ABSTRACT

Title of dissertation:	PROVABLE SECURITY FOR CRYPTOCURRENCIES
	Andrew Miller, Doctor of Philosophy, 2016
Dissertation directed by:	Professor Jonathan Katz and Professor Elaine Shi Department of Computer Science

The past several years have seen the surprising and rapid rise of Bitcoin and other "cryptocurrencies." These are decentralized peer-to-peer networks that allow users to transmit money, to compose financial instruments, and to enforce contracts between mutually distrusting peers, and that show great promise as a foundation for financial infrastructure that is more robust, efficient and equitable than ours today.

However, it is difficult to reason about the security of cryptocurrencies. Bitcoin is a complex system, comprising many intricate and subtly-interacting protocol layers. At each layer it features design innovations that (prior to our work) have not undergone any rigorous analysis. Compounding the challenge, Bitcoin is but one of hundreds of competing cryptocurrencies in an ecosystem that is constantly evolving.

The goal of this thesis is to formally reason about the security of cryptocurrencies, reining in their complexity, and providing well-defined and justified statements of their guarantees. We provide a formal specification and construction for each layer of an abstract cryptocurrency protocol, and prove that our constructions satisfy their specifications.

The contributions of this thesis are centered around two new abstractions: "scratch-off puzzles," and the "blockchain functionality" model. Scratch-off puzzles are a generalization of the Bitcoin "mining" algorithm, its most iconic and novel design feature. We show how to provide secure upgrades to a cryptocurrency by instantiating the protocol with alternative puzzle schemes. We

construct secure puzzles that address important and well-known challenges facing Bitcoin today, including wasted energy and dangerous coalitions.

The blockchain functionality, $\mathcal{F}_{\text{BLOCKCHAIN}}$, is a general-purpose model of a cryptocurrency rooted in the "Universal Composability" cryptography theory. We use this model to express a wide range of applications, including transparent "smart contracts" (like those featured in Bitcoin and Ethereum), and also privacy-preserving applications like sealed-bid auctions. We also construct a new protocol compiler, called Hawk, which translates user-provided specifications into privacy-preserving protocols based on zero-knowledge proofs.

PROVABLE SECURITY FOR CRYPTOCURRENCIES

by

Andrew Miller

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2016

Advisory Committee:
Professor Jonathan Katz, Chair
Professor Elaine Shi, Co-Chair
Professor Michael Hicks
Professor Bobby Bhattacharjee
Professor Lawrence C. Washington

Acknowledgements

I thank my advisors at Maryland, Jon Katz, Elaine Shi, Mike Hicks, and Bobby Bhattacharjee, each of whom provided me with invaluable opportunities and advice throughout my research apprenticeship. Much of this work was done collaboration with colleagues at Maryland, especially Ahmed Kosba. I also thank Joe LaViola and Mubarak Shah at Central Florida, who started me on my journey, and Brandyn, who cleared the path for me from FL to MD. Few students get to live out their topic as I have, embedded in such a vibrant and engaging community. I am grateful to everyone with whom I've interacted in Bitcoin, Ethereum, Zcash, and the broad cryptocurrency ecosystem, for making me feel a part of something great. I am especially grateful to Greg Maxwell and Zooko for first teaching me applied cryptography, and to everyone who has worked with me as a mentor or colleague. Finally, I thank my fiancée, Jamila, and family and friends for their love and patience over many years.
The work in this thesis was funded in part by NSF award #1518765.

Contents

Acknowledgements ii

1 Introduction 1

1.1 Overview and Contributions . . . 2

1 Background and Preliminaries 5

2.1 Background on Bitcoin and Cryptocurrencies . . . 5

2.2 Cryptographic Preliminaries . . . 8

2 Modeling Bitcoin as a Consensus Protocol 17

3.1 Related Work . . . 18

3.2 Scratch-off Puzzles . . . 19

3.3 The Message Diffusion Model and Monte Carlo Consensus . . . 24

3.4 The Nakamoto Consensus Protocol . . . 25

3 Nonoutsourceable Scratch-Off Puzzles to Deter Mining Coalitions 35

4.1 Overview . . . 35

4.2 Weakly Nonoutsourceable Puzzles . . . 37

4.3 Strongly Nonoutsourceable Puzzles . . . 44

4 Permacoin: Storage-based Scratch-off Puzzles to Recycle Mining Effort 49

5.1 Overview . . . 49

5.2 The Permacoin Scratch-Off Puzzle . . . 51

5.3 To Outsource or Not to Outsource . . . 54

5.4 Partially Outsourced Storage Analysis . . . 60

5 The Blockchain Model of Computation 63

6.1 Overview . . . 63

6.2 Related Work . . . 64

6.3 Programs, Functionalities, and Wrappers . . . 65

6.4 Implementing Transparent Blockchain Contracts on top of Nakamoto Consensus 69

6 Privacy Preserving Smart Contracts 73

7.1 Overview . . . 73

7.2 Warmup: Private Cash and Money Transfers . . . 76

7.3 Combining Privacy with Programmable Logic . . . 85

7.4 Private Smart Contract Specification . . . 85

7.5 Hawk Implementation and Examples . . . 93

7 Conclusion 105

Bibliography 107

Chapter 1

Introduction

The past several years have seen the unprecedented rise of "cryptocurrencies," virtual currencies that are not administered by any state or corporate entity, but rather exist solely within a decentralized peer-to-peer computer network that anyone can join. Bitcoin, the first (and currently largest) successful cryptocurrency, has operated with essentially uninterrupted service and significant growth since its launch in 2009. At the time of writing, its total market capitalization (the current market price per bitcoin, multiplied by the total number of bitcoin units in circulation) exceeds $10 billion US dollars. This quantity is comparable to the market capitalization of publicly-traded companies in the Fortune 500, and approximately equal to the M1 money supply[1] of the Quetzal (the national currency of Guatemala).

Cryptocurrencies are often referred to as a "glimpse into the future of finance" [1]. By eliminating central points of failure and dispersing influence over a wide network, they carry the potential promise of a more robust and equitable financial infrastructure. Existing outside of traditional regulated structures, they have also helped accelerate deployment of innovations like "smart contracts," which allow users to control their money by writing fragments of program code. The total value of venture capital investment in Bitcoin-related technology companies today exceeds $1 billion US dollars [2].

With such significant value at stake, security is clearly of paramount concern. Since they allow open participation from anonymous users, cryptocurrencies are unamenable to traditional means of policy enforcement, but instead derive their security from the strength of underlying network protocol itself. However, although Bitcoin has been an empirical success thus far, it is difficult to reason about its security. Cryptocurrencies are complex systems, comprising many intricate and subtly-interacting protocol layers. Furthermore, Bitcoin's success has led to a Cambrian explosion of "altcoins," hundreds of protocols that modify its template in various ways and compete with it for market share. Compounding the challenge, the cryptocurrency ecosystem also undergoes constant evolution as open source development communities improve their performance and extend their functionality.

[1] The M1 money supply is the amount of currency in circulation plus the sum of short-term deposit accounts (e.g., checking accounts and saving accounts).

The goal of this thesis is to formally reason about the security of cryptocurrencies, reining in their complexity, and providing well-defined and justified statements of their guarantees. *We show that it is possible to construct a provably secure cryptocurrency protocol that captures the underlying techniques and capabilities of the real systems in use today.*

The results of our effort have three main benefits. First, we provide positive evidence for the security of the Bitcoin protocol. By expressing it as a positive result in distributed computing, we provide a partial explanation for Bitcoin's phenomenal success thus far — it embodies a novel design that relies on weaker assumptions than those prior known.

Of course, a provably secure protocol model does not guarantee that real-life cryptocurrency systems will succeed. Several well-known drawbacks of Bitcoin and Ethereum threaten to undermine the assumptions that our protocol relies on, such as their high cost in computing power, and the fact that miners in reality are prone to collusion. Fortunately, our modular framework enables us to develop protocol upgrades that address these concerns, while still preserving the original security guarantees.

Finally, we show how to construct provably-secure applications atop a cryptocurrency. Not only can we model existing designs like Ethereum smart contracts and Zerocash's privacy-preserving payments, we extend the state-of-the-art by developing Hawk, a new protocol for privacy-preserving auctions, crowdfunding campaigns, and more.

1.1 Overview and Contributions

This thesis is structured as a layered construction of a provably secure cryptocurrency system. The layers range from the low-level (a model of an anonymous message propagation network in Section 3.3) to high-level (the user-provided applications that ensure privacy in Section 7.3). At each layer, we provide formal specifications of our security goals and primitive assumptions, and prove that our protocols satisfy these.

In summary, this thesis makes the following contributions:

- In Chapter 2, we provide the first formal analysis of Bitcoin's underlying consensus protocol. More specifically, we study a simplified variation, which we call Nakamoto consensus. To study this consensus protocol, we make develop an idealized model of Bitcoin's underlying peer-to-peer communication network, which roughly acts like an anonymous broadcast channel. Surprisingly, this communication model is significantly weaker than the standard setting in distributed systems, which provides some form of identity-based communications (e.g., a PKI that associates identities to public keys). Instead, it relies on an alternative assumption about the allocation of computational resources (i.e., a majority of the network's aggregate computational power must follow the protocol correctly) [3].

  We also provide a generalized abstraction of the core mechanism underlying Nakamoto consensus, which we call a Scratch-Off Puzzle (SOP). This abstraction expresses the characteristics needed to fill their role in the Nakamoto consensus protocol: in particular, they

must have no shortcut, and they must be efficiently solved by many individuals working independently.

- By instantiating Nakamoto consensus protocol with different scratch-off puzzles, we obtain provably secure consensus protocols with additional beneficial properties. In Chapter 3, we construct a novel SOP that discourages large Bitcoin mining coalitions, one of Bitcoin's widely-feared threats. Our construction is derived from hash-based signatures and zero-knowledge proofs. We analyze this construction in a formal framework that captures a wide range of coalitions, including both "mining pools" and "hosted mining," the two forms that occur in practice today [4].

- In Chapter 4, we construct a novel storage-based SOP and develop a system based on it, Permacoin, that recycles cryptocurrency mining resources for a secondary, socially useful purpose [5]: encouraging the distributed storage of an public archival dataset.

- In Chapter 5, we define a formal framework for constructing applications built atop the Nakamoto consensus protocol. Our framework is general enough to express applications based on "transparent smart contracts," such as those of Bitcoin and Ethereum, as well as privacy-preserving applications. Our framework is rooted in the Universal Composability (UC) theory of cryptography [6].

- In Chapter 6, we show how to use this framework to develop new, provably-secure applications for cryptocurrencies. We first use our framework to formalize (a simplification of) Zcash, an existing protocol for privacy-preserving payments [7]. Next, we use our framework to construct a novel protocol, Hawk, which implements a broad class of privacy-preserving mechanisms, including auctions, crowdfunding campaigns. The Hawk protocol is based on zero-knowledge proofs, hiding all the private inputs of the participants while ensuring that the application-specific functionality is computed correctly. We develop an application compiler for Hawk, which generates an executable for the protocol according to a user-provided specification [8].

Chapter 1

Background and Preliminaries

This chapter presents background information on cryptocurrency systems and defines notation and cryptographic primitives used in later chapters.

2.1 Background on Bitcoin and Cryptocurrencies

The idea of cryptographic currency dates back at least to Chaum's proposal for "untraceable electronic cash" in 1983 [9], a system involving bank-issued cash in the form of blindly signed coins. Unblinded coins are transferred between users and merchants, and redeemable after the bank verifies they have not been previously redeemed. Several startup companies in the 1990s including DigiCash [10] and Peppercoin [11] attempted to bring electronic cash protocols into practice but ultimately failed in the market. No schemes from this "first wave" of cryptocurrency research achieved significant deployment. Compared to these prior efforts, Bitcoin's most salient distinction is that it has no central authority, owner, or administrator; rather, the bank is implemented as a decentralized peer-to-peer computation in which anyone can participate.

In our development we provide self-contained definitions of a simplified protocols, abstracting away the less-relevant details about the Bitcoin protocol itself. However, below we give a brief technical introduction to the function of Bitcoin and the cryptocurrency ecosystem. For a more thorough explanation of the Bitcoin protocol, we refer the reader to surveys [12, 13].

Puzzles, rewards, and epochs. In Bitcoin, new money is printed at a predictable rate through a distributed coin-minting process. At the time of writing, roughly speaking, 25 bitcoins are minted every 10 minutes (referred to as an epoch) on average. When an epoch begins, a public puzzle instance is generated by computing an up-to-date hash of the global transaction log (called the "blockchain"). Then, Bitcoin nodes race to solve this epoch's puzzle. Whoever first finds an eligible solution to the puzzle can claim the newly minted coins corresponding to this epoch.

In slightly more detail, miners start with the puzzle instance puz, and construct a payload m which contains (a tree hash over) the miners' public keys and a new set of transactions to commit to the log during this epoch. They then search for a nonce r such that $\mathcal{H}(\mathsf{puz}\|m\|r) < 2^{-d}$, where $\mathcal{H} : \{0,1\}^* \to [0,1]$ is a hash function and d is a difficulty parameter, and puz is derived from the puzzle solution of the previous epoch. The difficulty parameter is adjusted according to the total amount of computational resources devoted to mining to ensure that each epoch lasts 10 minutes on average.

Bitcoin nodes reach consensus on the history of transactions by following a simple rule: they adopt the longest chain of puzzle solutions as the authoritative one, and attempt to extend this chain with further puzzle solutions. Roughly speaking, this defeats history revision attacks, since to revise history would involve computing a blockchain that is more difficult than the known good chain. An adversary must therefore possess a significant fraction of the total computational resources to successfully race against the rest of the network in extending the chain.

Bitcoin's Peer-to-Peer Network. Bitcoin's consensus mechanism relies on being able to broadcast messages throughout the entire network. Each time a puzzle solution is found by a miner, or a transaction is created by a user, this information must be propagated throughout the network as quickly as possible. These communications are carried out over a peer-to-peer overlay network, the topology of which is formed through a randomized process. Every node maintains a local list of potential peer addresses to connect to. New nodes initialize this list by querying one of a handful of "seed nodes" listed in the source code; thereafter, nodes propagate updated information about peers through gossip. By default, each node attempts to maintain outgoing connections to 8 peers, choosing randomly from its local list. Each node also accepts up to 125 incoming connections from arbitrary other peers.

In reality, the Bitcoin peer-to-peer network layer is imperfect, and susceptible to various denial of service and deanonymization attacks [14–19].

In this thesis, we make use of a idealized abstraction of this communication functionality (see Section 3.3). Our framework guarantees bounded-delay propagation of every message sent from an honest party, and furthermore conceals the origin of each message (i.e., provides anonymity to the sender). We note that subsequent modeling efforts [20–22] also make use of this abstraction.

Mining Pools. In the simplest case, each individual Bitcoin mining participant behaves independently, communicating only through messages defined by the protocol.

However, throughout the past several years, the vast majority of Bitcoin participants join coalitions called "mining pools" rather than participating as independents (i.e., "solo-mining"), primarily in order to reduce uncertainty in their payoff [23].

Most mining pools (with the exception of P2Pool [24]) are administered by a trusted "pool operator" who directs how the hashpower of the pool is used. The original Bitcoin whitepaper [25] draws an analogy between Bitcoin mining and voting in a democratic election (i.e. "one-cpu-one-vote"). Extending the analogy, these pools are akin to vote buying, since they offer an economic incentive in exchange for their influence.

At several times over the past year, the largest handful of mining pools have accounted for well over a third of the network's overall computing effort [26]. Recently the largest mining pool, GHash.IO, has even exceeded 50% of the total mining capacity. The Bitcoin community has therefore expressed a significant demand for technical solutions for this problem [27].

A key enabling factor in the growth of mining pools is a simple yet effective *enforcement mechanism*; members of a mining pool do not inherently trust one another, but instead submit cryptographic proofs (called "shares") to the other pool members (or to the pool operator), in order to demonstrate they are contributing work that can only benefit the pool (e.g., work that is tied to the pool administrator's public key). We address the problem or large pools by constructing a new protocol that thwarts such enforcement mechanisms.

Programmable Money and Smart Contracts. Although Bitcoin is primarily intended to provide an e-cash service, allowing users to transfer quantities of virtual money from one to another, it is clear that the underlying network can also be used for implementing more complicated financial applications. Bitcoin features a simple programming language for defining access control policies. For example, it is possible to define a policy script such that a quantity of money can only be transferred with the consent of (i.e., with a signed message from) 2 out of 3 principals. Furthermore, Ethereum [28], currently Bitcoin's largest competitor, features a very flexible programming model for controlling the flow of money: users provide program that are run as processes called "smart contracts," which are executed by the network and can send and receive money and data inputs from users accounts. Ethereum's tutorial documentation walks the programmer through creating mechanisms like auctions, lotteries, and elections, in a javascript-like programming language called Solidity and a python-like language called Serpent.

Though this smart contract model is powerful, programming even a simple smart contract remains an error prone exercise [29], and (prior to our work) there has not yet been a framework for proving the security of smart contract-based applications. An additional challenge is that since smart contracts are executed by the public peer-to-peer network, they are inherently "transparent," i.e., they cannot have any private state. The smart contract model can thus be thought of as a *third party that is trusted for correctness and availability, but not for privacy.*

Applications that require privacy, which include many natural settings in finance (since financial information is often sensitive for individuals and businesses) cannot be implemented with a smart contract alone, but would instead require an additional layer of cryptography, adding to the complexity. Our framework also supports provably secure, *private* contracts, which serve as specifications for privacy-preserving applications.

Cryptocurrencies and Cryptography. Our work is closely related to two main directions that the cryptography research community has pursued in response to cryptocurrencies.

The first direction is to use more sophisticated cryptography to improve the privacy of Bitcoin payments. Although Bitcoin is designed to allow pseudonymous use, it uses only simplistic cryptography (just digital signatures) and leaks considerable information. Several studies have found that it is possible to forensically analyze the Bitcoin blockchain, (especially when combined

with side-channels in the p2p network as mentioned earlier) and to trace the flow of funds by inferring connections between related transactions [30–33]. Fortunately, there have been several applicable techniques proposed in the e-cash literature predating Bitcoin — especially the so-called "auditable" e-cash schemes of Sander, Ta-Shma, and Yung [34, 35]. These auditable e-cash protocols obscure the transaction graph by using zero-knowledge proofs. and rely only on "transparent" central parties without public state (i.e., those that can be implemented using a public consensus protocol). Recent efforts have improved the efficiency of these schemes and even made them into practical implementations [36–39]. We formalize (a simplification of) one of these schemes, Zerocash, in our framework (in Section 7.2).

The second direction has been to use a cryptocurrency as a primitive within a cryptographic protocol, and in particular as a new way of providing "fairness" in multi-party computations. Roughly speaking, fairness means that either all parties receive the desired output of a distributed computation, or else none of them do. It is well-known that fairness is generally impossible in the plain model without any trusted intermediary [40]. Using cryptocurrencies as a primitive, however, we can implement a compelling form of "fairness with penalties," where either fairness holds, or else a penalty of digital currency is seized from the attacker and disbursed to compensate the honest parties [22, 41, 42].

Our work can be seen as subsuming both of these goals. The model of Kumaresan and Bentov [22] and Kiayias et al. [42] support both *public payments*, and *private computations*, but the private output of a computation can only be data, not a payment effect. Our framework supports both public and privacy-preserving payments (as described in Section 7.2), and allows the output of a private computation to include both private data and effects. The protocol we develop in Section 7.3 is a non-trivial extension of Zerocash [7], though it relies on the same suite of cryptographic tools (generic zero-knowledge proofs).

2.2 Cryptographic Preliminaries

2.2.1 Notation

Throughout our development, we assume that all protocols are implicitly paramterized by a security parameter, 1^λ. The term "negligible" is defined as $\mathsf{negl}(\lambda) < 1/\mathsf{poly}(\lambda)$. Similarly by "high probability" we mean $1 - \mathsf{negl}(\lambda)$.

We write $[\ell]$ to denote the set $\{i \in \mathbb{N} | 1 < i < \ell\}$

2.2.2 The UC Protocol Definition Framework

Our security notions are defined in a variation of the universal composability (UC) framework, introduced by Canetti [6]. We now give a brief and high-level description of this framework, highlighting the differences in our variation.

Universal composability is a general-purpose definitional framework, an alternative to. UC is defined by an experiment involving the concurrent execution of several kinds of processes, including protocol parties (Π), adversaries ($\mathcal{A}$), and an environment ($\mathcal{E}$). All of these processes are defined as interactive Turing machines. The distinguishing feature of UC is that both a) the assumptions of a protocol, such as the communication model, cryptographic setup assumptions, and b) the application or problem specifications, are all defined in terms of trusted services called "functionalities," modeled as an additional process in the system.

We benefit from the use of this framework in several ways. First, the UC framework is a versatile alternative to game-based or property-based definitions in cryptography and distributed systems. Instead of specifying an application by laying out several desired properties or classes of attacks to prevent, in UC an application is specified by providing program code for an "ideal functionality," a hypothetical third party that, if trusted, would fulfill all the desired guarantees. This approach helps us manage when specifying complex applications. This is especially important for our culminating application, Hawk (see Chapter 6), which requires many phases of input, involves many different roles for different parties, and offers fine-grained guarantees under multiple failure modes. Secondly, UC most naturally expresses a very strong form of secrecy/privacy guarantees, which our constructions satisfy. As we mention in Section 7.2.4, alternative definitions for similar applications have subtle gaps that allow for unintended leakage. Finally, since a functionality definition can be used either as a construction goal or as a starting assumption, the UC framework provides a natural approach for modular or incremental constructions. For example, in Chapter 5 we define a protocol that implements the $\mathcal{F}_{\text{BLOCKCHAIN}}$ functionality, and in Chapter 6 we make use of the $\mathcal{F}_{\text{BLOCKCHAIN}}$ functionality as a trusted service.

2.2.2.1 Interactive Turing Machine Systems

As mentioned above, all of the processes in the UC experiment are defined in terms of Interactive Turing Machine (ITM) systems. An ITM is a Turing machine with a collection of input and output tapes. A system consists of several of these machines, with their tapes connected up to each other (input tape to output tape). When two machines are connected by a tape, if one of them writes a full-formed message to the "output end" of the tape, then the machine on "input end" is "activated" and can read and process this message. The program code of an ITM thus defines how to respond to each incoming message, ultimately either terminating or writing to an output tape.

An essential composition technique is to define an ITM in terms of "sandbox execution" of other ITMs. We often describe this as an ITM that runs another subsystem of machines "locally," potentially defining a transformation of inputs and outputs between them.

2.2.2.2 The UC Execution Model

The UC framework is centered around an experiment involving the execution of a distributed system of processes. This execution is parameterized by several ITM programs: a protocol Π,

an adversary $\mathcal{A}$, an environment $\mathcal{E}$, and a functionality $\mathcal{F}$. We describe the roles of each of these processes shortly. The experiment is denoted $\text{EXEC}(\Pi, \mathcal{F}, \mathcal{A}, \mathcal{E})$.

We follow a convention of describing ITMs as reactive machines, by defining how to respond to each received message. By convention, we assume that messages are prepended with tags, as in `tag`(...). We often use the shorthand "Assert X", where X is some predicate; this means to ignore the previously received message if X does not hold. We now describe conventions for defining each of these kinds of ITMs.

Protocols. The protocol Π is jointly run by an arbitrary number of parties. Parties are associated with arbitrary strings, called party identifiers $\mathcal{P}_i$. Parties are able to condition on their own identifier strings, $\mathcal{P}_i$. Parties do not run until they are activated, thus while there may be an unbounded number of potential parties, a typical execution involves only a bounded number.

Functionalities. Protocol parties do not interact directly with each other, but instead communicate through an intermediary called a "functionality." Functionalities are used to model network primitives (e.g., synchronous point-to-point network) and cryptographic setup assumptions (e.g., a common reference string). The more services this functionality provides, the stronger are the assumptions thus modeled.

Functionalities also serve as a straightforward and versatile way of specifying an application. We start by defining a functionality that exhibits the behaviors we would desire for our protocol — for example, a consensus protocol is easily modeled as a functionality that accepts input from every party, chooses an acceptable value, and sends it to all the parties. Next we construct a protocol that is "just as good," but relies only on a simpler primitive (e.g., a lossy broadcast channel). "Just as good" is given a formal meaning, as we explain shortly, by the notion of UC-secure realization. Naturally, we can use functionalities as an interface for composition — a protocol that realizes a functionality can be substituted in place of that functionality.

Parties

Adversaries. Our attack model is the "static Byzantine" attacker. Such an attacker is given control over some number of "corrupted" parties. The adversary choose these parties at the outset of the protocol. We'll often bound the number of such compromises, such that the guarantees only hold against for adversaries that respect this bound. The functionality $\mathcal{F}$ is informed of which parties are corrupted, and can condition its behavior accordingly. For example, our network model includes a weak form of broadcast channel, where an honest sender's message is guaranteed to reach each other party within a bounded amount of time, but a corrupted sender can deliver messages to some party and not others.

Environments. The environment $\mathcal{E}$ and the adversary are also able to communicate over an arbitrary interface. In a sense, the environment is an extension of the adversary. Inputs to the protocol, and outputs from the protocol are modeled by interaction between the protocol parties

and the environment, $\mathcal{E}$. Thus "on input x" in a protocol definition means "on receiving x from $\mathcal{E}$".

Allowing the environment to control the inputs and outputs means that the protocol guarantees must hold even for arbitrary (i.e., potentially adversarially influenced) input choices. The environment can also be thought of as representing the arbitrary other programs that run alongside the protocol on a user's machine. This notion is essential to the compositionality (i.e., to the proof of the "composition theorem" [6]), since the inputs to a protocol may be chosen by an arbitrary other protocol that composes with it in blackbox fashion. Of course, the environment is limited to seeing "side-effects." It cannot see the internal state of the protocols of honest parties or the private states of the functionality, nor can it observe direct communications between parties and the functionality.

Secure emulation. In the UC framework, a pair of a protocol and functionality $(\Pi_1, \mathcal{F}_1)$, called the "real world" or "hybrid world," are said to securely emulate another pair $(\Pi_2, \mathcal{F}_2)$, called the "ideal world," if every attack that exists on the former also exists in the latter. This is formalized by stating that for every real world adversary $\mathcal{A}$, there must exist an ideal world adversary $\mathcal{S}_\mathcal{A}$ (that might depend on $\mathcal{A}$), such that the real and ideal world execution experiments are indistinguishable to any environment:

$$\forall \mathcal{A}, \exists \mathcal{S}_\mathcal{A}, \forall \mathcal{E}, \quad \text{EXEC}(\mathcal{E}, \Pi_1, \mathcal{F}_1, \mathcal{A}) \approx \text{EXEC}(\mathcal{E}, \Pi_2, \mathcal{F}_2, \mathcal{S}_A)$$

Due to Canetti [6], it suffices to construct a simulator $\mathcal{S}$ for the dummy adversary $\mathcal{A}_\emptyset$, which merely forwards information between the functionality and the environment. It's also useful to define a dummy protocol, called the ideal protocol $\Pi_\emptyset$, which simply forwards input messages from the environment directly to the functionality, and relays messages from the functionality directly as output to the environment. We say that a protocol Π realizes $\mathcal{F}_2$ (in the $\mathcal{F}_1$-hybrid world) if $(\Pi, \mathcal{F}_1)$ emulates $(\Pi_\emptyset, \mathcal{F}_2)$.

2.2.2.3 Timing and Message Delivery

We are interested in defining protocols that involve communication over a synchronous (i.e., time-aware) network. To maintain compatibility with UC (since the underlying model of ITM systems is *not* time-aware), we build our mechanisms for modeling time into the functionalities.

We define a mechanism for functionality programs to keep track of time (counted in discrete "rounds"); and `now` always refers to the current round. Functionality programs can use this mechanism to schedule tasks to be executed on (or before) a later round. This is expressed using the short hand "schedule `task` for no later than round r.' All such tasks scheduled for round r must be delivered before the round can advance to $r + 1$. This functionality gives the adversary significant control over the scheduling. The adversary triggers the delivery of tasks by sending `deliver`$(\cdot)$ messages to the functionality, and advances the round with an `advanceRound` message. We'll be interested in adversaries and environments that "make progress," that is they succeed at calling `advanceRound` sufficiently many times. *We stress that the scheduled time*

associated with a task is only a maximum — the adversary can deliver any task at any earlier time.

The functionalities we use guarantee that each party must be activated (i.e., sent a `tick` message) at least once per round. This is accomplished by scheduling the next round's tasks at the beginning of the previous round. Synchronous network communications are also implemented this way, by scheduling a task (for the next Δ^{th} round, i.e., the next communication round) that delivers a message to each party. Our model therefore inherently models "rushing" (i.e., "front-running") attacks, since the adversary is allowed to choose the order in which messages are delivered in a round, and can insert his own messages to deliver immediately.

