof_of_burn/slimcoin_whitepaper.pdf, May 2014. URL: https://www.doc.ic.ac.uk/~ids/realdotdot/crypto_papers_etc_worth_reading/proof_of_burn/slimcoin_whitepaper.pdf.

[122] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 643–673, 2017. doi:10.1007/978-3-319-56614-6_22.

[123] Sam Patterson. Proof-of-burn and reputation pledges. Available at: https://www.openbazaar.org/blog/proof-of-burn-and-reputation-pledges/, Aug 2014. URL: https://www.openbazaar.org/blog/proof-of-burn-and-reputation-pledges/.

[124] Andrew Poelstra. Mimblewimble. 2016.

[125] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, 2017.

[126] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[127] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[128] Minghua Qu. Sec 2: Recommended elliptic curve domain parameters. *Certicom Res., Mississauga, ON, Canada, Tech. Rep. SEC2-Ver-0.6*, 1999.

[129] Thomas Ristenpart and Scott Yilek. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. Cryptology ePrint Archive, Report 2007/264, 2007. https://eprint.iacr.org/2007/264.

[130] Phillip Rogaway. The moral character of cryptographic work. *IACR Cryptology ePrint Archive*, 2015:1162, 2015.

[131] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International workshop on fast software encryption*, pages 371–388. Springer, 2004.

[132] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[133] Meni Rosenfeld. Analysis of hashrate-based double spending. *arXiv preprint arXiv:1402.2009*, 2014.

[134] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*, pages 239–252. Springer, 1989.

[135] Victor Shoup. *A Computational Introduction to Number Theory and Algebra.* Cambridge University Press, 2008. URL: https://www.shoup.net/ntb/ntb-v2.pdf.

[136] Georg Simmel. *The Philosophy of Money.* 1900.

[137] Michael Sipser et al. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[138] Inc Smart Contract Solutions. Openzeppelin crowdsale contract. Available at: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/v2.0.0-rc.1/contracts/token/ERC20/ERC20.sol, 2017. URL: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/v2.0.0-rc.1/contracts/token/ERC20/ERC20.sol.

[139] Iain Stewart. Proof of burn - bitcoin wiki. Available at: https://en.bitcoin.it/wiki/Proof_of_burn, Dec 2012.

[140] suppp. Interesting bitcoin address, bitcoin forum post. Available at: https://bitcointalk.org/index.php?topic=237143.0, 2013. URL: https://bitcointalk.org/index.php?topic=237143.0.

[141] Paul Sztorc. Drivechain - the simple two way peg, November 2015. http://www.truthcoin.info/blog/drivechain/.

[142] Jason Teutsch and Christian Reitwießner. A scalable verification solution for blockchains. *arXiv preprint arXiv:1908.04756*, 2019.

[143] Jason Teutsch, Michael Straka, and Dan Boneh. Retrofitting a two-way peg between blockchains. *arXiv preprint arXiv:1908.03999*, 2019. URL: https://people.cs.uchicago.edu/~teutsch/papers/dogethereum.pdf.

[144] Stefan Thomas and Evan Schwartz. A protocol for interledger payments. https://interledger.org/interledger.pdf.

[145] Peter Todd. OpenTimestamps: Scalable, Trustless, Distributed Timestamping with Bitcoin (2016). Available at: https://petertodd.org/2016/opentimestamps-announcement, 2016. URL: https://petertodd.org/2016/opentimestamps-announcement.

[146] Peter Todd. Merkle mountain ranges, October 2012. https://github.com/opentimestamps/opentimestamps-server/blob/master/doc/merkle-mountain-range.md.

[147] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[148] Jan Van Leeuwen and Jiří Wiedermann. The turing machine paradigm in contemporary computing. In *Mathematics unlimited—2001 and beyond*, pages 1139–1155. Springer, 2001.

[149] Nicolas Van Saberhagen. Cryptonote v2.0. Available at: https://cryptonote.org/whitepaper.pdf, 2013. URL: https://cryptonote.org/whitepaper.pdf.

[150] Fabian Vogelsteller and Vitalik Buterin. Erc-20 token standard. sept. 2017. *URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-tokenstandard.md*, 2015.

[151] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[152] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework, 2016.

[153] Karl Wüst and Arthur Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.

[154] Joachim Zahnentferner. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. *IACR Cryptology ePrint Archive*, 2018:262, 2018.

[155] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: A fast blockchain protocol via full sharding. Cryptology ePrint Archive, Report 2018/460, 2018. `https://eprint.iacr.org/2018/460`.

[156] Alexei Zamyatin, Mustafa Al-Bassam, Dionysis Zindros, Eleftherios Kokoris-Kogias, Pedro Moreno-Sanchez, Aggelos Kiayias, and William J Knottenbelt. SoK: Communication across distributed ledgers, 2019.

[157] Alexei Zamyatin, Dominik Harz, Joshua Lind, Panayiotis Panayiotou, Arthur Gervais, and William Knottenbelt. XCLAIM: Trustless, interoperable, cryptocurrency-backed assets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 193–210. IEEE, 2019.