To maintain secrecy, the adversary is not given a way to inspect the task queue or learn its contents. Instead the adversary indicates which task to deliver by giving an array index into the task queue. To handle this, throughout our development, we discuss functionalities that enable the adversary to build a meaningful "mirror" of the task queue. For example, our broadcast channel functionality leaks the contents of each message to $\mathcal{A}$, which suffices to reconstruct the queue exactly.

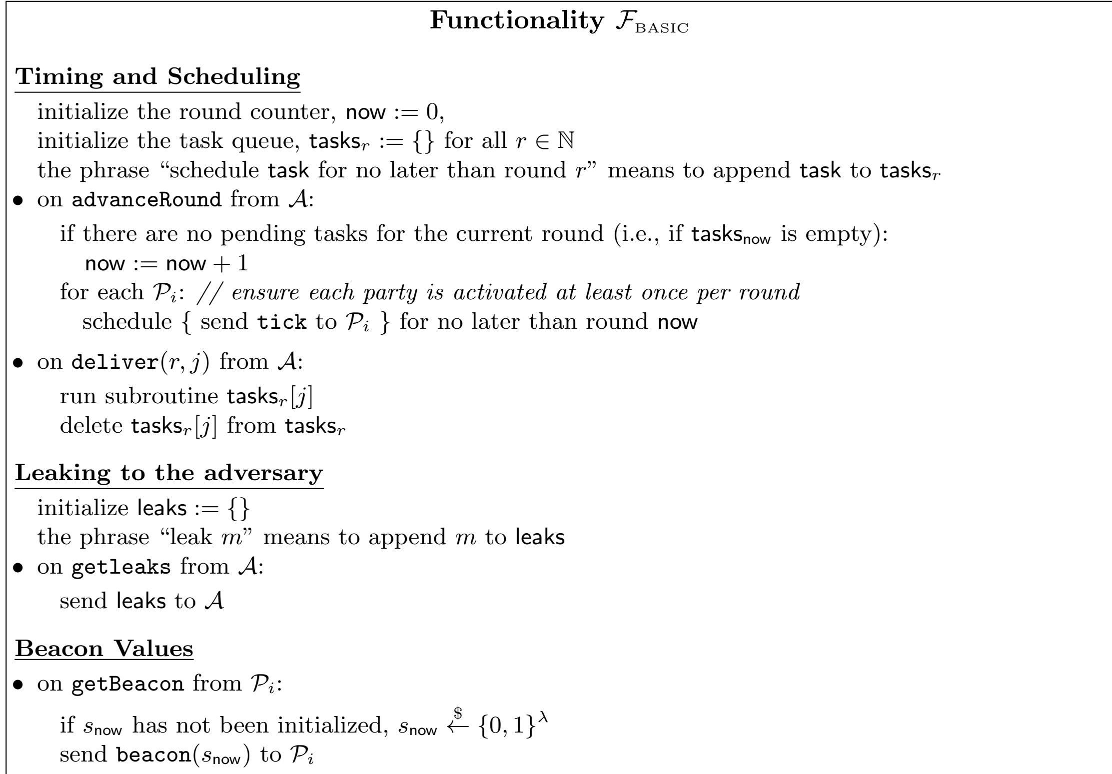
At a high level, our approach is similar to the $\mathcal{F}_{\text{SYN}}$ functionality of Canetti [6] and also to the $\mathcal{F}_{\text{CLOCK}}$ model of Katz et al. [43]. However, our model differs mainly in the following two technical respects:

1. Our functionalities are written in a way that avoids the "reentrancy" problem, a subtle modelling error identified by Camenisch et al. [44] that affects prior models including $\mathcal{F}_{\text{SYN}}$ and $\mathcal{F}_{\text{CLOCK}}$ do. This problem occurs when the functionality directly activates the adversary after receiving a message from a party, expecting a response from the adversary (e.g., "on receiving `send`(m) from $\mathcal{P}$, send m to $\mathcal{A}$ and wait for `OK` from $\mathcal{A}$"). Formally speaking, this leads to undefined behavior, since the adversary could discard such activation or invoke another protocol party before continuing, in which case the `OK` might never arrive. We avoid this problem by defining only functionality always return control immediately to the sender. [1]

2. We provide a convenient shorthand (based on scheduling tasks for later rounds) that enables specification of arbitrary time-aware functionalities with rushing adversaries. This can roughly be thought of as a central time service that functionalities (and protocols) can use.

Modeling constrained computational power. Our development of protocols based on computational puzzles requires us to make precise assumptions about the computational power of the processes, specifically that each process is able to perform a fixed amount of computation in each time instant. To account for this, we require that when a process receives a `tick` activation (i.e., once per round), it is permitted to perform a limited number of computational steps. We do not see how to enforce this constraint by defining it into the functionality, so instead we state this as an assumption about the model.

[1] Camenisch et al. [44] propose an alternate workaround, which is to extend the ITM model to incorporate special "restricting" messages which must be answered by the adversary immediately, without activating other machines.



Functionality $\mathcal{F}_{\text{BASIC}}$

Timing and Scheduling

initialize the round counter, $\mathsf{now} := 0$,
initialize the task queue, $\mathsf{tasks}_r := \{\}$ for all $r \in \mathbb{N}$
the phrase "schedule task for no later than round r" means to append task to tasks_r

- on `advanceRound` from $\mathcal{A}$:
  - if there are no pending tasks for the current round (i.e., if $\mathsf{tasks}_{\mathsf{now}}$ is empty):
    - $\mathsf{now} := \mathsf{now} + 1$
  - for each $\mathcal{P}_i$: *// ensure each party is activated at least once per round*
    - schedule { send `tick` to $\mathcal{P}_i$ } for no later than round now
- on `deliver`(r, j) from $\mathcal{A}$:
  - run subroutine $\mathsf{tasks}_r[j]$
  - delete $\mathsf{tasks}_r[j]$ from tasks_r

Leaking to the adversary

initialize $\mathsf{leaks} := \{\}$
the phrase "leak m" means to append m to leaks

- on `getleaks` from $\mathcal{A}$:
  - send leaks to $\mathcal{A}$

Beacon Values

- on `getBeacon` from $\mathcal{P}_i$:
  - if s_{now} has not been initialized, $s_{\mathsf{now}} \xleftarrow{\$} \{0,1\}^\lambda$
  - send `beacon`(s_{now}) to $\mathcal{P}_i$

FIGURE 2.1: A basic framework for defining synchronous functionalities

Leaking information. We often want to model the "leakage" of information to the adversary. For example, when one party sends a message to another over the diffusion channel, this message is revealed to the adversary. In order to maintain the responsiveness of the functionality (i.e., to guarantee that control returns immediately to the sender), we do not activate the adversary immediately, but instead store the leaked information in a buffer. The adversary can later read this buffer by polling. The phrase "leak m" is used as shorthand for appending to this buffer. This is in contrast to the usual presentation of authenticated channels in UC, for example, which activates the adversary and therefore control potentially never returns to the party.

On the omission of session IDs. Readers familiar with UC may notice that we do not describe any mechanisms for handling session identifiers (SIDs). In the standard UC framework, each protocol and functionality is parameterized by an arbitrary SID. This is especially useful when various instances of protocols and functionalities can run concurrently. Intuitively, the use of distinct SIDs for each protocol instance can help prevent messages from one instance being replayed in another instance. In each of our constructions, we do not make use of concurrent execution (i.e., each protocol uses only a single functionality), and hence elide description of the SID mechanism for simplicity.

2.2.3 Non-Interactive Zero-Knowledge Proofs

A non-interactive zero-knowledge proof system (NIZK) for an NP language $\mathcal{L}$ consists of the following algorithms:

- $\mathcal{K}(1^\lambda, \mathcal{L}) \rightarrow \mathsf{crs}$, also written as $\mathsf{KeyGen}_{\mathrm{nizk}}(1^\lambda, \mathcal{L}) \rightarrow \mathsf{crs}$: Takes in a security parameter λ, a description of the language $\mathcal{L}$, and generates a common reference string crs.
- $\mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w) \rightarrow \pi$: Takes in crs, a statement stmt, a witness w such that $(\mathsf{stmt}, w) \in \mathcal{L}$, and produces a proof π.
- $\mathsf{Verify}(\mathsf{crs}, \mathsf{stmt}, \pi) \rightarrow \{0, 1\}$: Takes in a crs, a statement stmt, and a proof π, and outputs 0 or 1, denoting accept or reject.
- $\widehat{\mathcal{K}}(1^\lambda, \mathcal{L}) \rightarrow (\widehat{\mathsf{crs}}, \tau, \mathsf{ek})$: Generates a simulated common reference string $\widehat{\mathsf{crs}}$, trapdoor τ, and extract key ek
- $\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt}) \rightarrow \pi$: Uses trapdoor τ to produce a proof π without needing a witness

Perfect completeness. A NIZK system is said to be perfectly complete, if an honest prover with a valid witness can always convince an honest verifier. More formally, for any $(\mathsf{stmt}, w) \in R$, we have that

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}),\ \pi \leftarrow \mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w) : \\ \quad \mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1 \end{array}\right] = 1$$

Computational zero-knowlege. Informally, an NIZK system is computationally zero-knowledge if the proof does not reveal any information about the witness to any polynomial-time adversary. More formally, a NIZK system is said to computationally zero-knowledge, if there exists a polynomial-time simulator $S = (\widehat{\mathcal{K}}, \widehat{\mathcal{P}})$, such that for any non-uniform polynomial-time adversary $\mathcal{A}$,

$$\begin{array}{rl} & \Pr\left[\mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\mathcal{P}(\mathsf{crs}, \cdot, \cdot)}(\mathsf{crs}) = 1\right] \\ \approx & \Pr\left[(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \widehat{\mathcal{K}}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}}, \tau, \cdot, \cdot)}(\widehat{\mathsf{crs}}) = 1\right] \end{array}$$

In the above, $\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt}, w)$ verifies that $(\mathsf{stmt}, w) \in \mathcal{L}$, and if so, outputs $\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt})$ which simulates a proof without knowing a witness. Otherwise, if $(\mathsf{stmt}, w) \notin \mathcal{L}$, the experiment aborts. This notion is *adaptive* zero knowledge in the sense that the simulator must specify the reference string before seeing the theorem statements.

Computational soundness. A NIZK scheme for the language $\mathcal{L}$ is said to be computationally sound, if for all polynomial-time adversaries $\mathcal{A}$,

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}), (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(\mathsf{crs}) : \\ (\mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1) \wedge (\mathsf{stmt} \notin \mathcal{L}) \end{array}\right] \approx 0$$

Simulation extractability. All of our NIZK-based constructions rely on simulation extractability. Simulation extractability is a strong notion which requires that even after seeing many simulated proofs (even for false theorems), whenever the adversary makes a new proof, a simulator is able to extract a witness. Simulation extractability implies simulation soundness and non-malleability (i.e., it is not feasible for an adversary to take a verifying proof and "maul" it into a verifying proof for another statement) since if the simulator can extract a valid witness from an adversary's proof, the statement must belong to the language. More formally, a NIZK system is

said to be simulation extractable if it satisfies computational zero-knowledge and additionally, there exists a polynomial-time algorithm $\mathcal{E}$, such that for any polynomial-time adversary $\mathcal{A}$, it holds that

$$\Pr\left[\begin{array}{l}(\widehat{\mathsf{crs}},\tau,\mathsf{ek})\leftarrow\widehat{\mathcal{K}}(1^\lambda,\mathcal{L});\\(\mathsf{stmt},\pi)\leftarrow\mathcal{A}^{\widehat{\mathcal{P}}(\widehat{\mathsf{crs}},\tau,\cdot)}(\widehat{\mathsf{crs}},\mathsf{ek});\\w\leftarrow\mathcal{E}(\widehat{\mathsf{crs}},\mathsf{ek},\mathsf{stmt},\pi):\mathsf{stmt}\notin Q\text{ and}\\\qquad\qquad(\mathsf{stmt},w)\notin\mathcal{L}\text{ and }V(\widehat{\mathsf{crs}},\mathsf{stmt},\pi)=1\end{array}\right]\approx 0$$

where in the above, Q is the list of oracle queries made by $\mathcal{A}$ to $\widehat{\mathcal{P}}(\widehat{\mathsf{crs}},\tau,\cdot)$. Here the $\widehat{\mathcal{K}}$ is identical to the zero-knowledge simulation setup algorithm.

Note that the extractor algorithm $\mathcal{E}$ works for all adversaries, and does not therefore depend or have access to the adversary's code. Rather, the extractor's advantage arises entirely from its special access to a trapdoor ek for the $\widehat{\mathsf{crs}}$. Next, note that the adversary may be able to fake a (different) proof for a statement that has been queried, however, it is not able to forge a proof for any other invalid statement.

Practical Implementations of NIZKs. There have been several recent advances in efficient and generic NIZK constructions for arbitrary NP languages [45–47]. There are several freely available library implementations of these (including libsnark [47], snarklib, Pinocchio [46], and Geppetto [48]). We make use of libnark in our prototype implementations.

More precisely, these constructions are zkSNARKs (zero-knowledge succinct noninteractive arguments of knowledge), which are NIZKs that are additionally succinct (i.e., the size of the proof is independent of the size of the witness), but are not simulation extractable. A known transformation can upgrade an arbitrary NIZK to simulation extractable [49, 50]. When applied to a zkSNARK, however, this transformation sacrifices the succinctness property (intuitively, the transformation includes an encryption of the witness alongside the zkSNARK proof); regardless, for small witnesses these costs are reasonable.

These constructions are defined for languages represented by arbitrary arithmetic circuits. As a set of constraints of the form $(\vec{A}\cdot\vec{x})(\vec{B}\cdot\vec{x})=\vec{C}\cdot\vec{x}$, where $\vec{x}$ is a vector of all the wire variables in the circuit (each element is in $\mathbb{Z}_p$, for a large (e.g., 254-bit) prime p), and $\vec{A}$, $\vec{B}$, and $\vec{C}$ are vectors of constants (also in $\mathbb{Z}_p$). The practical performance of the proof system depends on the size and structure of this circuit.

Chapter 2

Modeling Bitcoin as a Consensus Protocol

A parable is often told about the Rai-stone money of Yap [51]. As the story goes (coarsely abridged), the Micronesian islands of Yap once used enormous Rai stones as currency, making payments with boats to transport them up or down the river. At one point, a boat capsized, and the Rai stones sunk deep in the riverbed, unrecoverable. Rather than write off the ruined money as a loss, they simply kept accounting for it as though it were there. This story reveals a key principle underlying Bitcoin's design: the physical nature of the currency doesn't really matter — it's sufficient if everyone concerned can keep track of the state of things, agreeing on who owns how much money at any time.

Ownership of Bitcoin is expressed using public key cryptography: the network unanimously agrees on the association between public keys and portions of the money supply. Money can naturally be transferred from one public key to another using digitally signed messages. Bitcoin is thus easily understood as a distributed state machine, where the state at any given time corresponds to a ledger of account balances. Implementing money should therefore be no harder than agreeing on an ordered sequence of transactions.

The distributed computing community has spent more than three decades actively researching "consensus protocols" that enable a network of computers to reach agreement. The problems here are notoriously subtle, especially since networks can intermittently fail, and some of the computers in the network might be compromised by a malicious attacker. Regardless, the field is mature, and there are many iconic protocols, such as Paxos [52] and PBFT [53], that provide strong guarantees under rigorously-defined models. It's surprising that the first cryptocurrency differs so significantly from this template!

What fundamentally distinguishes Bitcoin's design from traditional consensus protocols is the adversarial setting it is designed for. Bitcoin operates on top of an entirely anonymous peer-to-peer network. Anyone that wants to can participate as a miner, and there is no reliance on "real names." In contrast, most traditional consensus protocols rely heavily on the use of individual identifiers, and have a basic structure of collecting "majority votes." Without having

an authoritative list of registered participants, it's hard to imagine how to securely count votes. Systems that allow parties to generate their own arbitrary identities are typically vulnerable to the Sybil attack [54], in which selfish or malicious processes claim a large number of extra identities, stuffing the ballot box with votes.

In place of identities, Bitcoin's security can be based on an alternative assumption about the allocation of computational resources. The network reaches consensus through a process of competitive puzzle solving, and security is guaranteed as long as the protocol-following participants can "outcompute" any attacker. This alternative assumption is well-suited to the anonymous decentralized setting — anyone can potentially obtain and contribute computing power.

In this chapter, we formally construct (a simplified form of) the Nakamoto consensus protocol that embodies these novel aspects. We define a network model that captures the difficult anonymous setting: our model allows participants to communicate anonymously by publicly broadcasting messages. We show that the Nakamoto protocol achieves a strong notion of consensus. This contribution is the first security proof of Bitcoin against arbitrary attackers.

In formalizing this protocol, we develop a general abstraction for a key component of Nakamoto's protocol, the computational puzzle scheme. We call this abstraction a "scratch-off puzzle," generalizing the particular puzzle construction used in Bitcoin. Our abstraction captures the essential security requirements for an appropriate puzzle — in a nutshell, it requires that an adversary cannot take any shortcuts, and that when many individual participants make independent and concurrent efforts to solve the puzzle, their efforts are efficiently combined. Our main result in this chapter is a proof that the Nakamoto Consensus protocol, instantiated with an arbitrary scratch-off puzzle, achieves consensus as long as the adversary controls a sufficiently small fraction of the network's overall hashpower. The scratch-off puzzle abstraction plays a central role in Chapters 3 and 4, where we construct alternative puzzles that also fit this definition, and therefore meet the minimum security requirements.

3.1 Related Work

Okun [55] studied distributed-computing problems in various models of anonymous networks. The weakest model he considered, the "port-unaware" model, is most similar to ours. However, our model is weaker still: in the port-unaware model, each corrupt process can send at most one message to a correct process in each round, whereas in our model the adversary can deliver an arbitrary number messages to honest parties. Okun's positive result crucially relies on this message bound, and is thus inapplicable to our model. Similarly our communication model is related to (but weaker than) other models such as partial broadcast [56] and homonymous networks [57].

Cryptographic puzzles (also called *proofs of work*) have been analyzed and proposed for a variety of other purposes, including timed-release encryption [58], spam prevention [59–61], DoS resistance [62, 63], and defense against Sybil attacks [64].

Aspnes et al. [65] studied Byzantine agreement without a PKI in a model where computational puzzles can be solved at some bounded rate. This work, however, also assumes pre-existing authenticated channels between honest parties.

Our work extends the informal security argument made by Nakamoto [25]. Nakamoto's argument also follows from an analysis of Poisson distributions, but it only defends against a particular attack strategy. Our proof is stronger, since our model captures any feasible attack strategy with the resources given. For example, our attack model proves that even the "selfish mining" attacker of Eyal and Sirer [66], as well as "stubborn miners" [67] and related strategies [68], cannot undermine the basic consensus guarantees. Finally we note that several subsequent works [20, 21] adopt our basic model of anonymous message diffusion, but analyze a more faithful representation of the actual Bitcoin protocol. We stress that our goal is not to model Bitcoin specifically, but rather to provide a complete but abstract construction of a cryptocurrency system.

Several (subsequent) works also analyze new consensus protocols inspired by Bitcoin and based on similar assumptions to ours [69, 70]. These incorporate the most significant features of our model, including anonymous message diffusion and the assumption that each process makes a bounded number of random oracle queries per round. The protocol of Dziembowski et al. [70] avoids relying on a beacon value to begin the protocol; instead, each party generates a challenge, and the puzzle is derived a combination of all such challenges. For very large networks, this may impose a significant additional communication cost. In the Nakamoto consensus protocol, only lucky parties that find a puzzle solution must broadcast a message, and the number of such puzzles is independent of n (i.e., depends only on the ratio of f to n). While scratch-off puzzles are inherently parallelizable, Miller et al. [69] show how to implement pseudonymous broadcast primitives using inherently sequential puzzles instead.

Finally, several authors have proposed deriving a timelock encryption scheme from a scratch-off puzzle scheme [71, 72]. Here, a puzzle instance is used as a public encryption key, and a solution to the scratch-off puzzle can be used as a decryption key. These schemes rely on methods that are currently impractically expensive methods (generic program obfuscation).

3.2 Scratch-off Puzzles

As explained in Chapter 1, Bitcoin miners compete to solve computational puzzles, and whoever solves a puzzle first in each epoch receives a reward. As there is no shortcut to solving this puzzle, for an attacker to dominate the network would require the attacker to expend more computational resources than the rest of the honest participants combined. Although the Bitcoin puzzle is commonly referred to as a *proof-of-work* puzzle, the requirements of the puzzle are somewhat different than existing definitions for proof-of-work puzzles [73–76].

In this section we provide a formal definition of the basic requirements of the Bitcoin puzzle. We call any puzzle satisfying these requirements a *scratch-off puzzle*.[1]

The first requirement captured by our definition is that the attacker should not be able to take "shortcuts" to solve the puzzle faster than honest parties. We express this using an explicit parameter γ, such that the attacker solves puzzles at most γ more efficiently than an honest party following the ordinary algorithm.

The second requirement is that the puzzles must be efficiently solved with parallel effort. In Bitcoin, all of the independent mining participants simultaneously attempt to solve puzzles, and only communicate in order to exchange solutions that they find. Even though the parties do not coordinate, they must not waste much effort on redundant work. This requirement is unique to our scratch-off puzzle definition; a traditional proof-of-work puzzle or client puzzle might only be solvable through a sequential computation carried out by a single party [75, 76]. For example, Rivest's classic time-lock puzzle scheme can only effectively be solved through sequential repeated squaring [58]. Since this procedure is deterministic and inherently sequential, multiple independent participants would not be able to solve this any faster than one working alone. In general, scratch-off puzzle constructions that satisfy this property are designed so that they can only be solved through a brute force search of a large solution space. Since the search space is large, the independent parties search in randomized regions and do not overlap in their search.

We also require puzzle solutions to be nonmalleable, in the sense that seeing solutions for an arbitrary number of set of puzzles does not help the adversary solve any *different* puzzle not included in that set. Each distinct puzzle should require fresh work to solve. If this property were not satisfied, then the worst-case scenario is that an adversary could perform a large precomputation step, and thereafter solve each new puzzle with very little marginal effort. Stebila et al. pointed out that early definitions of client puzzles did not prevent this scenario [75].

Finally, in the Nakamoto consensus protocol, solving a puzzle enables the puzzle solver to determine which new transactions get committed, and in particular to choose the public key where the puzzle reward is sent. We incorporate this in our definition by adding an arbitrary "payload" message to the solver routine. For this to be useful, we must require that the payload of a ticket is bound to it in the following sense: if an honest party publishes a ticket associated to a payload m (e.g., containing a public key belonging to the party to whom the reward must be paid), the adversary should not gain any advantage in obtaining a puzzle solution associated with some different payload m^* for the same puz. This is because in Bitcoin, each epoch is defined by a globally known, unique puzzle instance puz; at most one winning ticket for puz and a payload message is accepted into the blockchain; and a user who solves a puzzle only receives the reward if their message is the one that is associated. If an adversary can easily modify a victim's winning ticket to be associated to a different payload of its choice, then the adversary can listen for when the victim's ticket is first announced in the network, and then immediately start propagating the modified ticket (e.g., containing its own public key for the reward payment) and attempt to outrace the victim. It is possible that the network will now deem the adversary as the winner

[1] The terms "scratch-off puzzle" and "winning ticket" are motivated by the observation that Bitcoin's coin minting process resembles a lottery with scratch-off ticket, wherein a participant expends some effort to learn if he holds a winning ticket.

of this epoch—this is especially true if the adversary has better network connectivity than the victim (as described by Eyal and Sirer [66]). Intuitively, seeing a puzzle solution output by an honest party does not help noticeably in producing a solution associated to a different payload m^*.

In our definition, the difficulty, non-malleability, and non-transferability properties are all bundled into a single property called *incompressibility.* This property is defined by a game that allows the attacker to query a puzzle-solving oracle with arbitrary puzzles and payloads. The oracle queries do not help the attacker to solve different puzzles, or even previous puzzles with different payloads.

3.2.1 Definition of Scratch-Off Puzzles

A scratch-off puzzle is parameterized by parameters $(d, \underline{t}, t_0, \gamma)$ where, informally speaking, $\underline{t}$ denotes the amount of work needed to attempt a single puzzle solution, γ refers to the maximum amount by which an adversary can speed up the process of finding solutions, d affects the average number of attempts to find a solution, and t_0 denotes the initialization overhead of the algorithm. We typically assume that $t_0 \ll 2^d \underline{t}$, where $2^d \underline{t}$ is the expected time required to solve a puzzle.

Definition 3.1. A scratch-off puzzle is parameterized by parameters $(d, \underline{t}, t_0, \gamma)$, and consists of the following algorithms (satisfying properties explained shortly):

- $\mathcal{G}(1^\lambda) \rightarrow \mathsf{params}$: sets up public parameters for a puzzle scheme. The params are an implicit argument to each of the remaining functions, but we omit these for brevity.
- $\mathsf{Work}(\mathsf{puz}, m) \rightarrow \mathsf{ticket}$: The Work algorithm takes an arbitrary puzzle instance $\mathsf{puz} \in \{0,1\}^\lambda$, and some payload message m. The Work algorithm makes continual effort to find a puzzle valid puzzle solution, terminating once one is found. In all our constructions, this effort consists of a brute-force random search over the solution space, where each marginal attempt takes $\underline{t}$ steps (the *unit scratch time*), and the overhead for initialization and finalization is t_0.
- $\mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket}) \rightarrow \{0,1\}$: checks if a ticket is valid for a specific instance puz and payload m. If ticket passes this check, we refer to it as a *winning ticket* for (puz, m).

Intuitively, the honest Work algorithm makes repeated scratch attempts, and each attempt has probability 2^{-d} of yielding a winning ticket, where d is called the puzzle's difficulty parameter. We will henceforth use the notation

$$\zeta_\ell(t, d) := 1 - \sum_{i \in [\ell]} \binom{t}{i} 2^{-di} (1 - 2^{-d})^{t-i}$$

to denote the probability of producing at least ℓ successes after t independent Bernoulli trials, each with independent probability 2^{-d} of success. We will soon use these probabilities as a lower bound for the success of uncorrupted processes, and as upper-bound for the success of an

adversary. Finally we use $\mathsf{Work}_t(\mathsf{puz}, m)$ to denote running $\mathsf{Work}(\mathsf{puz}, m)$ for at most $t \cdot \underline{t} + t_0$ steps, and returning $\perp$ if it has not terminated normally.

A scratch-off puzzle must satisfy three requirements:

1. **Correctness.** For any (puz, m, t), if $\mathsf{Work}_t(\mathsf{puz}, m)$ outputs $\mathsf{ticket} \neq \perp$, then $\mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket}) = 1$.

2. **Parallel feasibility.** Solving scratch-off puzzles is feasible using the honest Work algorithm, even if the puzzles are adversarially chosen, and moreover this effort can be parallelized without much loss. More formally, for any $q = \mathsf{poly}(\lambda)$, $t = \mathsf{poly}(\lambda)$, and for any polynomial time adversary $\mathcal{A}$ that chooses puzzle instances,

$$\Pr\left[\begin{array}{l} \mathsf{params} \leftarrow \mathcal{G}(1^\lambda), \\ \{\mathsf{puz}_i, m_i\}_{i \in [q]} \leftarrow \mathcal{A}, \\ \forall i \in [q] : \mathsf{ticket}_i \leftarrow \mathsf{Work}_t(\mathsf{puz}_i, m_i) : \\ \exists i \in [q] : \mathsf{Verify}(\mathsf{puz}_i, m_i, \mathsf{ticket}_i) \end{array}\right] \geq \zeta_1(qt, 2^{-d}) - \mathsf{negl}(\lambda).$$

   Intuitively, each unit scratch attempt, taking time $\underline{t}$, has probability 2^{-d} of finding a winning ticket. Therefore, if q (potentially parallel) processes each makes t, the probability of finding one winning ticket overall is $\zeta_1(qt, 2^{-d}) \pm \mathsf{negl}(\lambda)$.