[158] Alexei Zamyatin, Nicholas Stifter, Aljosha Judmayer, Philipp Schindler, Edgar Weippl, William Knottenbelt, and Alexei Zamyatin. A wild velvet fork appears! inclusive blockchain protocol changes in practice. In *International Conference on Financial Cryptography and Data Security*. Springer, 2018.

[159] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 270–282, 2016.

[160] Dionysis Zindros. Trust in Decentralized Anonymous Marketplaces. Master's thesis, National Technical University of Athens, Athens, Greece, 2016.

Index

List of Symbols

$\cap$	intersection of chains 49
$\cup$	union of chains 49
$\{x \in \mathsf{C} : p(x)\}$	filter chain C 49
$\in$	element belongs to set or sequence 49
$\|$	concatenate strings 48
$\perp$	image of transaction with no effect 182
$\preceq$	sequence prefix 49
$\leftarrow$	sample from a distribution 47
$\leftarrow \$$	sample uniformly 47
$\mathfrak{A}$	concrete account-based asset 185
$\mathcal{A}$	the adversary 48
α	chain velocity 87
A	abstract asset 182
$\mathbb{A}$	set of security assumptions for ledger 181
B	block 85
$\mathsf{C}{\downarrow}$	0-level chain used to generate filtered chain C 96
$\mathsf{C}[i]$	i^{th} element of sequence C 49
$\mathsf{C}[-i]$	i^{th} element of sequence C from the end 49
$\|\mathsf{C}\|$	length of the sequence C 49
$\mathsf{C}[i{:}j]$	subsequence of C from indices i to j 49
$\mathsf{C}[{:}j]$	subsequence of C from start to index j 49
$\mathsf{C}\{A{:}Z\}$	subsequence of C from elements A to Z 49
$\mathsf{C}\{{:}Z\}$	subsequence of C from start to element Z 49
$\mathsf{C}\{A{:}\}$	subsequence of C from element A to end 49
$\mathsf{C}[i{:}]$	subsequence of C from index i to end 49
$\mathsf{C}{\uparrow}^{\mu}$	subchain of C at the μ^{th} level 96
C	blockchain 85
Δ	network delay 68
δ	honest advantage 80
ϵ	empty sequence or string 48
ϵ	Chernoff error 83
f	probability of successful round 83
g	generator of group 242
γ	bounded population evolution parameter 69
$\mathcal{G}$	the genesis block 85
$\mathbb{G}$	multiplicative group 242
H	a cryptographically secure hash function 50

k	common prefix parameter 88
κ	security parameter of the Random Oracle 64
$\mathbf{L}_i^{\cap}$	intersection of all ledger views 86
$\mathbf{L}_i^{\cup}$	union of all ledger views 86
L	ledger state 86
$\check{\mathbf{L}}P[t]$	unstable ledger view by party P at time t 86
$\mathbf{L}^P[t]$	stable ledger view by party P at time t 86
$[\mathcal{M}]$	support of distribution $\mathcal{M}$ 48
n	number of parties 65
$[n]$	natural numbers up to n 48
negl	a negligible function 48
NP	non-deterministic polynomial-time problems 243
p	single query probability of success 82
Π	honest protocol 66
P	class of problems solvable in polynomial time 48
PPT	probabilistic polynomial-time Turing machine 48
q	queries per party per round 81
ρ	chain quality 87
r	round 66
$S_{\pm\Delta}$	delta-expansion of a round sequence 166
T	Proof-of-Work target 81
t	number of adversarial parties 65
$\mathcal{U}$	uniform distribution 47
$\mathbb{V}$	validity language 72
$\mathbb{V}_{\mathfrak{A}}$	concrete account-based asset validity language 185
$\mathrm{view}_{\Pi,\mathcal{A}}^{n,t}$	transcript of execution 66
X	indicator variable for successful round 82
X'	Δ-isolated successful round indicator 84
Y	indicator variable for uniquely successful round 82
Y'	Δ-isolated uniquely successful round indicator 84
Z	number of adversarially successful queries 82
$\mathcal{Z}$	environment 65

License

Creative Commons Legal Code

Attribution 3.0 Unported

> Creative Commons corporation is not a law firm and does not provide legal services. Distribution of this license does not create an attorney-client relationship. Creative commons provides this information on an "as-is" basis. Creative commons makes no warranties regarding the information provided, and disclaims liability for damages resulting from its use.

License

The work (as defined below) is provided under the terms of this creative commons public license ("ccpl" or "license"). The work is protected by copyright and/or other applicable law. Any use of the work other than as authorized under this license or copyright law is prohibited.

By exercising any rights to the work provided here, you accept and agree to be bound by the terms of this license. To the extent this license may be considered to be a contract, the licensor grants you the rights contained here in consideration of your acceptance of such terms and conditions.

1. Definitions

(a) **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

(b) **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which,

to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

(i) **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

(a) to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

(b) to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

(c) to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

(d) to Distribute and Publicly Perform Adaptations.

(e) For the avoidance of doubt:

  (1) **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

  (2) **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

  (3) **Voluntary License Schemes**. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.