3. **γ-Incompressibility.** Roughly speaking, the work for solving a puzzle must be incompressible, in the sense that even the best adversary can speed up the process of finding puzzle solutions by at most a factor of γ. Furthermore, this holds even if the adversary has (polynomially-bounded) oracle access to the honest algorithm — that is, observing the output of $\mathsf{Work}(\mathsf{puz}, m)$ does not help the adversary solve a different puzzle $\mathsf{puz}' \neq \mathsf{puz}$, nor does it help the adversary find solutions to puz associated with a different payload $m^* \neq m$. More formally, a scratch-off puzzle is γ-incompressible (where $\gamma \geq 1$) if for any $\ell = \mathsf{poly}(\lambda)$ and any probabilistic polynomial-time adversary $\mathcal{A}$ taking at most $t \cdot \underline{t}$ steps,

$$\Pr\left[\begin{array}{l} \mathsf{params} \leftarrow \mathcal{G}(1^\lambda), \\ \{\mathsf{puz}_i, m_i, \mathsf{ticket}_i\}_{i \in [\ell]} \leftarrow \mathcal{A}^{\mathsf{Work}} : \\ \text{all } \{\mathsf{puz}_i\}_{i \in [\ell]} \text{ are distinct, and} \\ \forall i \in [\ell] : \mathsf{Verify}(\mathsf{puz}_i, m_i, \mathsf{ticket}_i) = 1 \wedge (\mathsf{puz}_i, m_i) \notin Q \end{array}\right] \leq \zeta(\ell, \gamma t, 2^{-d}) \pm \mathsf{negl}(\lambda).$$

   where Q is the transcript of oracle queries made by $\mathcal{A}$ to Work. Ideally, we would like the compressibility factor γ to be as close to 1 as possible. When $\gamma = 1$, the honest Work algorithm is the optimal way to solve a puzzle.

Note that this definition allows the adversary to find multiple puzzle solutions for a single puzzle instance puz, potentially for varying payload messages, for just the cost of finding one. The definition would be stronger if we bounded the adversary's advantage for distinct pairs $\{(\mathsf{puz}_i, m_i)\}$, whereas instead our definition only considers distinct puzzle instances $\{\mathsf{puz}_i\}$. Looking ahead, we will need to make use of this relaxation in Chapter 3 when constructing nonoutsourceable puzzles. Regardless, even with this relaxation our definition suffices to prove the correctness of the Nakamoto consensus protocol.

Let $\mathcal{H} : \{0,1\}^* \to \{0,1\}^\lambda$ be a hash function modeled as a random oracle. For measuring running time, we consider evaluating $\mathcal{H}$ to take t_{RO} steps. Furthermore, we do not count verification (e.g., given m, h, to check if $h = \mathcal{H}(m)$) towards the running time.

- $\mathcal{G}(1^\lambda)$: return $\perp$ (no setup is needed)
- $\mathsf{Work}(\mathsf{puz}, m)$:
  Loop indefinitely:

  Draw a random nonce, $s \xleftarrow{\$} \{0,1\}^\lambda$.
  Compute $h := \mathcal{H}(\mathsf{puz}\|m\|s)$
  If $h < 2^{\lambda-d}$ then return $\mathsf{ticket} := (s,h)$ and terminate. Otherwise, resume from the beginning of the loop.
- $\mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket})$:
  Parse ticket as (s,h), and check that $h = \mathcal{H}(\mathsf{puz}\|m\|\mathsf{ticket})$ and $h < 2^{\lambda-d}$.

FIGURE 3.1: The Bitcoin scratch-off puzzle.

Remarks on modeling oracles and running time. We assume, somewhat inelegantly, that verification is free. This assumption is also present in subsequent and independent models (Garay et al. [21] and Pass et al. [20]). It is justified by observing that computational resources used for verification are independent of the computational resources used for mining. In the case of Bitcoin mining equipment, most mining is performed using dedicated ASIC hardware, whereas the general purpose CPU is used for verification. Furthermore, in the schemes we consider, verification cost is independent of the overall puzzle difficulty. As the difficulty increases to accommodate more participants and more powerful mining equipment (while keeping the average time to solve a puzzle fixed at 10 minutes), verification cost remains unchanged. In practice, nontrivial verification costs result in a possible Denial-of-Service attack vector — the adversary could attempt to flood the communication network with bogus puzzle solutions. Various countermeasures are built into the actual peer-to-peer network: invalid puzzle solutions are not propagated by honest relay nodes, and any node that relays invalid puzzle solutions is disconnected by its peers.

We are careful to express running time as a construction-specific measure. Our proof of the Bitcoin puzzle assumes the random oracle model. However, later on, we construct a puzzle that involves zero-knowledge proofs. About NP statements — however, statements involving a random oracle are not in NP. Various efforts have been made to define notions of zero-knowledge proofs for languages involving oracles (e.g., [77, 78]). To sidestep these modeling difficulties, we therefore assume that the random oracle is instead instantiated with an actual cryptographic function (heuristically accepted to be secure) before applying zero-knowledge proofs.

3.2.2 The Bitcoin Scratch-Off Puzzle

We now show (as a sanity check) that the original Bitcoin puzzle satisfies our definition of scratch-off puzzles, and thus is indeed a generalization. An abstraction of Bitcoin's scratch-off puzzle is shown in Figure 3.1.[2] We assume that each random-oracle call takes time t_{RO}, and all other work in each iteration of Work takes $t_{\mathrm{other}} = O(\lambda)$ time. We then have the following:

Theorem 3.2. *The construction in Figure 3.1 is a $(d, \underline{t}, t_0, \gamma)$-scratch-off puzzle for arbitrary $d > 0$, where $\underline{t} = t_{\mathrm{other}} + t_{\mathrm{RO}}$, $t_0 = 0$, and $\gamma = t_{\mathrm{RO}}/(t_{\mathrm{RO}} + t_{\mathrm{other}})$.*

[2] This puzzle is also known as HashTrail [74–76], and was first used in Hashcash [59].

$\mathcal{F}_{\text{DIFFUSION}}$ **functionality**

Inherit $\mathcal{F}_{\text{BASIC}}$ (see Figure 2.1), including beacon values, scheduling events, and leaking messages to the adversary

Anonymous message diffusion

- on $\texttt{multicast}(m)$ from process $\mathcal{P}_i$:
  leak $\texttt{multicast}(m)$
  for each honest $\mathcal{P}_j$
   schedule delivery of $\texttt{multicast}(m)$ to $\mathcal{P}_j$ for $\mathsf{now} + \Delta$

FIGURE 3.2: Anonymous message diffusion functionality

Proof. The correctness proof is trivial. For γ-incompressibility, observe that for any adversary that makes only t random oracle calls, its probability of success in finding ℓ winning tickets is at most $\zeta_\ell(t, 2^{-d})$, since each distinct random oracle query has exactly an independent 2^{-d} chance of yielding a winning ticket. Since the preimage space for the oracle queries is prefixed by puz and m, oracle queries to Work do not help find solutions for other puzzle/message pairs. Since the honest Work algorithm takes $(t_{\text{RO}} + t_{\text{other}}) \cdot t$ time, this scratch-off puzzle is $t_{\text{RO}}/(t_{\text{RO}} + t_{\text{other}})$-incompressible. □

3.3 The Message Diffusion Model and Monte Carlo Consensus

Anonymous message diffusion. As explained in Chapter 1, Bitcoin is built atop a peer-to-peer broadcast system that allows users to anonymously publish puzzle solutions and transactions. In Figure 3.2 we define a functionality that provides an idealized form of this communication primitive. Any process is able to publish a message m by sending $\texttt{multicast}(m)$ to this functionality. Messages published this way are guaranteed to be delivered to each honest process within a maximum of Δ rounds. Except for this constraint, the adversary is given complete control over the order of message delivery. The identity of the source is hidden from the adversary, but the content of each message is immediately leaked to the adversary. The attacker can effectively "rush" (i.e., "front-run") by waiting for honest parties to submit messages, and then choosing and delivering the attacker's own messages ahead of the honest messages. Finally, the adversary is also permitted to deliver its own messages to some but not all of the parties. This means that an honest process that receives a message cannot tell for certain if every other process has also received that message.

Monte-Carlo Consensus. Below we provide a self-contained definition of a consensus protocol. Our definition is mostly standard: each process is provided an arbitrary string of input $\mathsf{proposed}_i$, and each (uncorrupted) process is required to output a common value corresponding chosen from among the inputs.

The main distinction compared to standard definitions is that the agreement property may be violated with non-zero probability. It is most common that randomized consensus protocols are defined as Las Vegas algorithms, terminating in finite time with probability 1, but guaranteeing

agreement unconditionally if they terminate [79]. Instead, our protocols are Monte Carlo algorithms, guaranteed to terminate in finite time, but may fail to reach agreement with negligible (though non-zero) probability. [3]

The validity property defined below says that the decided-upon value must correspond to one of the inputs with at least inverse polynomial probability (e.g., $\Omega(\frac{1}{\lambda})$). We choose this relatively weak goal as a convenience, in order to simplify the proofs in this chapter (particularly that of Theorem 3.8). Strengthening it is straightforward, as we later demonstrate in Chapter 5: we can simply run such a protocol $O(\lambda)$ times sequentially to boost the probability of succeeding at least once.

Definition 3.3. *(Monte Carlo Consensus)* A Monte Carlo consensus protocol for a set of n processes (f of which may be corrupted) begins with each correct process $\mathcal{P}_i$ receiving an input value $\mathsf{proposed}_i \in \{0,1\}^*$, and must satisfy the following properties:

- *(Termination)*: All correct processes must output a single value after a bounded time.
- *(Agreement)*: All correct processes must out *the same value*, except with negligible probability.
- *(Validity)*: The output value will be one of the inputs with inverse-polynomial probability.

3.4 The Nakamoto Consensus Protocol

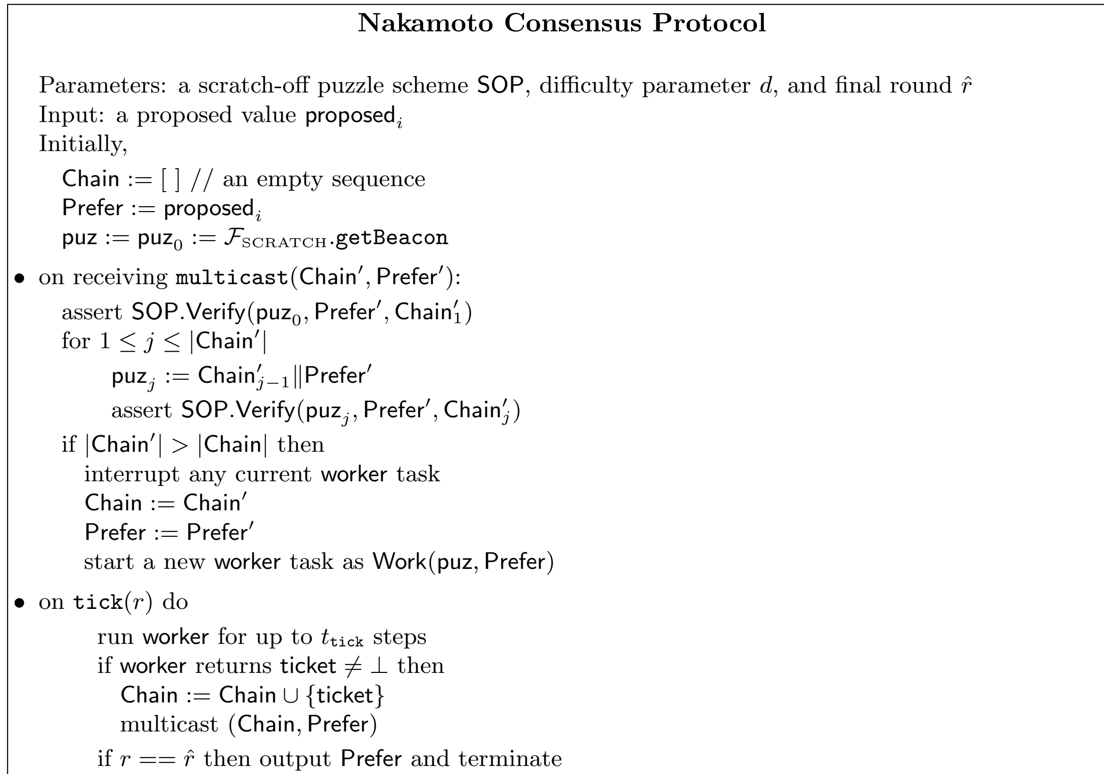
In Bitcoin, participants (called miners) continuously attempt to solve puzzles, where each solution is used to derive the challenge for the next puzzle instance, thereby forming a chain of puzzle solutions. Participants coordinate with each other according to a simple rule: the longest chain of puzzle solutions is considered the authoritative one. Each node works to extend the longest chain of puzzle solutions it knows, and when a node finds a new puzzle solution, it broadcasts this to the rest of the network. When the parameters are chosen correctly, this guarantees that the honest parties eventually agree on (a prefix of) their puzzle solution chains.

Unlike a traditional consensus protocol (including the Monte Carlo consensus definition presented earlier in Section 3.3), where processes must eventually terminate and output a final value, the actual Bitcoin protocol never terminates. Instead, it more closely resembles a "stabilizing consensus" protocol, wherein each process provides an output register reflecting the current most-likely value, and eventually this register stabilizes in the sense that the chance of it changing thereafter is negligible.

In Figure 3.3 we define an abstraction of the Bitcoin protocol that indeed terminates with a final value.[4] Each process $\mathcal{P}_i$ initially begins with an empty Chain of puzzle solutions, along with

[3] This is not a significant weakness in practice. A negligible probability of failure is inherent in any protocol whose safety relies on cryptography.

[4] Although the Bitcoin protocol never outputs a final value, users of the system often need to make irrevocable decisions. For example, a merchant must have confidence that a transaction is permanently committed before letting a customer walk out of the store with an item. The Bitcoin whitepaper recommends waiting for 6 puzzle solutions before considering a transaction committed, although this is not enforced by the system itself. Our consensus protocol can be thought of as implementing this policy (for some $O(\lambda)$ puzzles, not necessarily 6) for the system as a whole.



Nakamoto Consensus Protocol

Parameters: a scratch-off puzzle scheme SOP, difficulty parameter d, and final round $\hat{r}$
Input: a proposed value $\mathsf{proposed}_i$
Initially,
  $\mathsf{Chain} := [\,]$ // an empty sequence
  $\mathsf{Prefer} := \mathsf{proposed}_i$
  $\mathsf{puz} := \mathsf{puz}_0 := \mathcal{F}_{\text{SCRATCH}}.\mathtt{getBeacon}$
• on receiving $\mathtt{multicast}(\mathsf{Chain}', \mathsf{Prefer}')$:
  assert $\mathsf{SOP.Verify}(\mathsf{puz}_0, \mathsf{Prefer}', \mathsf{Chain}'_1)$
  for $1 \leq j \leq |\mathsf{Chain}'|$
    $\mathsf{puz}_j := \mathsf{Chain}'_{j-1} || \mathsf{Prefer}'$
    assert $\mathsf{SOP.Verify}(\mathsf{puz}_j, \mathsf{Prefer}', \mathsf{Chain}'_j)$
  if $|\mathsf{Chain}'| > |\mathsf{Chain}|$ then
    interrupt any current worker task
    $\mathsf{Chain} := \mathsf{Chain}'$
    $\mathsf{Prefer} := \mathsf{Prefer}'$
    start a new worker task as $\mathsf{Work}(\mathsf{puz}, \mathsf{Prefer})$
• on $\mathtt{tick}(r)$ do
    run worker for up to $t_{\mathtt{tick}}$ steps
    if worker returns $\mathsf{ticket} \neq \bot$ then
      $\mathsf{Chain} := \mathsf{Chain} \cup \{\mathsf{ticket}\}$
      multicast $(\mathsf{Chain}, \mathsf{Prefer})$
    if $r == \hat{r}$ then output Prefer and terminate

FIGURE 3.3: Nakamoto Consensus

their input value, $\mathsf{proposed}_i$, and outputs a final value after a fixed number of rounds $\hat{r}$. The first puzzle solution is determined by the current `beacon` value at the start of the protocol. In each `tick` of the clock, each process attempts to solve a puzzle solution that extends Chain by one solution. (Since only $t_{\mathtt{tick}}$ steps are allowed in clock tick but the Work routine runs until completion, we describe this effort as a background task that can be paused and resumed.) The jth puzzle solution is denoted Chain_j. When a process receives a `multicast` message containing a new Chain', it switches to this if Chain' is longer than the current one. Each puzzle solution chain is associated with a value Prefer, which corresponds to the payload message associated with each puzzle in the chain (all such payload messages must be consistent in a valid chain). At the beginning when the Chain is initially empty, processes Prefer their original input value. Once the final round $\hat{r}$ is reached, each process outputs Prefer as its final decision value.

In the remainder of this section, we will prove that the Nakamoto consensus protocol satisfies the requirements of a Monte Carlo consensus protocol. The puzzle difficulty parameter d and the number of rounds to wait $\hat{r}$ must be chosen with knowledge of the other model parameters, such that the message delay bound is small enough relative to the rate of puzzle solutions. We'll describe shortly how to instantiate these parameters.

Switching to the continuous Poisson process. So far, we have defined puzzle solving as a discrete process, where attempts to solve a puzzle are made in each discrete round. However, it is easier to derive bounds for the continuous-time Poisson process, rather than Bernoulli processes. Fortunately, the Bernoulli process converges to a Poisson process in the limit where time increments are very small, and therefore we can use this as an approximation.

The main idea is that we will consider the duration of a single clock tick (i.e., the time to compute $t_{\texttt{tick}}$ steps) as an infinitesimal dr, and consider the limit when $dr \to 0$. As we let dr vary, we wish to keep constant the expected (real-valued) time for the network to solve a puzzle. This is achieved by varying the puzzle difficulty 2^{-d} proportionally with dr.

We must also make a simplifying assumption for this to apply to our puzzle schemes, which is that the unit scratch cost, $\underline{t}$, is equal to the number of steps permitted during a clock tick, $t_{\texttt{tick}}$.[5]

A geometric distribution describes the number of trials in between each success in a Bernoulli process; the number of trials to have ℓ successes is a sum of ℓ geometric distributions. Similarly, an exponential distribution describes the (real-valued) interarrival time between events in a Poisson process, and the time to reach ℓ successes is a sum of ℓ exponential distributions. We now prove a lemma that allows us to switch between these approximations. We let $\bar{\mu}$ represent the expected amount of (real-valued) time to have one success in a sequence of Bernoulli trials, assuming that the time for each trial is dr (and the probability of success p in each trial varies proportionally with dr). Then in the limit as $dr \to 0$, the sum of ℓ such geometric distributions converges to a sum of ℓ exponential distributions each with expectation $\bar{\mu}$.

Lemma 3.4. *Let $\bar{\mu} > 0$ be a constant, and let $X_{\ell,dr} \sim \sum_{1 \le i \le \ell} Geom(p)dr$ be a sum of ℓ geometric distributions each with probability $p = dr/\bar{\mu}$ and scaled by dr. Also let $X_\ell \sim \sum_{1 \le i \le \ell} Exp(\bar{\mu})$ be a sum of ℓ exponential distributions with mean $\bar{\mu}$. Then X_ℓ serves as the limit of $X_{\ell,dr}$ as $dr \to 0$, in the sense that*

$$\forall r. \lim_{dr \to 0} \Pr[X_{\ell,dr} \le r] = \Pr[X_\ell \le r]$$

.

Proof. We first show the limit for the case that $\ell = 1$. That the limit also extends to the sum of ℓ identical distributions is trivial. Recall that $\zeta_1(n,p)$ denotes the probability of having at least 1 success out of n Bernoulli trials with success probability p. Therefore

$$\Pr[X_{1,dr} \le r] = \zeta_1(\frac{r}{dr}, p) = 1 - (1-p)^{\frac{r}{dr}}.$$

Substituting $p = dr/\bar{\mu}$, we have

$$\Pr[X_{1,dr} \le r] = 1 - (1 - \frac{dr}{\bar{\mu}})^{\frac{r}{dr}}.$$

Using the limit definition of $e^{-1} = \lim_{x\to\infty} \left(1 - \frac{1}{x}\right)^x$, and substituting $x = \frac{\bar{\mu}}{dr}$, we have

$$\lim_{dr \to 0} 1 - (1 - \frac{dr}{\bar{\mu}})^{\frac{r}{dr}} = \lim_{x \to \infty} 1 - ((1 - \frac{1}{x})^x)^{\frac{r}{\bar{\mu}}} = 1 - e^{-\frac{r}{\bar{\mu}}}.$$

[5] In practice, the assumption that the real time to complete a puzzle attempt, dr, is infinitesimal is justified because in the constructions we consider, the time to make one puzzle solving attempt is extremely small relative to the message propagation time and the average time to solve a puzzle. For example, in the real life deployment of Bitcoin, a puzzle solving attempt requires only hash function evaluation, which takes on the order of 10 microseconds on an ordinary CPU, whereas the average time to solve a puzzle in Bitcoin is 10 minutes.

This is exactly the CDF of the exponential distribution

$$\Pr[X_1 \leq r] = 1 - e^{-\frac{r}{\mu}}.$$

□

Bounding the puzzle solving rate. Our goal will be to prove that in the final round of the protocol, $\hat{r}$, all of the correct processes agree on a chain of puzzle solutions with a common prefix. To achieve this, we will prove two bounds about the rate at which the adversary can solve puzzles and the rate at which the honest parties can solve puzzles. We now define two processes that can serve as such bounds.

To reduce clutter, we assume without loss of generality that we measure time in units equal to the message delay bound plus the initialization/finalization overhead of work, such that $\Delta + t_0 = 1$. We will need to assume that the effective computational power of correct processes exceeds that of the adversary, in the sense they solve puzzles at a faster rate, even after accounting for the γ factor permitted by the scratch-off puzzle. [6] We introduce a parameter μ to represent the expected time needed for the network to find one puzzle solution, and let the puzzle difficulty 2^{-d} vary in order to maintain $\mu = \frac{2^d}{\gamma n} dr$. Since μ denotes (an upper bound for) the expected time for the network as a whole to find a puzzle solution, we use $\mu_B = \frac{n}{n-f}\gamma\mu$ to denote the expected time for a correct process to find a puzzle solution.

First, we define a process $\mathbf{A}$ as a Poisson arrival process with an expected interarrival time μ, where $\mathbf{A}_x$ represents the arrival time of the xth event.

$$\mathbf{A}_x \sim \sum_{1 \leq i \leq x} Exp(\mu) \tag{3.1}$$

We can show that $\mathbf{A}$ serves as an upper bound for the arrival rate of puzzle solutions found by the network as a whole.

Lemma 3.5. *Let $\mathbf{X}_r$ denote the total number of distinct puzzles with solutions, known to any uncorrupted process in an execution of the Nakamoto consensus protocol after time r. Then in the limit as $dr \to 0$, for any $x > 0$, we have $\Pr[\mathbf{X}_r \geq x] \leq \Pr[\mathbf{A}_x \leq r] + \mathsf{negl}(\lambda)$.*

Proof. This lemma follows from the γ-incompressibility property of the puzzle scheme SOP.

We first property to introduce an expression that bounds $\mathbf{X}$. Since the network takes at most $n \cdot \underline{t}$ computational steps during each clock tick, and $\frac{r}{dr}$ clock ticks occur during an interval of duration r, then applying the incompressibility property gives us the lower bound $\Pr[\mathbf{X}_r \geq x] < \zeta_x(\gamma n \frac{r}{dr}, 2^{-d}) + \mathsf{negl}(\lambda)$.

[6] The assumption that a majority of the network correctly follows the protocol is also present in Nakamoto's whitepaper [25], as well as the subsequent works of Garay et al. [21] and Pass et al. [20]. It would be more realistic to justify this assumption with an incentive-compatibility argument. We discuss incentive alignment in Chapter 3, although weakening this assumption remains an open challenge in general.

Recall that ζ denotes the probability of having at least x successes out of $\gamma n \frac{r}{dr}$ Bernoulli trials with success probability $2^{-d} = \frac{\gamma n}{\mu} dr$. Using Lemma 3.4, we can see that as dr approaches zero, this converges to a Poisson arrival process with expected interarrival time μ. □

Next we define a process $\mathbf{B}$ consisting of alternating phases of a Poisson arrival with expected arrival time μ_B, and a delay phase of constant time Δ, where $\mathbf{B}_x$ represents the time after completing x cycles through both phases.

$$\mathbf{B}_x \sim \sum_{1 \le i \le x} (\exp(\mu_B) + \Delta + t_0) \tag{3.2}$$

This process serves as a lower bound for the length of the puzzle chains known to each process.

Lemma 3.6. *In an execution of the Nakamoto consensus protocol, let $\mathbf{X}_r$ denote the minimum length of $|\mathsf{Chain}|$ known to any correct process at time r. Then as dr approaches zero, for any $x > 0$, we have $\Pr[\mathbf{X}_r \le x] \le \Pr[\mathbf{B}_x \ge r] + negl(\lambda)$.*

Proof. This lemma follows from the parallel feasibility property of the underlying puzzle scheme SOP. After subtracting the initialization and finalization overhead t_0, from an arbitrary point in time r_0, a correct process finds a new puzzle solution by time r with probability at least $\zeta_1((n-f)\frac{r-r_0}{dr}, 2^{-d})$. Recalling that ζ_1 describes the probability of having at least one success after $(n-f)\frac{r-r_0}{dr}$ Bernoulli trials, then by Lemma 3.4, as dr approaches 0, this converges to a Poisson arrival process with expected interarrival time μ_B. After finding a puzzle solution and sending it to the other processes, by at least the maximum message delay bound Δ, every process knows of a chain that is at least one larger. □

We can now use the processes $\mathbf{A}$ and $\mathbf{B}$ to justify the agreement property. If $\mathbf{B}_{\hat{r}} \ge x$, then that implies all clients decide on a value associated with a chain containing at least x puzzle solutions. If additionally $\mathbf{A}_{2x} > \hat{r}$, then fewer than $2x$ distinct puzzle solutions have been shown in total, and hence all processes must have output on a unique consistent value.

Instantiating the parameters. The Nakamoto consensus scheme must be instantiated with parameters $\hat{r}$ (the amount of time to wait until terminating), and d (the puzzle difficulty). We now show how to instantiate these parameters, along with the related values x and μ defined earlier, such that the necessary bounds $\mathbf{A}_{2x} > r > \mathbf{B}_x$ hold with high probability.

We will need to assume that the effective computational power of the uncorrupted processes is greater than half the power of the network overall, even when accounting for the γ advantage given to the adversary in the underlying puzzle, so $n - f \ge \frac{\gamma n}{2}$. We denote this relative advantage with δ,

$$\delta = \frac{2(n-f)}{\gamma n} > 1. \tag{3.3}$$

To choose our parameters, we divide up this advantage by thirds: one third each for deviation bounds on $\mathbf{A}_{2x}$ and $\mathbf{B}_x$ respectively, and one third to account for the message delay and initialization/finalization overhead incurred in $\mathbf{B}$. Intuitively, the puzzle should be difficult enough so

that correct processes often find and propagate puzzle solutions before two solutions are found overall. First, we define x in terms of δ and security parameter λ,

$$x = \lambda / D_{KL}(\delta^{1/3} \| 1) \tag{3.4}$$

where $D_{KL}(\mu_P \| \mu_Q)$ is the Kullback-Leibler divergence (KL-divergence) between exponential distributions P and Q with scale parameters μ_P and μ_Q respectively. [7]

We write out the formulas for two instances we will need later:

$$D_{KL}(\delta^{(\frac{1}{3})} \| 1) = \delta^{(\frac{1}{3})} - 1 - \log(\delta^{(\frac{1}{3})}) \tag{3.5}$$

$$D_{KL}(\delta^{-(\frac{1}{3})} \| 1) = \delta^{-(\frac{1}{3})} - 1 - \log(\delta^{-(\frac{1}{3})}) \tag{3.6}$$

Next we solve for $\hat{r}$ such that

$$\frac{\hat{r} - x}{x \mu_B} = \frac{2x\mu}{\hat{r}} = \delta^{(\frac{1}{3})}, \tag{3.7}$$

resulting in a quadratic equation that simplifies to

$$\hat{r} = \frac{4x}{4 - \delta^{(\frac{5}{3})}}. \tag{3.8}$$

After having determined $\hat{r}$ and x explicitly, we can solve for μ (and hence the puzzle difficulty d) by plugging these in to Equation 3.7.

Next we prove a lemma that these parameters are satisfactory. The proof is based on simple Chernoff bounds.

Lemma 3.7. *Let parameters μ, $\hat{r}$, and x be defined as in Equations 3.4, 3.7, and 3.8. As described above, let $\mathbf{A}$ be a Poisson arrival process with scale parameter μ; $\mathbf{A}_{2x}$ is the time of the $2x^{th}$ arrival. Let $\mathbf{B}$ be a process with alternating phases of (1) Poisson arrivals (scale parameter μ) followed by (2) constant time delay $\Delta + t_0 = 1$; $\mathbf{B}_x$ is the time at which $\mathbf{B}$ has completed x cycles through both phases. Then $\mathbf{A}_{2x} > \hat{r} > \mathbf{B}_x$ except for a negligible probability.*

Proof. We derive Chernoff bounds to give us the desired result. Consider $\mathbf{A}_1$, which is a single exponential distribution with scale parameter μ. The moment generating function of $\mathbf{A}_1$ is

$$\mathbf{E}[e^{t\mathbf{A}_1}] = (1 - t\mu)^{-1}. \tag{3.9}$$

The moment generating function for a sum of independent distributions is simply the product of the respective moment generating functions. Thus, for any x,

$$\mathbf{E}[e^{t\mathbf{A}_x}] = (1 - t\mu)^{-x}. \tag{3.10}$$

[7] The KL-divergence is often used to express the number of bits needed to distinguish between two probability distributions [80]. Analogously, we use this quantity to establish with high probability the separation between two distributions, $\mathbf{A}_{2x}$ and $\mathbf{B}_x$.

Process **B** is also a sum of independent variables, where the moment generating function for the distribution with constant value $\Delta + t_0 = 1$ is e^t. So,

$$\mathbf{E}[e^{t\mathbf{B}_x}] = (1 - t\mu_B)^{-x} e^{tx}. \tag{3.11}$$

Let **R** be a non-negative random variable. By Markov's inequality, $\Pr[R \geq \hat{r}] \leq \mathbf{E}[R]/\hat{r}$. This inequality provides a maximum probability of deviating from the expected value. The Chernoff bound technique is to apply this inequality to an exponential transformation on **R**. For any $t > 0$, we have

$$\Pr[\mathbf{R} \geq \hat{r}] = \Pr[e^{t\mathbf{R}} \geq e^{t\hat{r}}] \leq \mathbf{E}[e^{t\mathbf{R}}]/e^{t\hat{r}}. \tag{3.12}$$

A similar inequality exists for $\mathbf{R} \leq \hat{r}$. For any $t < 0$, we have

$$\Pr[\mathbf{R} \leq \hat{r}] = \Pr[e^{t\mathbf{R}} \geq e^{t\hat{r}}] \leq \mathbf{E}[e^{t\mathbf{R}}]/e^{t\hat{r}}. \tag{3.13}$$

In order to obtain the tightest bounds, we solve for t to minimize the probability. Using this technique, and by assuming $\mathbf{E}[\mathbf{A}_{2x}] > \hat{r} > \mathbf{E}[\mathbf{B}_x]$, we obtain the following bounds:

$$\Pr[\mathbf{B}_x \geq \hat{r}] \leq \exp(\hat{r} - x) \tag{3.14}$$

$$\Pr[\mathbf{A}_{2x} \leq \hat{r}] \leq \exp(\hat{r}) \tag{3.15}$$

Beginning with 3.14 and applying (3.7) then (3.5), we have an upper bound for **B**,

$$\begin{aligned}
\Pr[\mathbf{B}_x \geq \hat{r}] &\leq \exp\left(x - (\hat{r} - x)/\mu_B + x \log \frac{(\hat{r} - x)}{x\mu_B}\right) \\
&= \exp\left(x\left(1 - \frac{(\hat{r} - x)}{x\mu_B} + \log \frac{(\hat{r} - x)}{x\mu_B}\right)\right) \\
&= \exp\left(-x\left(\delta^{(\frac{1}{3})} - 1 - \log \delta^{(\frac{1}{3})}\right)\right) \\
&= \exp\left(-x D_{KL}(\delta^{(\frac{1}{3})} \| 1)\right) \\
&= e^{-\lambda}.
\end{aligned}$$

Finally, since $D_{KL}(\delta^{(\frac{1}{3})} \| 1) \geq D_{KL}(\delta^{-(\frac{1}{3})} \| 1)$ for all δ, we can easily provide a matching bound for **A** using (3.15), (3.7), and (3.6),

$$\begin{aligned}
\Pr[\mathbf{A}_{2x} \leq \hat{r}] &\leq \exp\left(2x - \hat{r}/\mu + 2x \log \frac{\hat{r}}{2x\mu}\right) \\
&= \exp\left(2x\left(1 - \frac{\hat{r}}{2x\mu} + \log \frac{\hat{r}}{2x\mu}\right)\right) \\
&= \exp\left(-2x\left(\delta^{-(\frac{1}{3})} - 1 - \log \delta^{-(\frac{1}{3})}\right)\right) \\
&= \exp\left(-2x D_{KL}(\delta^{-(\frac{1}{3})} \| 1)\right) \\
&= \exp\left(-x D_{KL}(\delta^{(\frac{1}{3})} \| 1)\right) \\
&= e^{-\lambda}.
\end{aligned}$$

These tail bounds complete our proof. □

We now prove the Nakamoto consensus protocol correct.

Theorem 3.8. *The Nakamoto Consensus protocol, when instantiated with a $(\underline{t}, \gamma, d, t_0)$-scratch-off puzzle, and with parameters $\hat{r}$ and d chosen as described in Equations 3.8, 3.4, and 3.7, and assuming the adversary corrupts no more than f processes where $\frac{2(n-f)}{\gamma n} > 1$, is a Monte Carlo consensus protocol as defined in 3.3.*

The termination property holds immediately — every process terminates after a fixed round $\hat{r}$.

The agreement property follows from Lemmas 3.5, 3.6, and 3.7, which establish that at time $\hat{r}$, each correct process outputs a value associated with a chain of puzzle solutions of at least length x, yet altogether the correct processes know of fewer than $2x$ distinct puzzle solutions in total. This implies that the value associated with x puzzle solutions is unique.

Finally, to prove the validity property, we must show that the unique value is one of the inputs to a correct process with a non-negligible probability. Here we rely again on the use of processes $\mathbf{A}$ and $\mathbf{B}$ as established in Lemmas 3.5 and 3.6. Suppose we initialize a counter to 0 at the beginning of the protocol. If a correct process finds and propagates a solution before two puzzle solutions are found in total (i.e., $\mathbf{B}_1 < \mathbf{A}_2$), then we increment the counter at time $\mathbf{B}_1$. Otherwise, we decrement the counter at time $\mathbf{A}_2$. In either case, after we modify the counter, we repeat the experiment, a total of $2x+1$ times. The value of this counter is a simple random walk. It follows from Equation 3.7 and the definitions of $\mathbf{A}$ and $\mathbf{B}$ that

$$\mathbf{E}[\mathbf{B}_1] = \mu_B + 1 < 2\mu = \mathbf{E}[\mathbf{A}_2], \tag{3.16}$$

and therefore the random walk is positive-biased. The final value is thus positive with probability better than 1/2 (since an odd number of steps have been taken, the final value is never exactly 0). When the final value is positive, then we know from the Ballot theorem [81] that with probability $\frac{1}{2x+1}$ the counter never returns to zero. When this holds, the correct processes converge to one of their inputs in the first step, and maintain the "lead" until the end, ultimately deciding on that value. □

Example and comparison with Bitcoin. We now give an example instantiation with concrete parameters. We stress that we have not attempted to optimize our choice of parameters for performance, but instead have aimed only to choose adequate parameters that let us prove asymptotic security. Regardless, we can compare our parameterization directly to that of Bitcoin.

In the Bitcoin whitepaper [25], Nakamoto provides concrete security estimates against a double-spend attack on a merchant, assuming that the merchant waits to observe a given number of puzzle solutions before accepting a payment. Against such an attacker with 10% of the network's hashpower (i.e., when $\delta = 1.8$), if the merchant waits for 5 blocks (an average of 50 minutes), then the probability of a successful attack is less than 0.1% (approximately $e^{-\lambda}$ where $\lambda = 7$). Taking $\lambda = 7$ as our security parameter, our instantiation (Equation 3.4) would require us to wait for at least $x \approx 340$ puzzle solutions, significantly more than in Bitcoin. By Equation 3.8, our protocol

calls for waiting for $\hat{r} \approx 1460$ communication rounds until terminating. Although Bitcoin does not explicitly express a bound for the communication delay, experimental measurements have shown that (at the current time) a Bitcoin block propagates to approximately 90% of the nodes within approximately 12 seconds [82]. Therefore, taking $\Delta \approx 12$ seconds, the total running time would be nearly five hours, while the average time to find one puzzle solution would be only $\mu \approx 22 seconds$.

Chapter 3

Nonoutsourceable Scratch-Off Puzzles to Deter Mining Coalitions

4.1 Overview

The most fundamental assumption made by decentralized cryptocurrencies is that *no single entity or adminstration wields a large fraction of the computational resources in the network*. Violation of this assumption can lead to severe attacks such as history revision and double spending which essentially nullify all purported security properties that are widely believed today.

However, two recent trends in mining – namely, *mining pools* and *hosted mining* – have led to the concentration of mining power, and have cast serious doubt on the well-foundedness of these fundamental assumptions that underly the security of Bitcoin-like cryptocurrencies. Specifically, mining pools exist because solo miners wish to hedge mining risks and obtain rewards at a more stable, steady rate. At several times over the past two years, the largest handful of mining pools have accounted for well over a third of the network's overall computing effort [26]. For example, recently the largest mining pool, GHash.IO, has even exceeded 50% of the total mining capacity.[1] Currently, Hosted mining, on the other hand, allows individuals to outsource their mining effort to one or a few large service providers. Hosted mining services have already emerged, such as Alydian [83], whose "launch day pricing was $65,000 per Terahash, and mining hosting contracts are available in 5 and 10 Th/sec blocks" [83]. Hosted mining is appealing because it can potentially reduce miners' cost due to economies of scale. Henceforth we will refer to both mining pools and hosted mining as mining coalitions.

Such large mining coalitions present a potential lurking threat to the security of Bitcoin-like cryptocurrencies. To exacerbate the matter, several recent works [66, 84] showed that it may

[1] See http://arstechnica.com/security/2014/06/bitcoin-security-guarantee-shattered-by-anonymous-miner-with-51-netw

be incentive compatible for a mining coalition to deviate from the honest protocol – in particular, Eyal and Sirer [66] showed that a mining concentration of about 1/3 of the network's mining power can obtain disproportionately large rewards by exhibiting certain "selfish mining" behavior.

While alternatives to centralized mining pools are well-known and have been deployed for several years, (such as *P2Pool*,[24] a decentralized mining pool architecture), these have unfortunatley seen extremely low user adoption (at the time of writing, they account for less than 2% of the network). Fundamentally, the problem is that Bitcoin's reward mechanism provides no particular incentive for users to use these decentralized alternatives.

Increasing understanding of these problems has prodded extensive and continual discussions in the broad cryptocurrency community, regarding how to deter such coalitions from forming and retain the decentralized nature of Bitcoin-like cryptocurrencies [27]. The community demands a technical solution to this problem.

4.1.1 Our Results and Contributions

Our work provides a timely response to this community-wide concern [27], providing the *first formally founded solution* to combat Bitcoin mining centralization. Our key observation is the following: an enabling factor in the growth of mining pools is a simple yet effective enforcement mechanism; members of a mining pool do not inherently trust one another, but instead submit cryptographic proofs (called "shares") to the other pool members (or to the pool operator), in order to demonstrate they are contributing work that can only benefit the pool (e.g., work that is tied to the pool operator's public key).

Strongly Nonoutsourceable puzzles. Our idea, therefore, is to disable such enforcement mechanisms in a cryptographically strong manner. a new form of proof-of-work puzzles which additionally guarantee the following:

> *If a pool operator can effectively outsource mining work to a worker, then the worker can steal the reward without producing any evidence that can potentially implicate itself.*

Intuitively, if we can enforce the above, then any pool operator wishing to outsource mining work to an untrusted worker runs the risk of losing its entitled mining reward, thus effectively creating a disincentive to outsource mining work (either in the form of mining pools or hosted mining). Our nonoutsourceable puzzle is broadly powerful in that it renders unenforceable even *external contractual agreements* between the pool operator and the worker. In particular, no matter whether the pool operator outsources work to the worker through a cryptocurrency smart contract or through an out-of-the-band legal contract, we guarantee that the worker can steal the reward without leaving behind evidence of cheating.

Technical insights. At a technical level, our puzzle achieves the aforementioned guarantees through two main insights:

P1: We craft our puzzle such that if a worker is doing a large part of the mining computation, it must possess a sufficiently large part of a "signing key" such that it can later sign over the reward to its own public key – effectively stealing the award from the pool operator;

P2: We offer a zero-knowledge spending option, such that a worker can spend the stolen reward in a way that reveals no information (including potential evidence that can be used to implicate itself).

As a technical stepping stone, we formulate a weaker notion of our puzzle referred to as a weakly nonoutsourceable puzzle. A weakly nonoutsourceable puzzle essentially guarantees property P1 above, but does not ensure property P2. In Section 4.3 we argue that weakly nonoutsourceable puzzles alone are inadequate to defeat mining coalitions, and in particular hosted mining. As a quick roadmap, our plan is to first construct a weakly nonoutsourceable puzzle, and from there we devise a generic zero-knowledge transformation to compile a weakly nonoutsourceable puzzle into a strongly nonoutsourceable one.

Community demand and importance of formal security. The community's demand for a nonoutsourceable puzzle is also seen in the emergence of new altcoins [85, 86] that (plan to) adopt their own home-baked versions of nonoutsourceable puzzles. Their solutions, however, offer only weak nonoutsourceability, and do not provide any formal guarantees. The existence of these custom constructions further motivates our efforts, and demonstrates that it is non-trivial to both formalize the security notions as well as design constructions with provable security. To date, our work provides *the only formally-founded solution*, as well as the *first strongly nonoutsourceable puzzle construction.*

4.2 Weakly Nonoutsourceable Puzzles

The Bitcoin scratch-off puzzle described in the previous section is amenable to secure outsourcing, in the sense that it is possible for one party (the *worker*) to perform mining work for the benefit of another (the *pool operator*) and to *prove* to the pool operator that the work done can only benefit the pool operator.

To give a specific example, let m be the public key of the pool operator; if the worker performs $2^{d'}$ scratch attempts, on average it will have found at least one value r such that $\mathcal{H}(\mathsf{puz}\|m\|r) < 2^{\lambda-d'}$. The value r can be presented to the pool operator as a "share" (since it represents a portion of the expected work needed to find a solution); intuitively, any such work associated with m cannot be reused for any other $m^* \neq m$. This scheme is an essential component of nearly every Bitcoin mining pool to date [23]; the mining pool operator chooses the payload m, and mining participants are required to present shares associated with m in order to receive participation credit. The rise of large, centralized mining pools is due in large part to the effectiveness of this mechanism.

We now formalize a generalization of this outsourcing protocol, and then proceed to construct puzzles that are *not* amenable to outsourcing (i.e., for which no effective outsourcing protocol exists).

4.2.1 Notation and Terminology

Pool operator and Worker. We use the terminology *pool operator* and *worker* referring respectively to the party outsourcing the mining computation and the party performing the mining computation. While this terminology is natural for describing mining pools, we stress that our results are intended to simultaneously discourage both mining pools and hosted mining services. In the case of hosted mining, the roles are roughly swapped; the cloud server performs the mining work, and the individuals who hire the service receive the benefit and must be convinced the work is performed correctly. We use this notation since mining pools are more well-known and widely used today, and therefore we expect the mining-pool oriented terminology to be more familiar and accessible.

Protocol executions. A protocol is defined by two algorithms $\mathcal{S}$ and $\mathcal{C}$, where $\mathcal{S}$ denotes the (honest) worker, and $\mathcal{C}$ the (honest) pool operator. We use the notation $(o_\mathcal{S}; o_\mathcal{C}) \leftarrow (\mathcal{S}, \mathcal{C})$ to mean that a pair of interactive Turing Machines $\mathcal{S}$ and $\mathcal{C}$ are executed, with $o_\mathcal{S}$ the output of $\mathcal{S}$, and $o_\mathcal{C}$ the output of $\mathcal{C}$.

In this paper we assume the pool operator executes the protocol program $\mathcal{C}$ correctly, but the worker may deviate arbitrarily.[2] We use the notation $(\mathcal{A}, \mathcal{C})$ to denote an execution between a malicious worker $\mathcal{A}$ and an honest pool operator $\mathcal{C}$. Note that protocol definition always uses the honest algorithms, i.e., $(\mathcal{S}, \mathcal{C})$ denotes a protocol or an honest execution; whereas $(\mathcal{A}, \mathcal{C})$ represents an execution.

4.2.2 Definitions

Outsourcing protocol. We now define a generalization of outsourced mining protocols, encompassing both mining pools and hosted mining services. Our definition of outsourcing protocol is broad – it captures any form of protocol where the pool operator and worker may communicate as interactive Turing Machines, and at the end, the pool operator may obtain a winning ticket with some probability. The protocol is parametrized by three parameters $t_\mathcal{C}, t_\mathcal{S}$, and t_e, which roughly models the pool operator's work, honest worker's work, and the "effective" amount of work during the protocol.

Definition 4.1. A $(t_\mathcal{S}, t_\mathcal{C}, t_e)$*-outsourcing protocol* for scratch-off puzzle $(\mathcal{G}, \mathsf{Work}, \mathsf{Verify})$, where $t_e < t_S + t_C$ and $t_c < t_e$, is a two-party protocol, $(\mathcal{S}, \mathcal{C})$, such that

- The pool operator's input is puz, and the worker's input is $\perp$.

[2]This is without loss of generality, and does not mean that we assume the mining pool operator is honest, since the protocol $(\mathcal{S}, \mathcal{C})$ may deviate from "honest" Bitcoin mining.

- The pool operator $\mathcal{C}$ runs in at most $t_{\mathcal{C}} \cdot \underline{t}$ time, and the worker $\mathcal{S}$ in at most $t_{\mathcal{S}} \cdot \underline{t}$ time.
- $\mathcal{C}$ outputs a tuple (ticket, m) at the end, where ticket is either a winning ticket for payload m or $\mathsf{ticket} = \perp$. Further, when interacting with an honest $\mathcal{S}$, $\mathcal{C}$ outputs a $\mathsf{ticket} \neq \perp$ with probability at least $\zeta_1(t_e, 2^{-d}) - \mathsf{negl}(\lambda)$.

Formally,

$$\Pr\left[\begin{array}{l} \mathsf{puz} \leftarrow \mathcal{G}(1^\lambda) \\ (\cdot; \mathsf{ticket}, m) \leftarrow (\mathcal{S}, \mathcal{C}(\mathsf{puz})) : \\ \mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket}) \end{array}\right] \geq \zeta_1(t_e, 2^{-d}) - \mathsf{negl}(\lambda).$$

The parameter t_e is referred to as the *effective billable work*, because the protocol $(\mathcal{S}, \mathcal{C})$ has the success probability of performing t_e unit scratch attempts. Note that it must be the case that $t_e < \mu(t_{\mathcal{S}} + t_{\mathcal{C}})$. Intuitively, an outsourcing protocol allows effective outsourcing of work by the pool operator if $t_e \gg t_{\mathcal{C}}$.

Note that this definition does not specify how the payload m is chosen. In typical Bitcoin mining pools, the pool operator chooses m so that it contains the pool operator's public key. However, our definition also includes schemes where m is jointly computed during interaction between $\mathcal{S}$ and $\mathcal{C}$, for example.

Weak nonoutsourceability. So far, we have formally defined what an outsourcing protocol is. Roughly speaking, an outsourcing protocol generally captures any possible form of contractual agreement between the pool operator and the worker. The outsource protocol defines exactly what the worker has promised to do for the pool operator, i.e., the "honest" worker behavior. If a worker is malicious, it need not follow this honest prescribed behavior. The notion of weak non-outsourceability requires that no matter what the prescribed contractual agreement is between the pool operator and the worker– as long as this agreement "effectively" outsources work to the worker– there exists an adversarial worker that can always steal the pool operator's ticket should the pool operator find a winning ticket during the protocol. Effectiveness is intuitively captured by how much effective work the worker performs vs. the work performed by the pool operator in the honest protocol. Note that there always exists a trivial, ineffective outsourcing protocol, where the pool operator always performs all the work by itself – in this case, a malicious worker will not be able to steal the ticket. Therefore, the weak non-outsourceability definition is parametrized by the effectiveness of the honest outsourcing protocol.

More specifically, the definition says that the adversarial worker can generate a winning ticket associated with a payload of its own choice, over which the pool operator has no influence. In a Bitcoin-like application, a natural choice is for an adversarial worker to replace the payload with a public key it owns (potentially a pseudonym), such that it can later spend the stolen awards. Based on this intuition, we now formally define the notion of a *weakly nonoutsourceable* scratch-off puzzle.

Definition 4.2. A scratch-off-puzzle is $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha, p_{\rm s})$-*weakly nonoutsourceable* if for every $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e)$-outsourcing protocol $(\mathcal{S}, \mathcal{C})$, there exists an adversary $\mathcal{A}$ that runs in time at most $t_{\mathcal{S}} \cdot \underline{t} + \alpha$, such that:

- Let $m^* \xleftarrow{\$} \{0,1\}^\lambda$. Then, at the end of an execution $(\mathcal{A}(\mathsf{puz}, m^*), \mathcal{C}(\mathsf{puz}))$, the probability that $\mathcal{A}$ outputs a winning ticket for payload m^* is at least $p_s\zeta_1(t_e, 2^{-d})$. Formally,

$$\Pr\left[\begin{array}{l}\mathsf{puz} \leftarrow \mathcal{G}(1^\lambda); m^* \xleftarrow{\$} \{0,1\}^\lambda \\ (\mathsf{ticket}^*; \mathsf{ticket}, m) \leftarrow (\mathcal{A}(\mathsf{puz}, m^*), \mathcal{C}(\mathsf{puz})) : \\ \mathsf{Verify}(\mathsf{puz}, \mathsf{ticket}^*, m^*)\end{array}\right] \geq p_s\zeta_1(t_e).$$

- Let view_h denote the pool operator's view in an execution with the honest worker $(\mathcal{S}, \mathcal{C}(\mathsf{puz}))$, and let view^* denote the pool operator's view in an execution with the adversary $(\mathcal{A}(\mathsf{puz}, m^*), \mathcal{C}(\mathsf{puz}))$. Then,

$$\mathsf{view}^* \stackrel{c}{\equiv} \mathsf{view}_h.$$

  When $\mathcal{C}$ interacts with $\mathcal{A}$, the view of the pool operator view^* is computationally indistinguishable from when interacting with an honest $\mathcal{S}$.

Later, when proving that puzzles are weakly nonoutsourceable, we typically construct an adversary $\mathcal{A}$ that runs the honest protocol $\mathcal{S}$ until it finds a ticket for m, and then transforms the ticket into one for m^* with probability p_s. For this reason, we refer to the adversary $\mathcal{A}$ in the above definition as a *stealing adversary* for protocol $(\mathcal{S}, \mathcal{C})$. In practice, we would like α to be small, and $p_s \leq 1$ to be large, i.e., $\mathcal{A}$'s run-time is not much different from that of the honest worker, but $\mathcal{A}$ can steal a ticket with high probability.

If the pool operator outputs a valid ticket for m *and* the worker outputs a valid ticket for m^*, then there is a race to determine which ticket is accepted by the Bitcoin network and earns a reward. Since the μ-incompressibility of the scratch-off puzzle guarantees the probability of generating a winning ticket associated with either m or m^* is bounded above by $\zeta_1(\mu(t_\mathcal{S} + t_\mathcal{C}), 2^{-d})$, the probability of the pool operator outputting a ticket — but not the worker — is bounded above by $\zeta_1(\mu(t_\mathcal{S} + t_\mathcal{C}), 2^{-d}) - p_s\zeta_1(t_e, 2^{-d})$.

4.2.3 A Weakly Nonoutsourceable Puzzle

In this section, we describe a weakly nonoutsourceable construction based on a Merkle-hash tree construction. We prove that our construction satisfies weak nonoutsourceability (for a reasonable choice of parameters) in the random oracle model. Informally, our construction guarantees that for *any* outsourcing protocol that can effectively outsource a fixed constant fraction of the effective work, an adversarial worker will be able to steal the puzzle with at least constant probability.

Our construction is formally defined in Figure 4.2, but here we provide an informal explanation of the intuition behind it.

Intuition. To solve a puzzle, a node first builds a Merkle tree with random values at the leaves; denote the root by digest. Then the node repeatedly samples a random nonce s, computes $h = \mathcal{H}(\mathsf{puz}\|s\|\mathsf{digest})$, and uses h to select q leaves of the Merkle tree and their corresponding

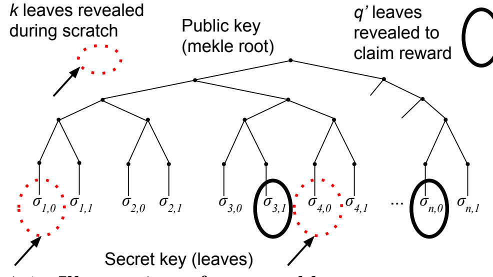


FIGURE 4.1: Illustration of our weakly-nonoutsourceable puzzle.

Let $\mathcal{H}, \mathcal{H}_2 : \{0,1\}^* \to \{0,1\}^\lambda$ denote random oracles.

- $\mathcal{G}(1^\lambda)$: no setup necessary
- $\mathsf{Work}(\mathsf{puz}, m)$:

  Construct Merkle tree. Sample L random strings $\mathsf{leaf}_1, \ldots, \mathsf{leaf}_L \xleftarrow{\$} \{0,1\}^\lambda$, and then construct a Merkle tree over the leaf_i's using $\mathcal{H}_2$ as the hash function. Let digest denote the root digest of this tree.

  Scratch. Repeat the following scratch procedure:

  * Draw a random nonce, $s \xleftarrow{\$} \{0,1\}^\lambda$.
  * Compute $h := \mathcal{H}(\mathsf{puz}\|m\|s\|\mathsf{digest})$, and use the value h to select q distinct leaves from the tree.
  * Let $B_1, B_2, \ldots, B_q$ denote the branches corresponding to the selected leaves. In particular, for a given leaf node, its *branch* is defined as the Merkle proof for this leaf node, including leaf node itself, and the sibling for every node on the path from the leaf to the root.
  * Compute $\sigma_h := \{B_i\}_{i\in[q]}$ in sorted order of i.
  * If $\mathcal{H}(\mathsf{puz}\|s\|\sigma_h) < 2^{\lambda-d}$ then record the solution pair (s, σ_h) and goto "Sign payload".

  Sign payload. Sign the payload m as follows:

  * Compute $h' := \mathcal{H}(\mathsf{puz}\|m\|\mathsf{digest})$, and use the value h' to select a set of $4q'$ distinct leaves from the tree such that these leaves are not contained in σ_h. From these, choose an arbitrary subset of q' distinct leaves. Collect the corresponding branches for these q' leaves, denoted $B_1, B_2, \ldots, B_{q'}$.
  * Let $\sigma'_h := \{B_i\}_{i\in[q']}$ in sorted order of i.
  * Return $\mathsf{ticket} := (\mathsf{digest}, s, \sigma_h, \sigma'_h)$.

- $\mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket})$:

  Parse $\mathsf{ticket} := (\mathsf{digest}, s, \sigma_h, \sigma'_h)$
  Compute $h := \mathcal{H}(\mathsf{puz}\|m\|s\|\mathsf{digest})$ and $h' := \mathcal{H}(\mathsf{puz}\|m\|\mathsf{digest})$.
  Verify that σ_h and σ'_h contain leaves selected by h and h' respectively.
  Verify that σ_h and σ'_h contain valid Merkle paths with respect to digest.
  Verify that $\mathcal{H}(\mathsf{puz}\|s\|\sigma_h) < 2^{\lambda-d}$.

FIGURE 4.2: A weakly nonoutsourceable scratch-off puzzle.

branches (i.e., the corresponding Merkle proofs). It then hashes those branches (along with puz and s) and checks to see if the result is less than $2^{\lambda-d}$.

Once successful, the node has a value s what was "difficult" to find, but is not yet bound to the payload message m. To effect such binding, a "signing step" is performed in which $h' = \mathcal{H}(\mathsf{puz}\|m\|\mathsf{digest})$ is used to select a set of $4q'$ leaf nodes (i.e., using h' a seed to a pseudorandom number generator). Any q' of these leaves, along with their corresponding branches, constitute a signature for m and complete a winning ticket. o Intuitively, this puzzle is weakly nonoutsourceable because in order for the worker to perform scratch attempts, it must

- either know a large fraction of the leaves and branches of the Merkle tree, in which case it will be able to sign an arbitrary payload m^* with high probability – by revealing q' out of the $4q'$ leaves (and their corresponding branches) selected by m^*,
- or incur a large amount of overhead, due to aborting scratch attempts for which it does not know the necessary leaves and branches,
- or interact with the pool operator frequently, in which case the pool operator performs a significant fraction of the total number of random oracle queries.

To formally prove this construction is weakly nonoutsourceable, we assume that the cost of the Work algorithm is dominated by calls made to random oracles. Thus, for simplicity, in the following theorems we equate the running time with the number of calls to the random oracle. However, the theorem can be easily generalized (i.e., relaxing by a constant factor) as long as the cost of the rest of the computation is only a constant fraction of the random-oracle calls.

Theorem 4.3. *The construction in Figure 4.2 is a $(d, 2t_{\mathrm{RO}}, t_0, \gamma)$-scratch-off puzzle, where $t_0 = 0$ and $\gamma = 1$, assuming that only the random oracle calls contribute to the running time of* Work.

Proof. Correctness and parallel feasibility proofs are trivial. We now prove incompressibility. We restrict our focus to the case where the adversary outputs only a single puzzle, message, and solution $(\mathsf{puz}, m^*, \mathsf{ticket})$ (i.e., when $\ell = 1$). We consider two cases. The first case is that the puzzle puz output by the adversary does not appear in the oracle query transcript Q. The second case is that the puzzle puz occurs in Q, but only along with a distinct message $m! = m^*$.

Case 1: Different puzzles. For any adversary $\mathcal{A}$ to obtain a winning ticket with $\zeta_1(t_e, 2^{-d})$ probability, the adversary must have made at least t_e good scratch attempts. A good scratch attempt consists of at least two random oracle queries, $h := \mathcal{H}(\mathsf{puz}\|s\|\mathsf{digest})$ and $\mathcal{H}(\mathsf{puz}\|s\|\sigma_h)$ such that the branches σ_h are consistent with h and digest. For each good scratch attempt the adversary must know at least some fraction of the branches, and have made random oracle calls to select the $O(\lambda)$ branches that are consistent with the digest. Without calling the random oracle to select the leaves, except for negligible probability the adversary cannot guess the correct branches to fetch (which are then fed into the next another random oracle to compare against the difficulty).

Case 2: Same puzzle, different message. For non-transferability, we need to show that for any polynomial time adversary $\mathcal{A}$, knowing polynomially many honestly generated tickets to puz for payload $m_1, m_2, \ldots, m_\ell$ does not help noticeably in computing a ticket for m^* to puz, where $m^* \neq m_i$ for any $i \in [\ell]$.

The adversary $\mathcal{A}$ may output two types of tickets for m^*: 1) m^* uses the same Merkle digest as one of the m_i's; and 2) m^* uses a different Merkle digest not seen among the m_i's.

In the latter case, it is not hard to see that the adversary $\mathcal{A}$ can only compress the computation at best as the best incompressibility adversary. Therefore, it suffices to prove that no polynomial time adversary can succeed with the first case except with negligible probability. Below we prove that.

Notice that the honest Work algorithm generates a fresh Merkle digest every time it is invoked. Therefore, with the honest algorithm, each Merkle digest will only be used sign a single payload except with negligible probability. Since the number of leaves $L \geq q + 8q'$, there are at least $8q'$ leaves to choose from in the signing stage. q' of those will be revealed for signing a message m. The probability that the revealed q' leaves are a valid ticket for message $m' \neq m$ is bounded by $\binom{8q'}{3q'}/\binom{8q'}{4q'} \propto \exp(-c_2 q')$. If the adversary has seen honestly generated tickets for ℓ different payloads, by union bound, the probability that there exists a ticket, such that its q' revealed leaves constitute a valid signature for a different message m^* is bounded by $\ell \cdot \exp(-c_2 q')$. □

Theorem 4.4. *Let $q, q' = O(\lambda)$. Let the number of leaves $L \geq q + 8q'$. Suppose $d > 10$ and $t_e \cdot 2^{-d} < 1/2$. Under the aforementioned cost model, the above construction is a $(t_\mathcal{S}, t_\mathcal{C}, t_e, \alpha, p_s)$ weakly nonoutsourceable puzzle, for any $0 < \eta < 1$ s.t. $t_\mathcal{C} < \eta t_e$, $p_s > \frac{1}{2}(1 - \eta) - \mathsf{negl}(\lambda)$, and $\alpha = O(\lambda^2)$.*

In other words, if the pool operator's work $t_\mathcal{C}$ is a not a significant fraction of t_e, i.e., work is effectively outsourced, then an adversarial worker will be able to steal the pool operator's ticket with a reasonably big probability, and without too much additional work than the honest worker.

For simplicity of presentation, we prove this for the case when $\eta = 1/2$. It is trivial to extend the proof for general $0 < \eta < 1$. To summarize, we would like to prove the following: for a protocol $(\mathcal{S}, \mathcal{C})$, if no adversary $\mathcal{A}$ (running in time not significantly more than the honest worker) is able to steal the winning ticket with more than $\frac{1}{2}\zeta_1(t_e, 2^{-d})$ probability, then $t_\mathcal{C}$ must be a significant fraction of t_e, i.e., the pool operator must be doing a significant fraction of the effective work. This would deter outsourcing schemes by making them less effective.

If the ticket output at the end of the protocol execution contains a σ_h such that 1) the selected leaves corresponding to σ_h were not decided by a random oracle call during the execution; or 2) σ_h itself has not been supplied as an input to the random oracle during the execution, then this ticket is valid with probability at most 2^{-d}. For $(\mathcal{S}, \mathcal{C})$ to be an outsourcing protocol with t_e effective billable work, the honest protocol must perform a number of "good scratch attempts" corresponding to t_e. Every good scratch attempt queries the random oracle twice, one time to select the leaves, and another time to hash the collected branches. We now define the notion of a "good scratch attempt".

Definition 4.5. During the protocol $(\mathcal{S}, \mathcal{C})$, if either the pool operator or worker makes the two random oracle calls $h := \mathcal{H}(\mathsf{puz}\|s\|\mathsf{digest})$ and $\mathcal{H}(\mathsf{puz}\|s\|\sigma_h)$ for a set of collected branches σ_h that is consistent with h and digest, this is referred to as a *good scratch attempt.* Each good scratch attempt requires calling the random oracle twice – referred to as scratch oracle calls.

Without loss of generality, we assume that in the honest protocol, if a good scratch attempt finds a winning ticket, the pool operator will accept the ticket. This makes our proof simpler because all good scratch attempts contribute equally to the pool operator's winning probability. If this is not the case, the proof can be easily extended – basically, we just need to make a weighted version of the same argument. For each good scratch attempt, there are two types of random oracle calls. Type 1 calls select the leaves. Type 2 calls hash the collected branches. Assume in the extreme case that the worker makes all the Type 1 calls (which accounts for 1/2 of all

work associated with good scratch attempts). Now consider Type 2 work which constitutes the other half: for each good scratch attempt, if the worker knows $< 1/3$ fraction of leaves of the corresponding tree before the Type 1 random oracle call for selecting the leaves, then the pool operator must have done at least one unit of work earlier when creating the Merkle tree digest. This is because if the worker knows $< 1/3$ fraction of the leaves before leaves are selected, then the probability that the selected leaves all fall into the fraction known by the worker is negligible. Since this is a good scratch attempt, for the selected leaves that the worker does not know, the pool operator must then know the leaves and the corresponding Merkle branches. This means that the pool operator earlier called the random oracle on those leaves to construct the Merkle digest.

If we want the pool operator's total work to be within 1/2 of the total effective work, then for at least 1/2 of good scratch attempts, the worker must know at least 1/3 of leaf nodes before the Type 1 oracle is called to select the leaves.

Suppose that $d > 10$ is reasonably large and that $t_e \cdot 2^{-d} < 1/2$. In this case, the probability that two or more tickets are found within t_e good attempts are a constant fraction smaller than the probability of one winning ticket being found. If for at least 1/2 of the good scratch attempts, the worker knows at least 1/3 fraction of leaves before leaves are selected, then an adversarial worker $\mathcal{A}$ would be able to steal the ticket with constant probability given that a winning ticket is found. To see this, first observe that the probability that a single winning ticket is found is a constant fraction of $\zeta_1(t_e, 2^{-d})$. Conditioned on the fact that a single winning ticket is found, the probability that this belongs to an attempt that the worker knows $> 1/3$ leaves before leaves are selected is constant. Therefore, it suffices to observe the following fact.

Fact 1. For a good scratch attempt, if a worker knows $> 1/3$ fraction of leaves before leaves are selected, then conditioned on the fact that this good scratch attempt finds a winning ticket, the worker can steal the ticket except with probability proportional to $\exp(-cq')$ for an appropriate positive constant c.

Proof. By a simple Chernoff bound.

The argument is standard. In expectation, among the selected $4q'$ leaves, the worker knows 1/3 fraction of them. Further, the worker only needs to know 1/4 fraction of them to steal the ticket. The probability that the worker knows less than q' out of $4q'$ leaves can be bounded using a standard Chernoff bound, and this probability is upper bounded by $\exp(-q'/27)$. □

4.3 Strongly Nonoutsourceable Puzzles

In the previous section, we formally defined and constructed a scheme for weakly nonoutsourceable puzzles, which ensure that for any "effective" outsourcing protocol, there exists an adversarial worker that can steal the pool operator's winning ticket with significant probability, should a winning ticket be found. This can help deter outsourcing when individuals are expected to behave selfishly.

One critical drawback of the weakly nonoutsourceable scheme (and, indeed, of Permacoin [5]) is that a stealing adversary may be detected when he spends his stolen reward, and thus might be held accountable through some external means, such as legal prosecution or a tainted public reputation.

For example, a simple detection mechanism would be for the pool operator and worker to agree on a $\lambda/2$-bit prefix of the nonce space to serve as a watermark. The worker can mine by randomly choosing the remaining $\lambda/2$-bit suffix, but the pool operator only *accepts* evidence of mining work bearing this watermark. If the worker publishes a stolen puzzle solution, the watermark would be easily detectable.

Ideally, we should enable the stealing adversary to evade detection and leave no incriminating trail of evidence. Therefore, in this section, we define a "strongly nonoutsourceable" puzzle, which has the additional requirement that a stolen ticket cannot be distinguished from a ticket produced through independent effort.

Definition 4.6. A puzzle is $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha, p_s)$-*strongly nonoutsourceable* if it is $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha, p_s)$-weakly nonoutsourceable, and additionally the following holds:

For any $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e)$-outsourcing protocol $(\mathcal{S}, \mathcal{C})$, there exists an adversary $\mathcal{A}$ for the protocol such that the stolen ticket output by $\mathcal{A}$ for payload m^* is computationally indistinguishable from a honestly computed ticket for m^*, even given the pool operator's view in the execution $(\mathcal{A}, \mathcal{C})$. Formally, let $\mathsf{puz} \leftarrow \mathcal{G}(1^\lambda)$, let $m^* \xleftarrow{\$} \{0,1\}^\lambda$. Consider a protocol execution $(\mathcal{A}(\mathsf{puz}, m^*), \mathcal{C}(\mathsf{puz}))$: let view^* denote the pool operator $\mathcal{C}$'s view and ticket^* the stolen ticket output by $\mathcal{A}$ in the execution. Let ticket_h denote an honestly generated ticket for m^*, ($\mathsf{ticket}_h := \mathsf{WorkTillSuccess}(\mathsf{puz}, m^*)$), and let view_h denote the pool operator's view in the execution $(\mathcal{S}, \mathcal{C}(\mathsf{puz}))$. Then,

$$(\mathsf{view}^*, \mathsf{ticket}^*) \stackrel{c}{\equiv} (\mathsf{view}_h, \mathsf{ticket}_h)$$

Recall that in Bitcoin, the message payload m typically contains a Merkle root hash representing a set of new transactions to commit to the blockchain in this round, including the public key to which the reward is assigned. Thus to take advantage of the strongly nonoutsourceable puzzle, the stealing worker should bind its substituted payload m^* to a *freshly generated* public key for which it knows the corresponding private key. It can then spend its stolen reward anonymously, for example by laundering the coins through a mixer[87].

In Figure 4.3, we present a generic transformation that turns any weakly nonoutsourceable puzzle into a strongly nonoutsourceable puzzle. The strengthened puzzle is essentially a *zero-knowledge* extension of the original – a ticket for the strong puzzle is effectively a proof of the statement "I know a solution to the underlying puzzle."

Theorem 4.7. *If* $(\mathsf{GenKey}', \mathsf{Work}', \mathsf{Verify}')$ *is a* $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha, p_s)$ *weakly nonoutsourceable puzzle, then the puzzle described in Figure 4.3 is a* $(t_{\mathcal{S}}, t_{\mathcal{C}}, t_e, \alpha + t_{enc} + t_{NIZK}, p_s - \mathsf{negl}(\lambda))$ *strongly nonoutsourceable puzzle, where* $t_{enc} + t_{NIZK}$ *is the maximum time required to compute the encryption and NIZK in the honest* Work *algorithm.*

Let NIZK be a non-interactive zero-knowledge proof system. Also assume that $\mathcal{E} = (\mathsf{Key}, \mathsf{Enc}, \mathsf{Dec})$ is a CPA-secure public-key encryption scheme.

Let $(\mathcal{G}', \mathsf{Work}', \mathsf{Verify}')$ be a weakly nonoutsourceable scratch-off puzzle scheme. We now construct a strongly nonoutsourceable puzzle scheme as below.

- $\mathcal{G}(1^\lambda)$: Run the puzzle generation of the underlying scheme $\mathsf{puz}' \leftarrow \mathcal{G}'(1^\lambda)$. Let $\mathsf{crs} \leftarrow \mathsf{NIZK}.\mathcal{K}(1^\lambda)$; and let $(\mathsf{sk}_\mathcal{E}, \mathsf{pk}_\mathcal{E}) \leftarrow \mathcal{E}.\mathsf{Key}(1^\lambda)$. Output $\mathsf{puz} \leftarrow (\mathsf{puz}', \mathsf{crs}, \mathsf{pk}_\mathcal{E})$
- $\mathsf{Work}(\mathsf{puz}, m, t)$:

  Parse $\mathsf{puz} := (\mathsf{puz}', \mathsf{crs}, \mathsf{pk}_\mathcal{E})$.
  $\mathsf{ticket}' \leftarrow \mathsf{Work}'(\mathsf{puz}', m, t)$,
  Encrypt $c \leftarrow \mathsf{Enc}(\mathsf{pk}_\mathcal{E}; \mathsf{ticket}'; \mathsf{s})$.
  Set $\pi \leftarrow \mathsf{NIZK}.\mathcal{P}(\mathsf{crs}, (c, m, \mathsf{pk}_\mathcal{E}, \mathsf{puz}'), (\mathsf{ticket}', s))$
  for the following NP statement:
  $\mathsf{Verify}'(\mathsf{puz}', m, \mathsf{ticket}') \wedge c = \mathsf{Enc}(\mathsf{pk}_\mathcal{E}; \mathsf{ticket}'; \mathsf{s})$
  Return $\mathsf{ticket} := (c, \pi)$.

- $\mathsf{Verify}(\mathsf{puz}, m, \mathsf{ticket})$;

  Parse $\mathsf{puz} := (\mathsf{puz}', \mathsf{crs}, \mathsf{pk}_\mathcal{E})$, and parse ticket as (c, π).
  Check that $\mathsf{Verify}(\mathsf{crs}, (c, m, \mathsf{pk}_\mathcal{E}, \mathsf{puz}'), \pi) = 1$.

FIGURE 4.3: A generic transformation from any *weakly nonoutsourceable* scratch-off puzzle to a *strongly nonoutsourceable* puzzle.

Proof. We first prove that this derived puzzle preserves weak nonoutsourceability of the underlying puzzle. Suppose that $(\mathcal{S}, \mathcal{C})$ is a $(t_\mathcal{S}, t_\mathcal{C}, t_e)$ outsourcing protocol. We will construct a suitable stealing adversary $\mathcal{A}$. To do so, we begin by deriving an outsourcing protocol $(\mathcal{S}', \mathcal{C}')$ for the underlying puzzle; the stealable property of the underlying puzzle allows to introduce an adversary $\mathcal{A}'$, from which we will derive $\mathcal{A}$. Let the outsourcing protocol $(\mathcal{S}', \mathcal{C}')$ for the underlying puzzle be constructed as follows: Suppose that $\mathcal{C}'$ is given the input puz'.

1. $\mathcal{S}'$ executes $\mathcal{S}$ unchanged.
2. $\mathcal{C}'$ first generates a keypair according to the encryption scheme, $(\mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{E}) \leftarrow \mathcal{E}.\mathsf{Key}(1^\lambda)$, runs the NIZK setup $\mathsf{crs} \leftarrow \mathsf{NIZK}.\mathcal{K}(1^\lambda)$, and then runs $\mathcal{C}$ using puzzle $(\mathsf{puz}', \mathsf{crs}, \mathsf{pk}_\mathcal{E})$.
3. If $\mathcal{C}$ outputs a ticket (c, π), then $\mathcal{C}'$ decrypts c and outputs $\mathsf{ticket}' \leftarrow \mathsf{Dec}(\mathsf{sk}_\mathcal{E}, \mathsf{c})$.

When run interacting with $\mathcal{S}$, the original pool operator $\mathcal{C}$ outputs a valid ticket with probability at least $\zeta_1(t_e) - \mathsf{negl}(\lambda)$; therefore, the derived pool operator $\mathcal{C}'$ decrypts a valid ticket with probability $\zeta_1(t_e) - \mathsf{negl}(\lambda)$. Since the underlying puzzle scheme is assumed to be weakly nonoutsourceable, there exists a stealing adversary $\mathcal{A}'$ running in time $t'_a = t_\mathcal{S} + \alpha$. We can thus construct an $\mathcal{A}$ that runs $\mathcal{A}'$ until it outputs $\mathsf{ticket}'_\mathcal{A}$ for the underlying puzzle, and then generate an encryption and a zero-knowledge proof.

We also need to prove it satisfies the additional indistinguishability property required from a strongly nonoutsourceable puzzle. This follows in a straightforward manner from the CPA-security of the encryption scheme, and the fact that the proof system is zero-knowledge. □

We next state a theorem that this generic transformation essentially preserves the non-transferability of the underlying puzzle.

Theorem 4.8. *If the underlying puzzle* $(\mathcal{G}', \mathsf{Work}', \mathsf{Verify}')$ *is* δ'*-non-transferable, then the derived puzzle through the generic transformation is* δ *non-transferable for*

$$\mu + \delta' \leq \frac{(\mu + \delta)t}{t \cdot \underline{t} + (t_{enc} + t_{nizk})\ell}$$

where t_{enc} *and* t_{nizk} *are the time for performing each encryption and NIZK proof respectively.*

Proof. We show that if an adversary $\mathcal{A}$ running in time t can win the non-transferability game of the derived puzzle, we can construct another adversary $\mathcal{A}'$ running in slightly more time than t that can win the non-transferability game of the underlying puzzle.

$\mathcal{A}'$ will call $\mathcal{A}$ as a blackbox. $\mathcal{A}'$ first receives a challenge for the underlying puzzle, in the form of puz', $m'_1, m'_2, \ldots, m'_\ell$, and winning tickets $\mathsf{ticket}'_1, \ldots, \mathsf{ticket}'_\ell$. Next, $\mathcal{A}'$ picks crs honestly, and picks $\mathsf{pk}_\mathcal{E}$ such that $\mathcal{A}'$ knows the corresponding $\mathsf{sk}_\mathcal{E}$. $\mathcal{A}'$ now gives to $\mathcal{A}$ the puzzle $\mathsf{puz} := (\mathsf{puz}', \mathsf{crs}, \mathsf{pk}_\mathcal{E})$. For $i \in [\ell]$, $\mathcal{A}'$ computes the zero-knowledge version $\mathsf{ticket}_i := (c_i, \pi_i)$. where c_i is an encryption of ticket'_i, and π_i is the NIZK as defined in Figure 4.3. $\mathcal{A}'$ gives $m'_1, \ldots, m'_\ell$ and $\mathsf{ticket}_1, \ldots, \mathsf{ticket}_\ell$ to $\mathcal{A}$ as well. Since $\mathcal{A}$ wins the non-transferability game, it can output a winning ticket (m^*, ticket^*) for puzzle puz with at least $\zeta_1((\mu + \delta)t, 2^{-d})$ probability where $t \cdot \underline{t}$ is the runtime of $\mathcal{A}$; further $m^* \neq m'_i$ for any $i \in [\ell]$.

$\mathcal{A}'$ now parses $\mathsf{ticket}^* := (c, \pi)$. $\mathcal{A}'$ then uses $\mathsf{sk}_\mathcal{E}$ to decrypt c and obtain ticket' – if the NIZK is sound, then ticket' must be a winning solution for the underlying puzzle puz' and payload m^* except with negligible probability – since otherwise one can construct an adversary that breaks soundness of the NIZK. Now, $\mathcal{A}'$ outputs (m^*, ticket') to win the non-transferability game. $\mathcal{A}'$ runs in $t \cdot \underline{t} + (t_{\text{enc}} + t_{\text{nizk}})\ell$ time, but wins the non-transferability game with probability at least $\zeta_1((\mu + \delta)t, 2^{-d})$. This contradicts the fact that the underlying puzzle is δ'-non-transferable. □

Cheap plaintext option. Although we have shown it is plausible for a stealing worker (with parallel resources) to compute the zero-knowledge proofs, this would place an undue burden on honest independent miners. However, it is possible to modify our generic transformation so that there are *two* ways to claim a ticket: the first is with a zero-knowledge proof as described, while the second is simply by revealing a plaintext winning ticket for the underlying weakly nonoutsourceable puzzle.

Chapter 4

Permacoin: Storage-based Scratch-off Puzzles to Recycle Mining Effort

5.1 Overview

Bitcoin mining is considerably expensive. At the current price and inflation rate, the Bitcoin network pays out over $1 million per day in the form of Bitcoin mining rewards. If we suppose that miners act like competitors in an efficient market, then we should predict that the costs of Bitcoin mining are close to the total value of the mining rewards. Further assuming electricity is the main cost, at $0.1 per KWH this would correspond to over 400 Megawatts - comparable to the output of a small gas power plant.[1]

The Bitcoin FAQ[2] has this to say on the matter: **Question**: Is [Bitcoin] not a waste of energy?

Answer: *Spending energy on creating and securing a free monetary system is hardly a waste.... [Banks] also spend energy, arguably more than Bitcoin would.*

Regardless of whether Bitcoin's resource costs are *justified*, we may still look to reduce them. At the time of writing, each Bitcoin puzzle solution requires about 2^{70} hash computations in expectation, and on average one solution is found every 10 minutes. The puzzle solutions themselves have no intrinsic use, but are valued only for their indirect role in securing the Bitcoin network. The Bitcoin FAQ also addresses the following question:

.

[1]Similar estimates can be found by observing the "hashpower" of the network ($2^7$0 hash computations per 10 minutes, today) and the power consumption of typical mining devices [13, 82, 88].

[2]Referenced 6 Apr. 2013 at https://en.bitcoin.it/wiki/FAQ.

Question: Why don't we use calculations that are also useful for some other purpose? **Answer**: *To provide security for the Bitcoin network, the calculations involved need to have some very specific features. These features are incompatible with leveraging the computation for other purposes.*

Indeed researchers have struggled to identify useful computational tasks outside Bitcoin, e.g., protein folding problems [89], that *also* have the predictable solution times and efficient public verifiability required for Bitcoin.

We propose a system, Permacoin , which shows how to modify Bitcoin to *repurpose* the computing resources that cryptocurrency miners invest for a general, useful goal and thus *reduce waste.*

5.1.1 Goal and approach

We show that *Bitcoin resources can be repurposed for other, more broadly useful tasks*, thereby refuting the widespread belief reflected in the Bitcoin FAQ. We propose a new scheme called *Permacoin.* The key idea in our scheme is to make a cryptocurrency in which mining depends upon *storage* resources, rather than *computation.*

Concretely, Permacoin is based around a modified SOP in which nodes in Bitcoin perform mining by constructing a *Proof of Retrievability (POR)* [90]. A POR proves that a node is investing memory or storage resources to store a target file or file fragment. By building a POR-based SOP into Bitcoin, our scheme creates a system of *highly distributed, peer-to-peer file storage* suitable for storing a large, publicly valuable digital archive **F**. Specifically, our aim is to distribute **F** to protect it against data losses associated with a single entity, e.g., the outages or wholesale data losses already incurred by cloud providers [91].

In contrast to existing peer-to-peer file storage schemes [92, 93], our scheme doesn't require an identity or reputation system to ensure storage of **F**, nor does it require that **F** be a popular download. We achieve file recoverability based strictly on clients' incentives to make money (i.e., mine Bitcoins).

5.1.2 Challenges

In constructing our SOP in Permacoin based on Proofs of Retrievability, we encounter three distinct challenges.

A standard POR involves a single prover holding a single file **F**. In our setting, however, multiple clients collectively store a large dataset **F** (too large for a single client) in a distributed manner. Of these an *adversarially selected fraction* may act maliciously. The first challenge in creating our SOP is to construct an adversarial model for this new setting, and then present a *distributed* POR protocol that is secure in this model. Assuming that clients have independent storage devices, we prove that with our SOP, for clients to achieve a high rate of mining, they must store **F** such that it is recoverable.

Additionally, we must ensure that clients indeed maintain independent storage devices. If, for instance, clients pooled their storage in the cloud to reduce their resource investment, the benefits of dataset-recovery robustness through distribution would be lost. Thus a second challenge in our SOP construction is to ensure that clients must make use of *local* storage to solve it.

To enforce locality of storage, we take two approaches. First, we base our SOP construction on the nonoutsourceable puzzle construction presented in Chapter 3. Second, we ensure that accesses to the file are made sequentially and pseudorandomly. Thus fetching blocks remotely from a provider would incur infeasibly high communication costs (e.g., extremely high latency). We show using benchmarks how our SOP scheme thus takes advantage of practical network-resource limitations to prevent dangerous storage pooling.

5.2 The Permacoin Scratch-Off Puzzle

Our idea, at a high level, is to build a scratch-off-puzzle out of a Proof-of-Retrievability (POR), such that the only way effective way to solve the puzzle is to store portions of the public dataset. In the following sections, we describe how we design the puzzle to ensure that **(a)** users reliably store a subset of the data, **(b)** participants assign themselves mostly non-overlapping subsets of data to ensure good diversity, and **(c)** the entire dataset is recoverable with high probability from the contents of participants' local storage devices.

5.2.1 Strawman: A Simple POR-based Puzzle

To reduce the energy wasted by Bitcoin's current proof-of-computation lottery, we propose replacing it in Permacoin with a POR lottery. In a POR lottery, every scratch-off attempt can be associated with the effort of computing a POR. There are at least two issues that must be addressed:

- *Choosing a random subset of segments based on each participant's public key.* Since each participant may not have sufficient storage to store the entire dataset, we have each participant choose a random subset of segments of the data to store, based on the hash of their public key.
- *Non-interactive challenge generation.* In traditional PORs, a verifier sends a random challenge to a prover, and the prover answers the challenge. In our system, the verifier is the entire Bitcoin network, and the challenge must be generated non-interactively.

  Thus, we have the participants generate challenges based on the publicly known epoch-dependent puzzle ID puz. A valid challenge is computed as $H(\mathsf{puz}||s)$ for some string s of the prover's choice.

Our strawman protocol is described in Figure 5.1.

- $\mathcal{G}(1^\lambda)$: **Setup.**
  The dealer computes and publishes the digest of the entire dataset $\mathbf{F}$, consisting of n segments.

- $\mathsf{Work}(\mathsf{puz}, m)$:

  Generate an arbitrary identity (e.g., a signing key or Bitcoin address) with public key pk.

  Next choose a subset S_{pk} of segments to store:

  $$\forall i \in [\ell]: \text{ let } \mathbf{u}[i] := H_0(\mathsf{pk}||i) \mod n, \qquad S_{\mathsf{pk}} := \{\mathbf{u}[i]\}_{i\in[\ell]}$$

  where ℓ is the number of segments stored by each participant. The participant stores $\{(\mathbf{F}[j], \pi_j) | j \in S_{\mathsf{pk}}\}$, where π_j is the Merkle proof for the corresponding segment $\mathbf{F}[j]$.

  Scratch. Repeat the following scratch procedure: Every scratch-off attempt is seeded by a random string s chosen by the user. A participant computes k random challenges from its stored subset S_{pk}:

  $$\forall i = 1, 2, \ldots, k: \quad r_i := \mathbf{u}[H(\mathsf{puz}||\mathsf{pk}||i||s) \mod \ell] \tag{5.1}$$

  The ticket is defined as:

  $$\mathsf{ticket} := (\mathsf{pk}, s, \{\mathbf{F}[r_i], \pi_i\}_{i=1,2,\ldots,k})$$

  where π_i is the Merkle proof for the r_i-th segment $\mathbf{F}[r_i]$.

- **Verify.** The Verifier is assumed to hold the digest of F. Given a $\mathsf{ticket} := (\mathsf{pk}, s, \{\mathbf{F}[r_i], \pi_{r_i}\}_{i=1,2,\ldots,k})$ verification first computes the challenged indices using Equation (5.1), based on pk and s, and computing elements of $\mathbf{u}$ as necessary. Then verify that all challenged segments carry a valid Merkle proof.

FIGURE 5.1: A simple POR lottery.

5.2.2 A Nonoutsourceable POR-based Puzzle

One drawback of the strawman POR lottery (Figure 5.1) is that it is outsourceable, so it does not incentivize *distributed* storage, which undermines our goal of long-term, resilient data-storage.

In particular, participants can potentially benefit from economies of scale if they outsource the puzzle solving process to a cloud server, including the storage and computation necessary. If most users outsource the puzzle solving process to the cloud, then Permacoin's distributed computational and storage network would effectively become centralized with a few companies. To increase our resilience against correlated disasters, we wish to increase the geographical diversity of the storage. Therefore, we wish to disincentivize users from outsourcing their storage to cloud providers.

We now propose a nonoutsourceable POR lottery mechanism (see Figure 5.2) that discourages users from outsourcing puzzle solving to the cloud.

Idea 1: Tie the payment private key to the puzzle solution. Our first idea is to tie to the puzzle solution to the private key to which the lottery reward is paid out. This key must be kept private in order to claim the reward for oneself. By tying the payment private key to the puzzle solution, a user must reveal her private key to the cloud if she wishes to reduce her own

- $\mathcal{G}(1^\lambda)$: **Setup.** Let $r > 1$ denote a constant. Suppose that the original dataset $\mathbf{F}_0$ contains f segments.

  First, apply a maximum-distance-separable code and encode the dataset $\mathbf{F}_0$ into $\mathbf{F}$ containing $n = rf$ segments, such that any f segments of F suffice to reconstruct $\mathbf{F}_0$. Then, proceed with the **Setup** algorithm of Figure 5.1.

- $\mathsf{Work}(\mathsf{puz}, m)$:

  Construct Merkle tree. Sample L random strings $\mathsf{leaf}_1, \ldots, \mathsf{leaf}_L \xleftarrow{\$} \{0,1\}^\lambda$, and then construct a Merkle tree over the leaf_i's using $\mathcal{H}_2$ as the hash function. Let pk denote the root digest of this tree.

  Next choose a subset S_{pk} of segments to store:

  $$\forall i \in [\ell] : \text{ let } \mathbf{u}[i] := H_0(\mathsf{pk}||i) \mod n, \qquad S_{\mathsf{pk}} := \{\mathbf{u}[i]\}_{i \in [\ell]}$$

  where ℓ is the number of segments stored by each participant. The participant stores $\{(\mathbf{F}[j], \pi_j) | j \in S_{\mathsf{pk}}\}$, where π_j is the Merkle proof for the corresponding segment $\mathbf{F}[j]$.

  Scratch. For each scratch-off attempt seeded by a random none s chosen by the user, compute the following:

  $\sigma_0 := 0$
  $r_1 := \mathbf{u}[\mathcal{H}(\mathsf{puz}||\mathsf{pk}||s) \mod \ell]$
  For $i = 1, 2, \ldots, k$:
  $\quad h_i = \mathcal{H}(\mathsf{puz}||\mathsf{pk}||\sigma_{i-1}||\mathbf{F}[r_i]) \quad (*)$
  $\quad$ Use the value h_i to pseudorandomly select q distinct leaves from $\{\mathsf{leaf}_i\}$.
  $\quad$ Let $B_1, B_2, \ldots, B_q$ denote the Merkle branches corresponding to the selected leaves.
  $\quad \sigma_i := \{B_i\}_{i \in [q]}$ in sorted order of i.
  $\quad r_{i+1} := \mathcal{H}(\mathsf{puz}||\mathsf{pk}||\sigma_i) \mod \ell$
  This attempt succeeds if $\mathcal{H}(\mathsf{puz}\|r\|\sigma_k) < 2^{\lambda - d}$

  Sign payload. Sign the payload m as follows:

  * Compute $h' := \mathcal{H}(\mathsf{puz}\|m\|\mathsf{digest})$, and use the value h' to select a set of $4q'$ distinct leaves from $\{\mathsf{leaf}_i\}$ such that these leaves are not contained in $\{\sigma_i\}$. From these, choose an arbitrary subset of q' distinct leaves. Collect the corresponding branches for these q' leaves, denoted $B_1, B_2, \ldots, B_{q'}$.
  * Let $\sigma'_h := \{B_i\}_{i \in [q']}$ in sorted order of i.
  * The ticket is defined as:

  $$\mathsf{ticket} := (\mathsf{pk}, s, \sigma'_h, \{\mathbf{F}[r_i], \sigma_i, \pi_{r_i}\}_{\forall i=1,2,\ldots,k},)$$

  where π_{r_i} is the Merkle proof of $\mathbf{F}[r_i]$.

- **Verify.** Given $\mathsf{ticket} := (\mathsf{pk}, s, \sigma'_h \{\mathbf{F}[r_i], \sigma_i, \pi_{r_i}\}_{\forall i=1,2,\ldots,k})$, verification is essentially a replay of the scratch-off, where the signing is replaced with signature verification. This way, a verifier can check whether all iterations of the scratch-off were performed correctly.

FIGURE 5.2: Nonoutsourceable POR-based puzzle.

costs by outsourcing the puzzle to the cloud. We assume that at least a fraction of the users will choose not to entrust the cloud with their payment private keys.

Idea 2: Sequential and random storage access. We also need to discourage a user from outsourcing storage to the cloud, but performing computation locally on her own machine. To achieve this goal, we craft our puzzle such that access to storage is sequentialized during the scratch-off attempt. Furthermore, the storage access pattern is random (based on outcomes of calling a random oracle) and cannot be precomputed ahead of time. Thus, if the data is stored in the cloud and the computation is performed locally, many round-trips must be incurred during the scratch-off attempt, which will reduce the user's chance of finding a winning ticket.

Idea 3: Boosting recoverability with erasure codes. As in standard proof-of-retrievability schemes, we boost the probability of successful recovery through erasure coding. In the setup phase, we erasure code a dataset containing f segments into rf segments, where $r > 1$, such that any f segments suffice to recover the dataset.

Parameterizations and security. In order for this to be secure, unforgeable signature scheme for all $k+1$ messages, we can set $L = 2kq + 8q'$, $q = O(\lambda)$ and $q' = O(\lambda)$, as in Section 4.2.

In practice, we need only set our parameters such that any reasonable user would prefer to store at least $L/2$ leaves on the server. We can therefore set $q = 1$ for all the internal iterations of the scratch-off step. However, for the $(k+1)$-th signature, we set $q' = O(\lambda)$, and the solver must choose $4q'$ leaves and reveal any q' of them. In this case, if the client withholds $L/2$ leaves from the server, the server must in expectation contact the client $k/2$ times during the scratch-off attempt – in Section 5.3, we show that the cost of transmitting even small packets of data greatly exceeds the cost of simply computing scratch-off iterations locally. Therefore, a rational user would not outsource its computation yet withhold $L/2$ or more leaves.

5.3 To Outsource or Not to Outsource

As mentioned earlier, since we bind possession of newly minted coins to a user's private key in Permacoin, we assume that a substantial fraction of users will *not* entrust their private keys to a service provider and risk theft of their coins.

A user j who only stores her private key sk_j locally can choose between two ways of storing her assigned blocks of $\mathbf{F}$: a *local storage* device or *outsourced storage* leased from a remote cloud storage service. (A combination of the two is also possible.) We now analyze the storage choice of rational participants, those seeking to maximize their return on mining by achieving the lowest expected cost per SOP. We argue that rational users will choose *local storage* to drive down their resource costs.

In both the local storage and outsourced storage scenarios, the user locally provisions a basic computational resource (incurring the hardware costs of a motherboard, CPU, and RAM and

TABLE 5.1: Notation used for Permacoin system parameters

f	# segments necessary for recovery
m	total # segments stored by good users during recovery
n	total # encoded segments
ℓ	# segments assigned to each identity
k	# iterations per puzzle
B	# size of each block (bytes)

power costs, but not of a substantial storage medium). The cost differences for the two storage scenarios—again, favoring local storage—stem from the following:

Cost of Storage and I/O: In the local scenario, a client's costs are its investment in storage equipment for mining, specifically, for the purchase of RAM or SSD. (These costs may be characterized in terms of equipment depreciation.)

In the outsourced scenario, a client's costs include the: 1) Cost of storage and disk I/O charged by the service provider; 2) Cost of network bandwidth, including that of the network link provided by an ISP, and the cost per GB of network transfer charged by a service provider. In our setting, storage of the file F can be amortized across multiple users, so we assume the storage cost and disk I/O cost are close to zero. What remains is the cost of network I/O.

We show that based on typical market prices today, the costs of storage and I/O are significantly cheaper for the local storage option.

Latency: By design, our SOP sequentially accesses blocks in $\mathbf{F}$ in a random (pseudorandom) order. The resulting, unpredictable fetches penalize outsourced storage, as they introduce substantial latency: a single round-trip for every fetched block, which is vastly larger than disk I/O latency. This latency overhead reduces a miner's chance of finding a valid puzzle solution and winning the reward when the number k of outsourced fetches is large.

If each block fetch incurs roundtrip latency τ, then for large k, the total incurred latency $k\tau$ may be quite large. For example, with $k = 6000$, one iteration parameter we analyze below, and a 45ms roundtrip latency, typical for regional internet accesses, $k\tau$ would be 4.5 minutes—almost half the length of an epoch. Boosting to $k > 13,333$ would render $k\tau$ *larger* than an average epoch, making outsourcing infeasible.

Of course, if $k\tau$ is small enough, a client can parallelize fetches across SOP guesses. It is helpful to quantify formally the value of time, and penalties associated with latency, as we do now.

5.3.1 Stochastic model

We now present a stochastic model that offers a quantitative comparison of the economics of local vs. outsourced storage. The notation used for the parameters of our scheme are summarized in Table 5.1.

We consider a stochastic process in which a single-threaded mining process is trying to find a ticket. This mining thread will keep computing the iterations sequentially as described in Figure

5.2. At any time, if another user in the network finds a winning ticket first, the current epoch ends, and the mining thread aborts the current scratch-off attempt and starts a new attempt for the new epoch.

We consider the following cost metric: *expected cost invested until a user succeeds in finding one ticket.* Every time a user finds a ticket (before anyone else finds a winning ticket), the user has a certain probability of having found a winning ticket, and hence being rewarded.

Game with a giant. We can think of this stochastic process as a user playing a game against a *giant.* The giant models the rest of the network, which produces winning tickets at a certain rate. The stochastic process in which the giant produces winning tickets is a *memoryless* process. At any time, the remaining time T it takes for the giant to find a winning ticket follows an exponential distribution. The expectation of T is also the *expected epoch length.* In Bitcoin, as noted, the difficulty of its SOP is periodically adjusted with respect to the computational power of the network to keep the expected epoch length at about 10 minutes.

If the giant generates a puzzle solution, it is immediately communicated to the user, who aborts her current attempt. Thus the stochastic process can be modeled as a Markov chain as follows:

- Every iteration takes t time, and costs c.
- If k iterations are finished (before the giant wins), a user finds a ticket (which may or may not be a winning ticket). In this case the user gets a positive reward in expectation.
- Let s_i denote the state in which the user has finished computing the i-th iteration of the puzzle.
- If $i < k-1$: with probability p, the giant does not win in the current iteration, and the state goes from s_i to s_{i+1}. With probability $1-p$, the giant wins, and the state goes back to s_0, i.e., the current epoch ends, and a new scratch-off attempt is started. Suppose that the expected epoch length is T; then it is not hard to see that $p = 1 - t/T$ given that the stochastic process of the giant winning is memoryless.
- In state s_{k-1}, with probability 1, the state goes back to s_0. Furthermore, in state s_{k-1}, with probability p, the user will finish computing all k iterations — in which case another random coin is drawn to decide if the ticket wins.

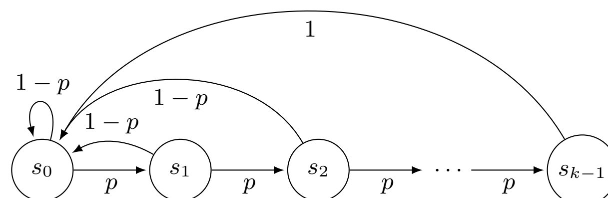


FIGURE 5.3: Markov chain model for a sequential mining process that resets if the epoch ends during an iteration.

We analyze the stationary distribution of the above Markov chain. Let π_{k-1} denote the probability that it is in state s_{k-1}. It is not hard to derive that $\pi_{k-1} = (1-p)p^{k-1}/(1-p^k)$. Therefore,

in expectation, $1/\pi_{k-1}$ time is spent between two visits to the state s_{k-1}. Every time the state s_{k-1} is visited, there is a p probability that the user will finish all k iterations of the puzzle. Therefore, in expectation, $1/(\pi_{k-1}p)$ time (in terms of number of iterations) is spent before the user finds a ticket before the giant does. If a user finds a ticket before the giant does, we call this a "success". Hence, we have that

$$\mathcal{E}[\text{expected cost per success}] = \frac{c(1-p^k)}{(1-p)p^k}$$

5.3.2 Local Storage vs. Outsourced Storage

Based on the above analysis, we now plug in typical practical values for the parameters and investigate the economics of local vs. outsourced storage.

Local storage. The cost of a scratch-off attempt depends on two things, the power consumed and the cost of the equipment. We consider two hardware configurations,

1. with SSD drives as the storage medium; and
2. using RAM as the storage medium.

Both are typical configurations that an amateur user can easily set up. Note that while it is possible to optimize the local hardware configuration further to have better amortized cost, it is outside the scope of this paper to do so, since our goal is to show that, even for an amateur user, local mining is economically superior to outsourced storage mining.

First we estimate the cost of local mining using an SSD and standard CPU. Today, the cost of a desktop containing a high-end processor (Intel Core i7, 3.4GHz and 8 virtual cores) is approximately \$500. The cost of a 100GB SSD is about \$100. Amortized over three years, the effective cost is 6.34*e*-6 \$/second. We measured the power consumption while mining to be about 40 watts; assuming an electricity cost of 15 cents/kWh, the energy cost of mining is 1.67*e*-6 \$/second in power. Note the mining cost is dominated by equipment, not power. The latency for a single disk read of up to 4096 bytes is measured at approximately 30 microseconds.

We assume for now that the size of a file segment is 64 bytes, and every puzzle iteration requires hashing a single leaf with two 120-bit secrets ($y = 1$). Computing a hash over a message of less than 128 bytes takes no more than $\sim$ 15 microseconds on an ordinary CPU, suggesting that for a single-threaded mining program, the SSD and CPU would be in approximately equal utilization. Thus assuming an average of 30 microseconds per iteration, the cost of mining with a local SSD is roughly 3.2*e*-10 \$/iter.

Next we consider the cost of local mining using RAM rather than an SSD. A 2GB stick of DDR3 SDRAM can be purchased for about \$20, and has a data transfer rate of 12,800 megabytes per second. Assuming a segment size of 64 bytes, the average throughput of this memory is approximately 200 million puzzle iterations per second. This is faster than a single-threaded CPU performing signing operations can keep up with. On the other hand, many desktop computers

have a graphics processor (GPU) that can be used to accelerate Bitcoin mining. Taking one example, the ATI Radeon 6750 costs $100, consumes 150 watts, and can perform 150 million Bitcoin hashes per second. Thus, under this scheme the GPU would be utilized approximately as much as the RAM.

Outsourced storage. The cost of outsourced storage mining may vary according to the pricing of the specific service provider. Our goal is to show that under most conceivable scenarios for outsourced mining, local mining will be superior. To demonstrate this, we consider a wide spectrum of cost ranges for the outsourced storage setting, and show that even when we unfairly favor the outsourced option by assuming aggressive lower bounds for its cost, the local option is still more more economical.

We consider multiple cost configurations for the outsourced storage option:

1. *EC2.* First, we rely on the pricing representative of today's high-end cloud providers. In particular, our estimates are based of Amazon EC2's pricing. EC2 charges 10 cents per gigabyte of transfer, and a base rate of 10 cents for the smallest virtual machine instance.

2. *Bandwidth + CPU.* Amazon EC2's setup is not optimized for constant-use high-bandwidth applications. Other rental services (such as http://1gb.com/en/) offer "unmetered" bandwidth at a fixed monthly cost. To model this, we consider a cost lower bound by assuming that the cloud provider charges nothing, and that the user only needs to pay for its local CPU and the bandwidth cost charged by the ISP.

   Internet transit costs are measured in $ per mbps, per month. Costs have diminished every year; the median monthly cost of bulk bandwidth during 2013 has been estimated at $\$1.71/mbps$, corresponding to 0.53 cents per gigabyte under constant use.[3] Each puzzle iteration requires transferring a file segment.

   Since the SSD accounts for about 16% of the equipment cost in the local SSD configuration, and the CPU is approximately matched with the SSD in terms of utilization, for this model we assume that the latency is equivalent, but reduce the local equipment and power cost by 16%.

3. *CPU only or bandwidth only.* We consider an even more aggressive lower bound for outsourcing costs. In particular, we consider a scenario in which the user only needs to pay for the local CPU; or she only needs to pay the ISP for the bandwidth.

   While this is not realistic today, this lower bound models a hypothetical future world where cloud costs are significantly lowered, or the scenario where a powerful adversary can reimburse users' mining costs assuming they join its coalition.

[3] According to an October 2013 press release by market research firm TeleGeography: http://www.telegeography.com/press/press-releases/2013/10/08/ip-transit-port-upgrades-yield-steeper-price-declines-for-buyers/index.html

TABLE 5.2: Costs per iteration for different mining configurations in Permacoin (mining with local storage vs. three forms of cloud storage). Latency is the time to compute one iteration of the puzzle. Effective Latency accounts for the pipelining of computation and storage requests. Equipment is the fixed cost of the system. Total cost per iteration is shown assuming the transfer of a 64-byte segment.

Model	Latency	Eff. Lat.	Equipment	Power	Bandwidth	Total
CPU & SSD	45μs	30μs	$600	40W	n/a	2.10e$-10/iter
GPU & RAM	600ns	300ns	$700	190W	n/a	5.04e$-14/iter
EC2	30ms	0	$0.10/$s$	n/a	$.10/GB	8.39e$-7/iter
CPU + BW	30ms	15μs	$500	33.6W	5.3e$-3/GB	4.04e$-10/iter
CPU Only	30ms	15μs	$500	33.6W	n/a	8.76e$-11/iter
BW Only	30ms	n/a	n/a	n/a	5.33e$-3/GB	$3.16-10/iter

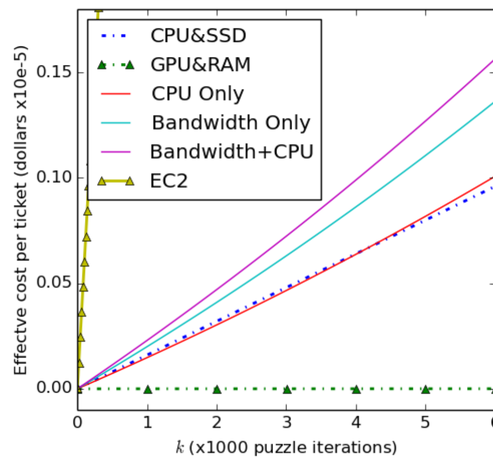


FIGURE 5.4: Cost effectiveness versus number of iterations k, for different hardware configurations. Note that for $k > 4e3$ iterations, the CPU/SSD configuration with local storage is more cost effective than the CPU-only (zero-cost bandwidth) with remote storage.

Findings. Table 5.2 compares the costs of local mining to those of outsourced storage.

Notice that in our protocol in Figure 5.2 one tunable parameter is the number of bytes that must be transferred between the server and the client per iteration if storage were to be outsourced to a server. In general, when more bytes are transferred per iteration, the bandwidth cost per iteration also increases. In Table 5.2 we assume a conservative parameter setting where only 64-byte segments are transferred.

Although latency is listed in the second-leftmost column, the effect of latency is not accounted for in the rightmost Total cost column, since this depends on the number of iterations of the puzzle. Figure 5.4 illustrates that cost effectiveness diminishes when the number of iterations is increased sufficiently. The figure suggests that under almost all scenarios, local mining is strictly more economical than outsourcing storage, regardless of the number of iterations k for the scratch-off attempt. We stress that this is true even when 1) the local mining user did not spend too much effort at optimizing its hardware configuration; and 2) we give the outsourced storage option an unfair advantage by using an aggressive lower bound for its costs. Recall that local mining saves in cost for two reasons: 1) local storage and I/O costs less than remote (in the latter case the client has to pay for both the storage, disk I/O, and network bandwidth); and 2) lower storage I/O latency gives the user an advantage in the stochastic lottery against the "giant".

The only exception is the "CPU only" entry in Table 5.2 — in this case, the user is not paying anything for bandwidth, and the only cost is for the CPU hashing operation. In this case, the cost per iteration is lower for the outsourced option than for the local CPU/SSD option (though even here GPU/RAM with local storage remains more efficient). However, longer roundtrip latency to the remote storage will penalize the user during the mining. Therefore, even in this case, we could discourage outsourced storage by setting k very large (thousands of iterations), so that the effect of longer storage I/O latency dominates. For the rest of our analysis, we do include the price of bandwidth in our model and so small values of k are sufficient.

5.4 Partially Outsourced Storage Analysis

While our analysis in Section 5.3 shows that a puzzle-solving configuration using local storage is typically more cost-effective than a configuration using remote cloud storage, we also wish to consider a hybrid-strategy, in which the user chooses to store just a fraction $\gamma < 1$ of her assigned segments on the cloud. We suppose that by doing so, the user may be able to recover an γ fraction of the cost of her storage device. However, this results in a diminished puzzle-solving rate, since with probability γ, a given iteration requires accessing one of the remotely stored segments. The user has two options in this case; the first is to incur the cost of bandwidth by fetching the block from the cloud, while the second is to simply abort the attempt and start a fresh attempt from the beginning. Let C_1 (resp., C_2) denote the cost of fetching one block from the cloud (resp., local device), and p_1 (resp., p_2) denote the probability of the epoch ending during the fetch from the cloud (resp., local device), and let c_i be the expected cost of reaching the next ticket beginning from state i (using the optimal strategy). It is preferable to abort after state s_i (when the next segment to fetch is remote) if $c_0 < C_2 + p_2 c_{i+1} + (1 - p_2) c_0$. We can simplify the strategy space by observing that c_i is monotonically decreasing, thus the optimal strategy is defined by a critical state s_μ, after which it is preferable to fetch from the cloud, and before which it is preferable to restart. The stochastic process describing this hybrid strategy is illustrated in Figure 5.5.

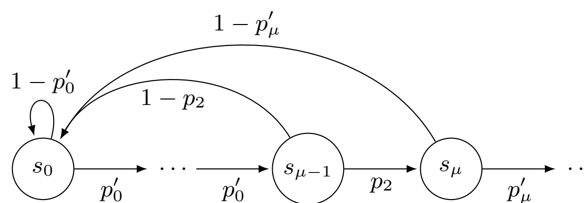


FIGURE 5.5: Markov chain describing the optimal hybrid strategy. State s_i represents the instant before fetching the ith segment (either locally or remotely). There is a critical state μ, before which it is advantageous to restart from s_0 rather than incur the cost of fetching from the cloud. We abbreviate the transition probabilities in the states before s_μ as $p'_0 = (1-\gamma)p'_2$, and after s_μ as $p'_\mu = \gamma p_2 + (1-\gamma)p_1$.

The optimal value for μ can be computed iteratively by plugging in appropriate values for C_1, C_2, p_1, p_2. We illustrate the expected cost per ticket (using the optimal hybrid strategy) for various settings of the puzzle iterations parameter k (and leaving the segment size as 64 bytes)

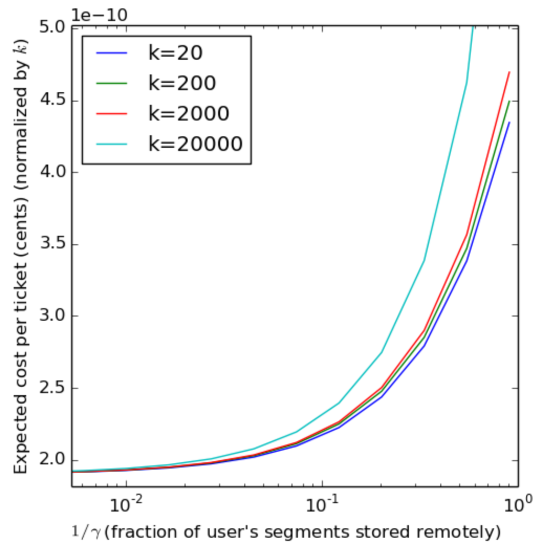


FIGURE 5.6: Cost effectiveness of the optimal hybrid strategy, when a portion γ of the file is not stored locally.

in Figure 5.6. Given the relative cost of bandwidth to the cost of a local storage device, this analysis shows that is preferable to store the entire file ($\gamma = 0$) for any setting of k.

Based on the above analysis, we can make the following claim about the behavior of rational mining participants:

Claim 1. For realistic estimates of bandwidth and equipment costs (see Table 5.2), even assuming participants that omit segments can recoup the proportional cost of the local storage devices, rational participants will choose to locally store the entire set of file segments associated with their public key, rather than omitting any segments or relying on remote cloud storage.

Proof. This claim follows from the optimal hybrid strategy (illustrated in Figure 5.5) when some segments are not stored locally, and plugging in our estimated bandwidth and equipment costs (see Table 5.2 and Figure 5.6). The claim holds whenever the cost of bandwidth is high relative to the cost of a storage device, or when the cost of the signature-computing device is a significant fraction of the overall equipment cost. □

Chapter 5

The Blockchain Model of Computation

6.1 Overview

Although Bitcoin is primarily intended as a payment system, in Chapter 2, we have shown that Bitcoin's underlying abstraction, the Nakamoto consensus protocol, can be used as a mechanism for reaching agreement about arbitrary data values. It is well known in the literature that such consensus protocols can be composed to implement other general purpose abstractions, such as replicated state machines, and therefore a wide range of applications [94]. In fact, Bitcoin already includes a small scripting language for "programming money" with access control policies. Bitcoin's recent competitor, Ethereum [28] (also built from a variation of Nakamoto consensus) provides a general purpose and expressive programming interface for user-provided applications, called "contracts."[1]

A key challenge in defining applications is that the programs in this setting are inherently "transparent," since the underlying consensus algorithm reveals all of the information to the public. If an application is to provide privacy, it must rely on an additional layer of cryptographic protocols.

In this chapter, we define a formal model of blockchain programs. Our framework serves two main uses:

- It can be used for specifying "transparent contracts," such as those implemented directly in Bitcoin and Ethereum.
- Our framework can also be used for formally specifying and constructing privacy-preserving applications.

[1] We use the terms "contracts" and "smart contracts" interchangeably to refer to user-provided programs running on a blockchain.

Our main result in this chapter is a proof that the Nakamoto consensus protocol (our abstraction of the Bitcoin protocol, as defined in Chapter 2), can be used to implement any transparent contract. This chapter therefore serves as a bridge between the Nakamoto consensus protocol defined in Chapter 2 and our development of privacy-preserving blockchain applications in Chapter 6.

Our approach. We develop a simplified abstraction of the functionality that existing cryptocurrency systems provide, eliding many practical details but capturing most salient aspects. Our model includes pseudonymous interactions between users, a framework for defining "time-aware" applications, and the ability of attackers to mount "front-running" attacks. We model the cryptocurrency system as a functionality, i.e. as a trusted third party service.

Assuming that the decentralized consensus protocol is secure, smart contracts can be thought of as a conceptual party (in reality decentralized) that can be *trusted for correctness and availability but not for privacy.* Our model makes use of a mechanism called a *transparent contract wrapper.* This wrapper is parameterized by an arbitrary program (i.e., called a "transparent contract") and runs the programs such that all of its internal state is made publicly visible, both to the adversary and to honest parties. During a protocol, users interact with the contract by exchanging messages (also referred to as transactions). Money can expressed as special data stored on the blockchain, which is interpreted as a financial ledger. Our contracts can access and update the ledger to implement money transfers between users (as represented by their pseudonymous public keys).

Our model allow us to easily capture the time and pseudonym features of cryptocurrencies. In cryptocurrencies such as Bitcoin and Ethereum, time progresses in block intervals, and a smart contract program is allowed to query the current time, and make decisions accordingly, e.g., make a refund operation after a timeout. In addition, our model captures the role of a smart contract as a party trusted for correctness but not for privacy. Lastly, our formalism modularizes our notations by factoring out common specifics related to the smart contract execution model, and implementing these in central wrappers.

In a real-life cryptocurrency system such as Bitcoin or Ethereum, users can make up any number of identities by generating new public keys. In our formal model, for simplicity, we assume that there can be any number of identities in the system, but that they are fixed a priori. It is easy to extend our model to capture registration of new identities dynamically. As mentioned later, we allow each identity to generate an arbitrary (polynomial) number of pseudonyms.

6.2 Related Work

To keep our modeling simple, and in order to support our highly generalized view, we choose to analyze a simplified abstraction of the Bitcoin protocol.

Other modeling efforts have focused instead on modeling the specific Bitcoin protocol itself. In particular, a significant difference between our model protocol and Bitcoin is that our protocol proceeds in well defined epochs, such that transactions are submitted at the beginning of an

epoch and committed at the end. In reality, the Bitcoin protocol continuously processes new transactions, in a pipelined fashion, and only reaches agreement in a stabilizing (rather than a final) sense. Garay et al [21], as well as Pass and shelat [20] analyze more faithful models of the Bitcoin protocol that capture this aspect. Even these protocols make significant simplifications of the actual Bitcoin protocol. For example, the actual Bitcoin protocol features dynamic adjustment of the puzzle difficulty, and therefore tolerates organically varying numbers of participants.

Its features combine to make the blockchain a new computation model capable of enforcing notions of *financial fairness* even in the presence of aborts. As is well-known in the cryptography literature, fairness against aborts is in general impossible in standard models of interactive protocol execution (e.g., secure multi-party computation), when the majority of parties can be corrupted [40, 95]. In the blockchain model, protocol aborts can be made evident by *timeouts*. Therefore, the blockchain can enforce financial remuneration to the honest counterparties in the presence of protocol breach or aborts. Bentov et al. [22, 96, 97] formalized several financially fair protocols using a "claim-or-refund" gadget, a simple primitive that can be implemented even using Bitcoin's scripting language. Compared to their model, our transparent contracts are significantly more flexible and convenient to write.

Our most closely resembles the functionality provided by Ethereum, a so-called "Turing-complete" cryptocurrency featuring a general purpose smart-contract programming language [28]. Ethereum also features many other practical design innovations (e.g., a gas mechanism that prevents resource exhaustion attacks) that our high-level abstraction does not model. Although Ethereum provides a formal definition of (a portion of) its protocol, it does not come with any specification of its security guarantees. We address this shortcoming with our $\mathcal{F}_{\text{BLOCKCHAIN}}$ framework, which not only serves as a formal specification for the guarantees by our transparent contract platform, but further provides a way to specify and construct secure and privacy-preserving applications built on top.

6.3 Programs, Functionalities, and Wrappers

To make our notation simple for writing ideal functionalities and smart contracts, we make a conscious notational choice of introducing *wrapper* functionalities. Wrapper functionalities implement in a central place a set of common features (e.g., timer, ledger, pseudonyms) that are applicable to all ideal functionalities and contracts in our smart contract model of execution. In this way, we can modularize our notational system such that these common and tedious details need not be repeated in writing ideal functionalities and contract programs.

Contract functionality wrapper $\mathcal{F}_{\text{BLOCKCHAIN}}$. A contract functionality wrapper $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{C})$ takes in a *contract program*, denoted C, and produces a contract functionality. The contract program might either be *transparent* or *privacy-preserving*, as we'll describe shortly. The functionality wrapper is formally defined in Figure 6.1.

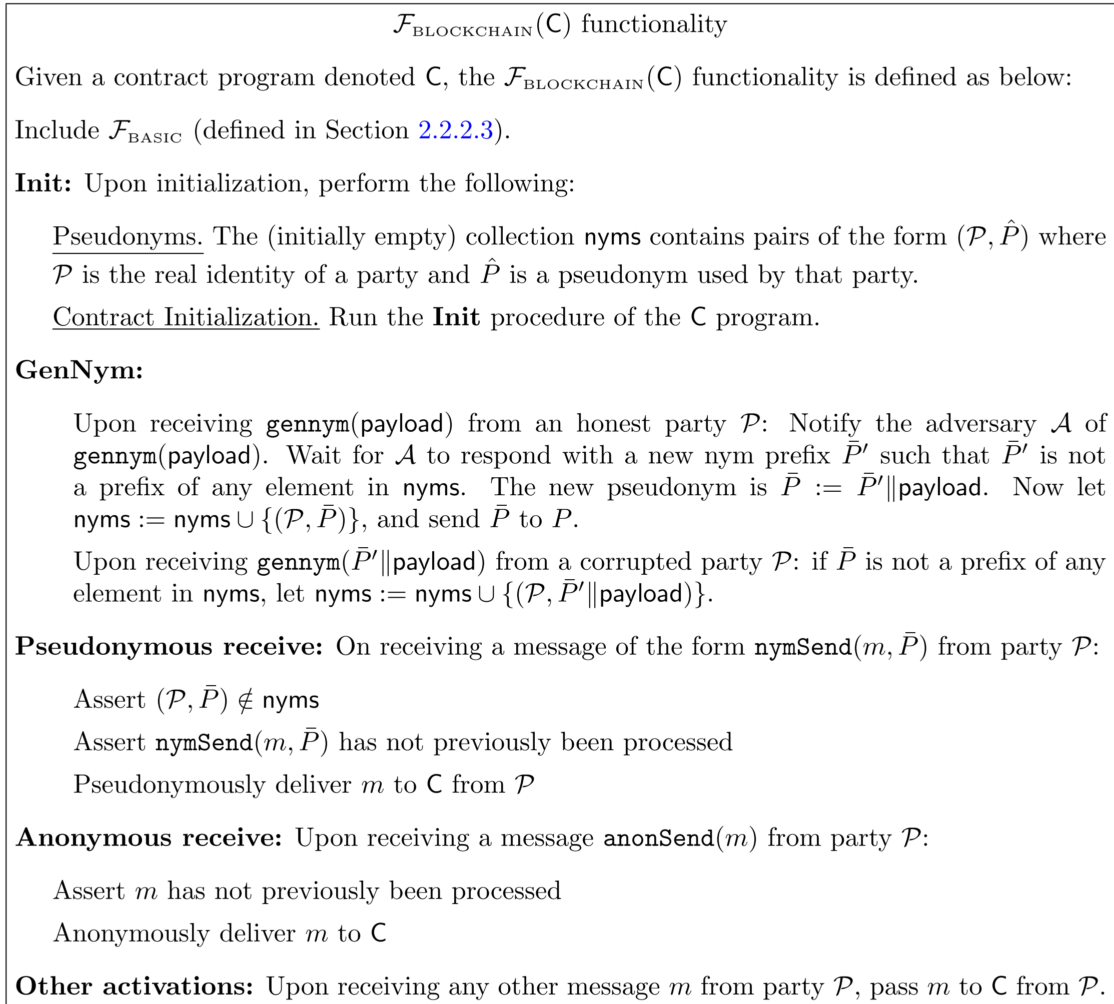
The contract functionality wrapper allows for arbitrary synchronous (i.e., round-based and time-aware) interactions between the contract program and the users. It also features a model of pseudonyms, where users can create an arbitrary number of pseudonyms to interact with the contract program.

In more detail, the functionality models the following features:

- *Time, and batched processing of messages.* In popular decentralized cryptocurrencies such as Bitcoin and Ethereum, time progresses in block intervals marked by the creation of each new block. Intuitively, our $\mathcal{F}_{\text{BLOCKCHAIN}}(\cdot)$ wrapper captures the following fact. In each round (i.e., block interval), the smart contract program may receive multiple messages (a.k.a. transactions). The order of processing these transactions is determined by the miner who mines the next block. In our model, we allow the adversary to specify an order of the messages collected in a round, and our contract program will then process the messages in this adversary-specified ordering.
- *Rushing adversary.* The contract wrapper $\mathcal{F}_{\text{BLOCKCHAIN}}(\cdot)$ naturally captures a rushing adversary. Specifically, the adversary can first see all messages sent to the contract by honest parties, and then decide its own messages for this round, as well as an ordering in which the contract should process the messages during the next round. Modeling a rushing adversary is important since it captures a class of well-known front-running attacks, e.g., those that exploit transaction malleability [36, 98]. For example, in a "rock, paper, scissors" game, if inputs are sent in the clear, an adversary can decide its input based on the other party's input. An adversary can also try to modify transactions submitted by honest parties to potentially redirect payments to itself. Since our model captures a rushing adversary, we can write ideal functionalities that preclude such front-running attacks.

Transparent Contract Wrapper, $\mathcal{T}(\cdot)$ Transparent contracts are contract programs that can be *trusted for correctness but not for privacy*. More specifically, they have no private internal state, immediately leak their inputs/outputs to the adversary, and furthermore allow the adversary to front-run messages sent from users within a round. We model this notion formally by defining a *transparent contract wrapper*, denoted by $\mathcal{T}(\cdot)$, that implements this behavior in a general way: $\mathcal{T}(\mathsf{Contract})$ represents a transparent contract for any contract program $\mathsf{Contract}$. The transparent contract wrapper is defined in Figure 6.2. The main technical result of this section is to show that the Nakamoto consensus protocol can be used to implement any transparent contract defined by this wrapper. That is, we construct a protocol that UC-realizes $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{Contract}))$ for any transparent contract program $\mathsf{Contract}$.

Blockchain Protocols. In the following chapter, our real world protocols will be defined as protocols in the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\cdot))$-hybrid world. Such a protocol is defined in two parts. The first part is a transparent contract $\mathsf{Contract}$, that specializes the behavior of the functionality. The second part is local code to be run by each of the parties, for example to generate cryptographic messages that are included in transactions posted on the blockchain.



$\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{C})$ functionality

Given a contract program denoted C, the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{C})$ functionality is defined as below:

Include $\mathcal{F}_{\text{BASIC}}$ (defined in Section 2.2.2.3).

Init: Upon initialization, perform the following:

Pseudonyms. The (initially empty) collection nyms contains pairs of the form $(\mathcal{P}, \hat{P})$ where $\mathcal{P}$ is the real identity of a party and $\hat{P}$ is a pseudonym used by that party.

Contract Initialization. Run the **Init** procedure of the C program.

GenNym:

Upon receiving $\texttt{gennym}(\mathsf{payload})$ from an honest party $\mathcal{P}$: Notify the adversary $\mathcal{A}$ of $\texttt{gennym}(\mathsf{payload})$. Wait for $\mathcal{A}$ to respond with a new nym prefix $\bar{P}'$ such that $\bar{P}'$ is not a prefix of any element in nyms. The new pseudonym is $\bar{P} := \bar{P}' \| \mathsf{payload}$. Now let $\mathsf{nyms} := \mathsf{nyms} \cup \{(\mathcal{P}, \bar{P})\}$, and send $\bar{P}$ to P.

Upon receiving $\texttt{gennym}(\bar{P}' \| \mathsf{payload})$ from a corrupted party $\mathcal{P}$: if $\bar{P}$ is not a prefix of any element in nyms, let $\mathsf{nyms} := \mathsf{nyms} \cup \{(\mathcal{P}, \bar{P}' \| \mathsf{payload})\}$.

Pseudonymous receive: On receiving a message of the form $\texttt{nymSend}(m, \bar{P})$ from party $\mathcal{P}$:

Assert $(\mathcal{P}, \bar{P}) \notin \mathsf{nyms}$

Assert $\texttt{nymSend}(m, \bar{P})$ has not previously been processed

Pseudonymously deliver m to C from $\mathcal{P}$

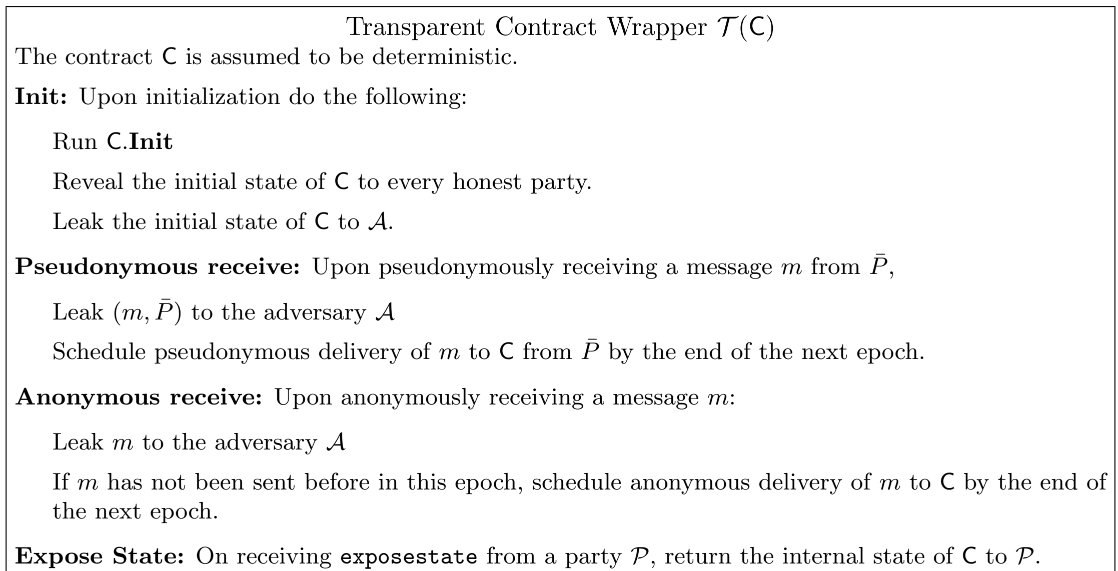
Anonymous receive: Upon receiving a message $\texttt{anonSend}(m)$ from party $\mathcal{P}$:

Assert m has not previously been processed

Anonymously deliver m to C

Other activations: Upon receiving any other message m from party $\mathcal{P}$, pass m to C from $\mathcal{P}$.

FIGURE 6.1: The $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{C})$ functionality is parameterized by a contract program denoted C, and allows pseudonymous interaction with the underlying contract.



Transparent Contract Wrapper $\mathcal{T}(\mathsf{C})$

The contract C is assumed to be deterministic.

Init: Upon initialization do the following:

Run C.**Init**

Reveal the initial state of C to every honest party.

Leak the initial state of C to $\mathcal{A}$.

Pseudonymous receive: Upon pseudonymously receiving a message m from $\bar{P}$,

Leak $(m, \bar{P})$ to the adversary $\mathcal{A}$

Schedule pseudonymous delivery of m to C from $\bar{P}$ by the end of the next epoch.

Anonymous receive: Upon anonymously receiving a message m:

Leak m to the adversary $\mathcal{A}$

If m has not been sent before in this epoch, schedule anonymous delivery of m to C by the end of the next epoch.

Expose State: On receiving $\texttt{exposestate}$ from a party $\mathcal{P}$, return the internal state of C to $\mathcal{P}$.

FIGURE 6.2: Transparent contract wrapper $\mathcal{T}(\cdot)$ exposes all of its internal states and messages received to the adversary, and makes the functionality time-aware (messages received in one epoch and queued and processed by the end of the next epoch), allowing the adversary to determine the exact ordering.

6.3.1 Modeling Pseudonyms

We model a notion of "pseudonymity" that provides a form of privacy, similar to that provided by typical cryptocurrencies such as Bitcoin. Any user can generate an arbitrary (polynomially-bounded) number of pseudonyms, and each pseudonym is "owned" by the party who generated it. The correspondence of pseudonyms to real identities is hidden from the adversary.

Effectively, a pseudonym is a public key for a digital signature scheme, the corresponding private key to which is known by the party who "owns" the pseudonym. The public contract functionality allows parties to publish authenticated messages that are bound to a pseudonym of their choice. Thus each interaction with the public contract is, in general, associated with a pseudonym but not to a user's real identity.

We abstract away the details of pseudonym management by implementing them in our wrappers. This allows user-defined applications to be written very simply, as though using ordinary identities, while enjoying the privacy benefits of pseudonymity.

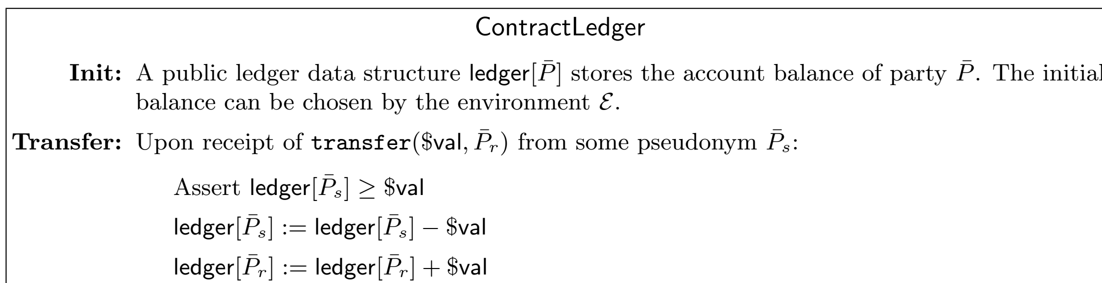
Our wrapper provides a user-defined hook, `gennym`, that is invoked each time a party creates a pseudonym. This allows the application to define an additional per-pseudonym payload, such as application-specific public keys. From the point-of-view of the application, this is simply an initialization subroutine invoked once for each participant.

Our wrapper provides several means for users to communicate with a contract. The most common way is for a user to publish an authenticated message associated with one of their pseudonyms, as described above. Additionally, `anonSend` allows a user to publish a message without reference to any pseudonym at all.

6.3.2 Modeling Money

Rather than building a notion of money into our model, we show how to model money as a transparent contract program, defined in Figure 6.3. We model money as a public ledger, which associates quantities of money to pseudonyms. Users can transfer funds to each other (or among their own pseudonyms) by sending "transfer" messages to the public contract (as with other messages, these are delayed until the next round and may be delivered in any order). The ledger state is public knowledge, and can be queried immediately using the `exposestate` instruction.

There are many conceivable policies for introducing new currency into such a system: for example, Bitcoin "mints" new currency as a reward for each miner who solves a proof-of-work puzzles. We take a simple approach of defining an arbitrary, publicly visible (i.e., common knowledge) initial allocation that associates a quantity of money to each party's real identity. Except for this initial allocation, no money is created or destroyed. This invariant is immediately apparent from the specification!


ContractLedger

Init: A public ledger data structure $\mathsf{ledger}[\bar{P}]$ stores the account balance of party $\bar{P}$. The initial balance can be chosen by the environment $\mathcal{E}$.

Transfer: Upon receipt of `transfer`($\$\mathsf{val}, \bar{P}_r$) from some pseudonym $\bar{P}_s$:

Assert $\mathsf{ledger}[\bar{P}_s] \geq \$\mathsf{val}$

$\mathsf{ledger}[\bar{P}_s] := \mathsf{ledger}[\bar{P}_s] - \$\mathsf{val}$

$\mathsf{ledger}[\bar{P}_r] := \mathsf{ledger}[\bar{P}_r] + \$\mathsf{val}$

FIGURE 6.3: Transparent ledger contract

6.4 Implementing Transparent Blockchain Contracts on top of Nakamoto Consensus

We now prove the main result for this Chapter, which is that the Nakamoto Consensus protocol suffices for implementing any transparent contract. To do this, we build a protocol ProtBlockchain that makes use of the Nakamoto consensus protocol as a primitive.

Our protocol closely resembles the overall behavior of Ethereum, an actual deployed cryptocurrency with a smart contract programming system. In a nutshell, users interact with the contract program by broadcasting messages called "transactions." Participants in the protocol collect and buffer transactions they hear about from others, and submit these as input to the underlying consensus protocol. Our model also includes pseudonymous interactions with the contract, where interactions are authenticated to the owner of a pseudonym. To implement this, we derive pseudonyms from the public keys in a digital signature scheme, and require that pseudonymous input to the contract must come in the form of a message signed with the corresponding private key.

A practical difference between our protocol and real-life cryptocurrency systems like Bitcoin and Ethereum is that our protocol runs the Nakamoto consensus subroutine many times sequentially in order to guarantee that honest transactions within an epoch are included by the end of the next epoch. As a performance optimization, most cryptocurrency systems run in a pipelined fashion.

6.4.1 Blockchain Protocol

The protocol ProtBlockchain is formally defined in Figure 6.4. The consensus protocol proceeds in epochs, where in each epoch we run several instances of the underlying consensus protocol. Participants keep track of a buffer of unconfirmed transactions they have received as input from the environment, and submit these as input to the underlying consensus protocol.

To manage pseudonymous interactions, we derive the pseudonym strings from the public keys in a digital signature scheme, and require that "pseudonymous input" to the contract must come in the form of a message signed with the corresponding private key.

Protocol ProtBlockchain(C)

The ProtBlockchain protocol is parameterized by a scratch-off puzzle SOP, and an unforgeable signature scheme SIG.

Init: Upon initialization, perform the following:

Receive the initial state for the contract C. Run a local instance of C, initialized with this state.

Initialize a pending transactions buffer, $\mathsf{buf} := \emptyset$

Initialize a committed transaction log, $\mathsf{log} := \{\}$

GenNym: On input $\texttt{gennym}(\mathsf{payload})$ from $\mathcal{E}$, do the following:

Generate a new signing keypair $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{SIG.Keygen}_{\mathrm{sign}}(1^\lambda)$.

Record $\bar{P} := (\mathsf{spk} \| \mathsf{payload})$ as a newly created pseudonym. Output $\bar{P}$ to $\mathcal{E}$.

Pseudonymous send: Upon receiving input $\texttt{nymSend}(m, \bar{P})$, where $\bar{P}$ is a previously generated pseudonym,

Compute $\sigma' := \mathsf{SIG.Sign}(\mathsf{ssk}, (\mathsf{nonce}, m))$ where ssk is the recorded secret signing key corresponding to $\bar{P}$, and nonce is a freshly generated random string. Let $\sigma := (\mathsf{nonce}, \sigma')$.

Send $\texttt{multicast}(\texttt{ptx}(m, \bar{P}, \sigma))$ to the underlying $\mathcal{F}_{\text{DIFFUSE}}$ functionality.

Anonymous send: Upon receiving input $\texttt{anonSend}(m)$ from $\mathcal{E}$:

Send $\texttt{multicast}(\texttt{atx}(m))$ to the underlying $\mathcal{F}_{\text{DIFFUSE}}$ functionality.

Buffer incoming transactions: Upon receiving a message of the form $\texttt{multicast}(\texttt{ptx}(m, \bar{P}, \sigma))$ (or of the form $\texttt{multicast}(\texttt{atx}(m))$), append $\texttt{ptx}(m)$ (or $\texttt{atx}(m)$) to the pending transactions buffer buf.

Committing Transactions: We proceed in consecutive epochs, where epoch lasts for a duration $\Delta^* = \lambda x \hat{r} + \Delta$, where $\hat{r}$ and x are the appropriately chosen parameters of the Nakamoto Consensus protocol (see Section 3.4), and Δ is the duration of one communication round. In each epoch, do the following:

Wait for Δ time to elapse (so that messages sent at the beginning of this epoch propagate to every honest party). Let $\underline{\mathsf{buf}} := \mathsf{buf}$ be a snapshot of the buffer at this point.

Next, for each of λx iterations, repeat the following:

Run an instance of Nakamoto Consensus with $\underline{\mathsf{buf}}$ as the input

When the Nakamoto Consensus protocol terminates with output newTxs (after $\hat{r}$ rounds), add newTxs to the log of committed transactions, log, and set $\mathsf{buf} := \mathsf{buf} - \mathsf{newTxs}$.

At the end of each epoch, do the following:

For each $\mathsf{tx} \in \mathsf{log}$ that can be parsed as $\texttt{ptx}(m, \bar{P}, (\sigma', \mathsf{nonce}))$, assert that $\mathsf{SIG.Verify}(\bar{P}.\mathsf{spk}, (\mathsf{nonce}, m), \sigma') = 1$, and that (m, nonce) has not been processed previously. Pass m to C as a pseudonymous message from $\bar{P}$.

For each $\mathsf{tx} \in \mathsf{log}$ that can be parsed as $\texttt{atx}(m)$, check that m has not been processed previously. Pass m to C as an anonymous message.

Clear log $:= \{\}$

FIGURE 6.4: Implementation of $\mathcal{F}_{\text{BLOCKCHAIN}}(\cdot)$ using Nakamoto consensus

6.4.2 Ideal World Simulator

We now prove that the protocol in Figure 6.4 is a secure and correct implementation of $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{C}))$ for an arbitrary transparent contract program C. For any real-world adversary $\mathcal{A}$, we construct an ideal-world simulator $\mathcal{S}$, such that no polynomial-time environment $\mathcal{E}$ can distinguish whether it is in the real or ideal world.

According to Canetti [6], to prove this theorem it suffices to construct a simulator $\mathcal{S}$ for the dummy adversary that simply passes messages to and from the environment $\mathcal{E}$. In Figure 6.5 we define the dummy simulator for $\mathsf{ProtBlockchain}$.

Theorem 6.1. *Assuming that the signature scheme* SIG *is unforgeable, and the conditions are satisfied for the Nakamoto Consensus protocol instantiated with* SOP *(as in Theorem 3.8, then our protocol in Figure 6.4 securely emulates the ideal functionality* $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{C}))$.

6.4.2.1 Indistinguishability of Real and Ideal Worlds

To prove indistinguishability of the real and ideal worlds from the perspective of the environment, we will go through a sequence of hybrid games.

Real world. We start with the real world and a dummy adversary that simply passes messages to and from the environment $\mathcal{E}$.

Hybrid 1. The simulator simulates the $\mathcal{F}_{\text{DIFFUSE}}$ functionality internally. Since all messages to the $\mathcal{F}_{\text{DIFFUSE}}$ are public, simulating the contract functionality is trivial. Therefore, Hybrid 1 is identically distributed as the real world from the environment $\mathcal{E}$'s view.

Hybrid 2. Hybrid 2 is the same as Hybrid 1 except the following changes. When an honest party sends a message to the contract (now simulated by the simulator $\mathcal{S}$), it will sign the message with a signature verifiable under an honestly generated nym. In Hybrid 1, the simulator will replace all honest parties' nyms and generate these nyms itself. In this way, the simulator will simulate honest parties' signatures by signing them itself. Hybrid 2 is identically distributed as Hybrid 1 from the environment $\mathcal{E}$'s view.

Hybrid 3. Hybrid 3 is the same as Hybrid 2 except for the following changes. Whenever the environment $\mathcal{E}$ passes to the simulator $\mathcal{S}$ a message signed on behalf of an honest party's nym, if the message/signature pair was not among the ones previously passed to the environment $\mathcal{E}$, then the simulator $\mathcal{S}$ aborts.

Assume that the signature scheme employed is unforgeable; then the probability of aborting in Hybrid 3 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 3 would otherwise be identically distributed as Hybrid 2 modulo aborting.

Simulator $\mathcal{S}$ for ProtBlockchain

Init. The simulator $\mathcal{S}$ maintains an internal copy of the $\mathcal{F}_{\text{DIFFUSE}}$ functionality, including a leaks buffer a queue of scheduled tasks, as well as an internal execution of the ProtBlockchain protocol.

Simulating honest parties.

- **GenNym:** Environment $\mathcal{E}$ sends input gennym(payload) to an honest party $\mathcal{P}$: the simulator $\mathcal{S}$ receives notification gennym(payload) from the ideal functionality. Simulator $\mathcal{S}$ honestly generates a signing key as defined in Figure 6.4, and remembers the corresponding secret keys. Finally, the simulator passes the nym $\bar{P} = (\mathsf{spk}, \mathsf{payload})$ to the ideal functionality.
- **Pseudonymous Send.** Environment $\mathcal{E}$ sends input nymSend$(m, \bar{P})$ to an honest party $\mathcal{P}$ that owns the pseudonym $\bar{P}$: the simulator $\mathcal{S}$ receives notification nymSend$(m, \bar{P})$. Simulator $\mathcal{S}$ generates a correct signature σ on the message m using the secret key associated with $\bar{P}$, and schedules delivery of multicast(ptx$(m, \bar{P}, \sigma)$) to every correct party in $\mathcal{F}_{\text{DIFFUSE}}$.
- **Anonymous Send.** Environment $\mathcal{E}$ sends input anonSend(m) to an honest party $\mathcal{P}$: the simulator $\mathcal{S}$ receives notification anonSend(m). Simulator $\mathcal{S}$ schedules delivery of multicast(atx(m)) to every correct party in $\mathcal{F}_{\text{DIFFUSE}}$.

Simulating instructions to the dummy adversary.

- **Deliver.** Upon receiving deliver(i) from the environment $\mathcal{E}$, pass deliver(i) to the internally running instance of $\mathcal{F}_{\text{DIFFUSE}}$.
- **Get Leaks.** Upon receiving getleaks from the environment $\mathcal{E}$, return the contents of leaks in the internally running instance of $\mathcal{F}_{\text{DIFFUSE}}$.

Simulating corrupted parties.

- **Multicast.** Upon receiving multicast(m) from the environment $\mathcal{E}$ on behalf of a corrupted party $\mathcal{P}$, schedule m to be delivered to each simulated honest party in the next communication round via the internally running instance of $\mathcal{F}_{\text{DIFFUSE}}$. This includes messages of the form ptx$(\cdot)$ and atx$(\cdot)$, as well as messages related to the Nakamoto Consensus protocol.
- **Applying Transactions.** After the end of each epoch, we can assume that the contents of log are identical for each party in the internally simulated protocol. Process each item in log the same as in ProtBlockchain. For items of the form ptx$(m, \bar{P}, \sigma)$, verify if σ is a valid signature of m under $\bar{P}$, and if m has not been previously applied.

FIGURE 6.5: Simulator for ProtBlockchain.

That Hybrid 3 is computationally indistinguishable from the ideal simulation to any polynomial-time environment $\mathcal{E}$ follows from the *Agreement* and *Validity* properties of the consensus protocol. With high probability, the log after each epoch is identical for all parties. Furthermore, $\underline{\mathsf{buf}}$ contains every transaction submitted by an honest party prior to the beginning of the epoch; by running the consensus protocol for λx consecutive iterations, we guarantee that with high probability, the $\underline{\mathsf{buf}}$ of some honest party is committed during at least one iteration. This concludes the proof of Theorem 6.1.

Chapter 6

Privacy Preserving Smart Contracts

7.1 Overview

Despite the expressiveness and power of blockchain and smart contracts, the present form of these technologies *lacks transactional privacy.* The entire sequence of actions taken in a smart contract is propagated across the network and/or recorded on the blockchain, and therefore is publicly visible. Even though parties can create new pseudonymous public keys to increase their anonymity, the values of all transactions and balances for each (pseudonymous) public key are publicly visible. Further, recent works have also demonstrated deanonymization attacks by analyzing the transactional graph structures of cryptocurrencies [30, 31]. The privacy requirements of many financial transactions will likely preclude the use of existing smart contract systems. Although there has been progress in designing privacy-preserving cryptocurrencies such as Zerocash [36] and several others [37–39], these systems forgo programmability, and it is unclear *a priori* how to enable programmability without exposing transactions and data in cleartext to miners.

We propose Hawk, a framework for building privacy-preserving smart contracts. With Hawk, a *non-specialist* programmer can easily write a Hawk program without having to implement any cryptography. Our Hawk compiler automatically compiles the program to a cryptographic protocol between the blockchain and the users. A Hawk program contains two parts:

- A *private contract* program denoted ϕ_{priv} which takes in parties' input data (e.g., choice in a "rock, paper, scissors" game) as well as currency units (e.g., bids in an auction). The program ϕ_{priv} performs computation to determine the payout distribution among the parties. For example, in an auction, winner's bid goes to the seller, and others' bids are refunded. The private contract ϕ_{priv} is meant to protect the participants' data and the exchange of money.
- A *public contract* program denoted ϕ_{pub} that does not touch private data or money.

Our Hawk compiler compiles the public contract to execute directly on the public blockchain, and compiles the private contract ϕ_{priv} into a cryptographic protocol involving the following pieces: *i)* protocol logic to be executed by the blockchain; and *ii)* protocol logic to be executed by contractual parties.

Security guarantees. Hawk's security guarantees encompass two main aspects:

- *On-chain privacy.* On-chain privacy stipulates that transactional privacy be provided against the public (i.e., against any party *not* involved in the contract) – unless the contractual parties themselves voluntarily disclose information. Although in Hawk protocols, users exchange data with the blockchain, and rely on it to ensure fairness against aborts, the flow of money and amount transacted in the private contract ϕ_{priv} is cryptographically hidden from the public's view.
- *Contractual security.* While on-chain privacy protects contractual parties' privacy against the public (i.e., parties not involved in the contract), contractual security protects parties in the same contractual agreement from *each other.* Hawk assumes that contractual parties act *selfishly* to maximize their own financial interest. In particular, they can *arbitrarily* deviate from the prescribed protocol or even *abort* prematurely. Therefore, contractual security is a multi-faceted notion that encompasses not only cryptographic notions of confidentiality and authenticity, but also financial fairness in the presence of cheating and aborting behavior. The best way to understand contractual security is through a concrete example, and we refer the reader to Section 7.5 for a more detailed explanation.

Minimally trusted manager. Hawk-generated protocols assume a special contractual party called a manager (e.g., an auction manager) besides the normal users. The manager aids the efficient execution of our cryptographic protocols while being minimally trusted. A skeptical reader may worry that our use of such a party trivializes our security guarantees; we dispel this notion by directly explaining what a corrupt manager can and cannot do: Although a manager can see the transactions that take place in a private contract, *it cannot affect the outcome of the contract, even when it colludes with other users.* In the event that a manager aborts the protocol, it can be financially penalized, and users obtain remuneration accordingly. The manager also need not be trusted to maintain the security or privacy of the underlying currency (e.g., it cannot double-spend, inflate the currency, or deanonymize users). Furthermore, if multiple contract instances run concurrently, each contract may specify a different manager and the effects of a corrupt manager are confined to that instance.

7.1.1 Conventions for Writing Programs

Thanks to our wrapper-based modularized notational system, The ideal program and the contract program are the main locations where user-supplied, custom program logic is defined. We use the following conventions for writing the ideal program and the contract program.

Delayed processing in ideal programs. When writing the contract program, every message received by the contract program is already delayed by a round due to the $\mathcal{F}_{\text{WRAP}}(\cdot)$ wrapper.

When writing the ideal program, we introduce a simple convention to denote delayed computation. Program instructions that are written in gray background denote computation that does not take place immediately, but is deferred to the beginning of the next epoch. This is a convenient shorthand because in our real-world protocol, effectively any computation done by a contract functionality will be delayed. For example, in our $\mathsf{Ideal}_{\mathsf{cash}}$ ideal program (see Figure 7.1), whenever the ideal functionality receives a `mint` or `pour` message, the adversary is notified immediately; however, processing of the messages is deferred till the next round. Formally, delayed processing can be implemented simply by storing state and invoking the delayed program instructions on the next `tick`. To avoid ambiguity, we assume that by convention, the delayed instructions are invoked at the beginning of the `tick` call. In other words, upon the next timer click, the delayed instructions are executed first.

Pseudonymity. All party identifiers that appear in ideal programs, contract programs, and user-side programs by default refer to *pseudonyms*. When we write "upon receiving message from *some* P", this accepts a message from any pseudonym. Whenever we write "upon receiving message from P", without the keyword *some*, this accepts a message from a fixed pseudonym P, and typically which pseudonym we refer to is clear from the context.

Whenever we write "send m to $\mathcal{F}_{\text{BLOCKCHAIN}}$ as nym P" inside a user program, this sends an internal message ("send", m, P) to the protocol wrapper Π. The protocol wrapper will then authenticate the message appropriately under pseudonym P. When the context is clear, we avoid writing "as nym P", and simply write "send m to $\mathcal{F}_{\text{BLOCKCHAIN}}$". Our formal system also allows users to send messages anonymously to a contract – although this option will not be used in this paper.

We want to be able to allow the contract to have an initial state (i.e., an arbitrary initial allocation of money for each party), which formally speaking, must depend on the pseudonyms. To allow for this in our model, we let the functionality to depend on a protocol-specific pseudonym generation, Keygen, as a parameter. During the initialization, the functionality generates a pseudonym for each party. The secret key for each pseudonym is sent to each party, while the environment receives the pseudonyms. Note that this does not amount to creating a PKI (i.e., with exactly N publicly-known pseudonyms, which would trivialize the need for an anonymous consensus protocol), since the adversary can also generate an arbitrary number of pseudonyms.

Ledger and money transfers. A public ledger is denoted ledger in our ideal programs and contract programs. When a party sends amt to an ideal program or a contract program, this represents an ordinary message transmission. Money transfers only take place when ideal programs or contract programs update the public ledger ledger. In other words, the symbol $ is only adopted for readability (to distinguish variables associated with money and other variables), and does not have special meaning or significance. One can simply think of this variable as having the money type.

7.2 Warmup: Private Cash and Money Transfers

7.2.1 Private Cash Specification $\mathsf{Ideal}_{\mathsf{cash}}$

To begin, we define the $\mathsf{Ideal}_{\mathrm{cash}}$ contract program (in Figure 7.1), which specifies the requirements of a private ledger and currency transfer system. At a high-level, this functionality keeps track of both the original ledger of publicly-visible (but pseudonymous) account balances (inherited from $\mathsf{ContractLedger}$), as well as a new ledger of private account balances (called coins).

We adopt the same `mint` and `pour` terminology from Zerocash [36]. A `mint` transaction allows a user to move money from the public ledger into a new private coin he controls. A `pour` transaction is used to transfer money to a recipient (referred to by their pseudonym). The specification ensures the natural invariants: the total quantity of money is conserved and a party can only spend money that they have previously received. However, unlike public ledger transfers, a `pour` transaction does not reveal any information about *which* private coin is being spent. This effectively breaks the transaction graph, thwarting attempts to reconstruct the transaction history and deanonymize transactions.

Informally speaking, prior works such as Zerocash [36] are meant to realize (approximations of) this ideal functionality – although technically this ought to be interpreted with the caveat that these earlier works prove indistinguishability or game-based security instead UC-based simulation security. As we discuss later, our simulation-based notion is subtly stronger.

Public ledger. The $\mathsf{Ideal}_{\mathrm{cash}}$ contract also includes all the functionality of the transparent ledger contract by inheriting from $\mathcal{T}(\mathsf{ContractLegder})$.

Mint. The `mint` operation allows a user $\mathcal{P}$ to transfer money from the public ledger denoted ledger to the private pool denoted $\mathsf{Coins}[\mathcal{P}]$. With each transfer, a private coin for user $\mathcal{P}$ is created, and associated with a value val.

For correctness, the ideal program $\mathsf{Ideal}_{\mathrm{cash}}$ checks that the user $\mathcal{P}$ has sufficient funds in its public ledger $\mathsf{ledger}[\mathcal{P}]$ before creating the private coin.

Pour. The `pour` operation allows a user $\mathcal{P}$ to spend money in its private bank privately. For simplicity, we define the simple case with two input coins and two output coins. This is sufficient for users to transfer any amount of money by "making change," although it would be straightforward to support more efficient batch operations as well.

For correctness, the ideal program $\mathsf{Ideal}_{\mathrm{cash}}$ checks the following: 1) for the two input coins, party $\mathcal{P}$ indeed possesses private coins of the declared values; and 2) the two input coins sum up to equal value as the two output coins, i.e., coins neither get created or vanish.

Privacy. When an honest party $\mathcal{P}$ mints, the ideal-world adversary $\mathcal{A}$ learns the pair $(\mathcal{P}, \mathsf{val})$ – since minting is raising coins from the public pool to the private pool. Operations on the public pool are observable by $\mathcal{A}$.

When an honest party $\mathcal{P}$ pours, however, the adversary $\mathcal{A}$ learns only the output pseudonyms $\mathcal{P}_1$ and $\mathcal{P}_2$. It does not learn which coin in the private pool Coins is being spent nor the name of the spender. Therefore, the spent coins are anonymous with respect to the private pool Coins. To get strong anonymity, new pseudonyms $\mathcal{P}_1$ and $\mathcal{P}_2$ can be generated on the fly to receive each pour. We stress that as long as `pour` hides the sender, this "breaks" the transaction graph, thus preventing linking analysis.

If a corrupted party is the recipient of a pour, the adversary additionally learns the value of the coin it receives.

Additional subtleties. Later in our protocol, honest parties keep track of a wallet of coins. Whenever an honest party pours, it first checks if an appropriate coin exists in its local wallet – and if so it immediately removes the coin from the wallet (i.e., without delay). In this way, if an honest party makes multiple pour transactions in one round, it will always choose distinct coins for each pour transaction. Therefore, in our $\mathsf{Ideal}_{\mathrm{cash}}$functionality, honest pourers' coins are immediately removed from Coins. Further, an honest party is not able to spend a coin paid to itself until the next round. By contrast, corrupted parties are allowed to spend coins paid to them in the same round – this is due to the fact that any message is routed immediately to the adversary, and the adversary can also choose a permutation for all messages received by the contract in the same round (see Section 6.3).

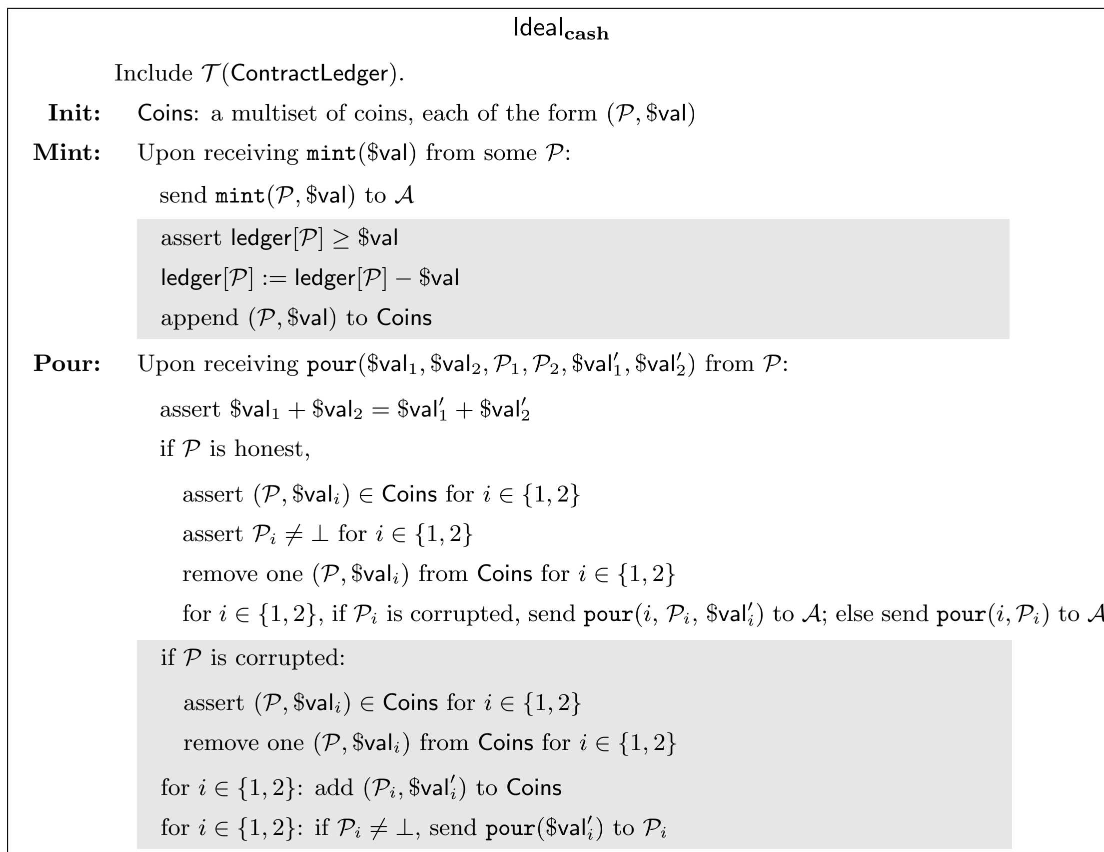
Another subtlety in the $\mathsf{Ideal}_{\mathrm{cash}}$functionality is that honest parties will always pour to existing pseudonyms. However, the functionality allows the adversary to pour to non-existing pseudonyms denoted $\perp$ – in this case, effectively the private coin goes into a blackhole and cannot be retrieved. This enables a performance optimization in our $\mathsf{ProtCash}$ and $\mathsf{ContractCash}$ protocol later – where we avoid including the ct_i's in the NIZK of $\mathcal{L}_{\mathtt{POUR}}$ (see Section 7.2). If a malicious pourer chooses to compute the wrong ct_i, it is as if the recipient $\mathcal{P}_i$ did not receive the pour, i.e., the pour is made to $\perp$.

7.2.2 Construction of Private Cash

Our construction adopts a Zerocash-like protocol for implementing private cash and private currency transfers. For completeness, we give a brief explanation below, and we mainly focus on the `pour` operation which is technically more interesting. The contract $\mathsf{ContractCash}$ maintains a set Coins of private coins. Each private coin is stored in the format

$$(\mathcal{P}, \mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val}))$$

where $\mathcal{P}$ denotes a party's pseudonym, and coin commits to the coin's value $\$\mathsf{val}$ under randomness s.



$\mathsf{Ideal}_{\mathbf{cash}}$

Include $\mathcal{T}(\mathsf{ContractLedger})$.

Init: Coins: a multiset of coins, each of the form $(\mathcal{P}, \$\mathsf{val})$

Mint: Upon receiving $\mathtt{mint}(\$\mathsf{val})$ from some $\mathcal{P}$:

send $\mathtt{mint}(\mathcal{P}, \$\mathsf{val})$ to $\mathcal{A}$

assert $\mathsf{ledger}[\mathcal{P}] \geq \$\mathsf{val}$

$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] - \$\mathsf{val}$

append $(\mathcal{P}, \$\mathsf{val})$ to Coins

Pour: Upon receiving $\mathtt{pour}(\$\mathsf{val}_1, \$\mathsf{val}_2, \mathcal{P}_1, \mathcal{P}_2, \$\mathsf{val}'_1, \$\mathsf{val}'_2)$ from $\mathcal{P}$:

assert $\$\mathsf{val}_1 + \$\mathsf{val}_2 = \$\mathsf{val}'_1 + \$\mathsf{val}'_2$

if $\mathcal{P}$ is honest,

assert $(\mathcal{P}, \$\mathsf{val}_i) \in \mathsf{Coins}$ for $i \in \{1, 2\}$

assert $\mathcal{P}_i \neq \perp$ for $i \in \{1, 2\}$

remove one $(\mathcal{P}, \$\mathsf{val}_i)$ from Coins for $i \in \{1, 2\}$

for $i \in \{1, 2\}$, if $\mathcal{P}_i$ is corrupted, send $\mathtt{pour}(i,\ \mathcal{P}_i,\ \$\mathsf{val}'_i)$ to $\mathcal{A}$; else send $\mathtt{pour}(i, \mathcal{P}_i)$ to $\mathcal{A}$

if $\mathcal{P}$ is corrupted:

assert $(\mathcal{P}, \$\mathsf{val}_i) \in \mathsf{Coins}$ for $i \in \{1, 2\}$

remove one $(\mathcal{P}, \$\mathsf{val}_i)$ from Coins for $i \in \{1, 2\}$

for $i \in \{1, 2\}$: add $(\mathcal{P}_i, \$\mathsf{val}'_i)$ to Coins

for $i \in \{1, 2\}$: if $\mathcal{P}_i \neq \perp$, send $\mathtt{pour}(\$\mathsf{val}'_i)$ to $\mathcal{P}_i$

FIGURE 7.1: Definition of $\mathsf{Ideal}_{\mathsf{cash}}$. Notation: ledger denotes the public ledger, and Coins denotes the private pool of coins. As mentioned in Section 7.1.1, gray background denotes batched and delayed activation. All party names correspond to pseudonyms due to notations and conventions defined in Section 6.3.

During a $\mathtt{pour}$ operation, the spender $\mathcal{P}$ chooses two coins in Coins to spend, denoted $(\mathcal{P}, \mathsf{coin}_1)$ and $(\mathcal{P}, \mathsf{coin}_2)$ where $\mathsf{coin}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i)$ for $i \in \{1, 2\}$. The $\mathtt{pour}$ operation pays val'_1 and val'_2 amount to two output pseudonyms denoted $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively, such that $\mathsf{val}_1 + \mathsf{val}_2 = \mathsf{val}'_1 + \mathsf{val}'_2$. The spender chooses new randomness s'_i for $i \in \{1, 2\}$, and computes the output coins as

$$\left(\mathcal{P}_i, \mathsf{coin}_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)\right)$$

The spender gives the values s'_i and val'_i to the recipient $\mathcal{P}_i$ for $\mathcal{P}_i$ to be able to spend the coins later.

Now, the spender computes a zero-knowledge proof to show that the output coins are constructed appropriately, where correctness compasses the following aspects:

- *Existence of coins being spent.* The coins being spent $(\mathcal{P}, \mathsf{coin}_1)$ and $(\mathcal{P}, \mathsf{coin}_2)$ are indeed part of the private pool Coins. We remark that here the zero-knowledge property allows the spender to hide which coins it is spending – this is the key idea behind transactional privacy.
  To prove this efficiently, $\mathsf{ContractCash}$ maintains a Merkle tree MT over the private pool Coins. Membership in the set can be demonstrated by a Merkle branch consistent with the root hash, and this is done in zero-knowledge.
- *No double spending.* Each coin $(\mathcal{P}, \mathsf{coin})$ has a cryptographically unique serial number sn that can be computed as a pseudorandom function of $\mathcal{P}$'s secret key and coin. To pour a coin, its

serial number sn must be disclosed, and a zero-knowledge proof given to show the correctness of sn. $\mathsf{ContractCash}$ checks that no sn is used twice.

- *Money conservation.* The zero-knowledge proof also attests to the fact that the input coins and the output coins have equal total value.

We make some remarks about the security of the scheme. Intuitively, when an honest party pours to an honest party, the adversary $\mathcal{A}$ does not learn the values of the output coins assuming that the commitment scheme Comm is hiding, and the NIZK scheme we employ is computational zero-knowledge. The adversary $\mathcal{A}$ can observe the nyms that receive the two output coins. However, as we remarked earlier, since these nyms can be one-time, leaking them to the adversary would be okay. Essentially we only need to break linkability at spend time to ensure transactional privacy.

When a corrupted party $\mathcal{P}^*$ pours to an honest party $\mathcal{P}$, even though the adversary knows the opening of the coin, it cannot spend the coin $(\mathcal{P}, \mathsf{coin})$ once the transaction takes effect by the $\mathsf{ContractCash}$, since $\mathcal{P}^*$ cannot demonstrate knowledge of $\mathcal{P}$'s secret key. We stress that since the contract binds the owner's nym $\mathcal{P}$ to the coin, only the owner can spend it even when the opening of coin is disclosed.

7.2.3 Secure Emulation Proof for Private Cash

We now prove that the protocol in Figure 7.2 is a secure and correct implementation of $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{hawk}})$. For any real-world adversary $\mathcal{A}$, we construct an ideal-world simulator $\mathcal{S}$, such that no polynomial-time environment $\mathcal{E}$ can distinguish whether it is in the real or ideal world. We first describe the construction of the simulator $\mathcal{S}$ and then argue the indistinguishability of the real and ideal worlds.

Theorem 7.1. *Assuming that the hash function in the Merkle tree is collision resistant, the commitment scheme* Comm *is perfectly binding and computationally hiding, the* NIZK *scheme is computationally zero-knowledge and simulation sound extractable, the encryption schemes* ENC *and* SENC *are perfectly correct and semantically secure, the PRF scheme* PRF *is secure, then our protocol in Figure 7.2 securely emulates the ideal functionality* $\mathcal{F}(\mathsf{Ideal}_{cash})$.

7.2.3.1 Simulator Wrapper

Looking ahead, we will need to use many of the same mechanisms for simulating our full application as for private cash. Therefore we factor out much of the functionality of this into a simulator wrapper. The simulator wrapper is formally defined in Figure 7.3. Below we describe the high level approach of our simulation strategy, and the role of the simulator wrapper.

Due to Canetti [6], it suffices to construct a simulator $\mathcal{S}$ for the dummy adversary that simply passes messages to and from the environment $\mathcal{E}$. The ideal-world simulator $\mathcal{S}$ also interacts with the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal})$ functionality in the ideal world.

Since the real world dummy adversary would be able to reveal the entire state of the transparent contract to the environment upon request (via the `exposestate` instruction), the simulator wrapper must also be able to provide a similar view. To help with this, the simulator wrapper maintains an internal simulation of the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{contract})$ functionality, which can be viewed at any time.

A key challenge comes from the fact that our ideal functionalities have private state, which are represented by commitments in our protocols. Updates to the private state come along with zero knowledge proofs in the protocol. The simulator must be able to create plausible-looking proofs and commitments about information that it does not know! To take care of this, the simulator uses the simulator setup $\mathsf{NIZK}.\widehat{\mathcal{K}}$, which enables the simulator to create false proofs and commitments.

A second challenge is that the simulator must be able to decrypt messages sent from corrupted partes to honest parties. As one example, a `pour` transaction that sends coins from a corrupted party to an honest party must contain an encryption of the amount of coins, which the honest must be able to decrypt. To achieve this, the simulator generates encryption keypairs for the simulated honest parties, and keeps the secret key available.

The simulator wrapper also maintains a perfect simulation of the public ledger ledger. Since the state of ledger is publicly visible even in the ideal functionality, the simulator maintains a perfect correspondence between ledger in the ideal world and ledger in the local simulation of $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{C}))$.

In summary, the simulator wrapper performs the following behaviors:

- The simulator wrapper runs an internal instance of the real world functionality, $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{C}))$.
- The simulator wrapper runs the simulated setup $\mathsf{NIZK}.\widehat{\mathcal{K}}$ to create the crs, and retains the "trapdoor" information for forging proofs and extracting witnesses.
- The simulator wrapper simulates the additional $\mathsf{payload}$ for the pseudonyms of honest parties, consisting of the PRF secret $\mathsf{PRF}_{\mathsf{sk}}$ and the public key encryption secret key esk.
- The simulator wrapper maintains a perfect simulation of the public ledger ledger.

7.2.3.2 Ideal World Simulator for $\mathsf{Ideal}_{\text{cash}}$

Below we construct the $\mathsf{Ideal}_{\text{cash}}$-specific portion of our simulator simP. Our overall simulator $\mathcal{S}$ can be obtained by applying the simulator wrapper $\mathcal{S}(\mathsf{simP})$. The simulator wrapper modularizes the simulator construction by factoring out the common part of the simulation pertaining to all protocols in this model of execution.

Recall that the simulator wrapper performs the ordinary setup procedure, but retains the "trapdoor" information used in creating the crs for the NIZK proof system, allowing it to forge proofs for false statement and to extract witnesses from valid proofs. Such a `pour` transaction contains a zero-knowledge proof involving the values of coins being spent or created; the simulator must rely

on its ability to extract witnesses in order to learn these values and trigger $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{cash}})$ appropriately.

The environment may also send `mint` and `pour` instructions to honest parties that in the ideal world would be forwarded directly to $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{cash}})$. These activate the simulator, but only reveal partial information about the instruction — in particular, the simulator does not learn the values of the coins being spent. The simulator handles this by passing *bogus* (but plausible-looking) transactions to ContractCash.

Thus the simulator must translate transactions submitted by corrupt parties to the contract into ideal world instructions, and must translate ideal world instructions into transactions published on the contract.

The simulator simP is defined in more detail below:

Simulating corrupted parties. The following messages are sent by the environment $\mathcal{E}$ to the simulator $\mathcal{S}(\mathsf{simP})$ which then forwards it on to both the internally simulated contract G(ContractCash) and the inner simulator simP.

- simP receives a pseudonymous mint message $\texttt{mint}(\$\mathsf{val}, r)$. No extra action is necessary.
- simP receives an anonymous pour message, $\texttt{pour}(\{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i, \mathsf{ct}_i\}_{i\in\{1,2\}\}})$. The simulator uses τ to extract the witness from π, which includes the sender $\mathcal{P}$ and values $\$\mathsf{val}_1$, $\$\mathsf{val}_2$, $\$\mathsf{val}'_1$ and $\$\mathsf{val}'_2$. If $\mathcal{P}_i$ is an uncorrupted party, then the simulator must check whether each encryption ct_i is performed correctly, since the NIZK proof does not guarante that this is the case. The simulator performs a trial decryption using $\mathcal{P}_i.\mathsf{esk}$; if the decryption is *not* a valid opening of coin_i, then the simulator must avoid causing $\mathcal{P}_i$ in the ideal world to output anything (since $\mathcal{P}_i$ in the real world would not output anything either). The simulator therefore substitutes some default value (e.g., the name of any corrupt party $\mathcal{P}$) for the recipient's pseudonym. The simulator forwards $\texttt{pour}(\$\mathsf{val}_1, \$\mathsf{val}_2, \mathcal{P}_1^\dagger, \mathcal{P}_2^\dagger, \$\mathsf{val}'_1, \$\mathsf{val}'_2)$ anonymously to $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{cash}})$, where $\mathcal{P}_i^\dagger = \mathcal{P}$ if $\mathcal{P}_i$ is uncorrupted and decryption fails, and $\mathcal{P}_i^\dagger = \mathcal{P}_i$ otherwise.

Simulating honest parties. When the environment $\mathcal{E}$ sends inputs to honest parties, the simulator $\mathcal{S}$ needs to simulate messages that corrupted parties receive, from honest parties or from functionalities in the real world. The honest parties will be simulated as below:

- Environment $\mathcal{E}$ gives a `mint` instruction to party $\mathcal{P}$. The simulator simP receives $\texttt{mint}(\mathcal{P}, \$\mathsf{val}, r)$ from the ideal functionality $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{cash}})$. The simulator has enough information to run the honest protocol, and posts a valid `mint` transaction to the contract.
- Environment $\mathcal{E}$ gives a `pour` instruction to party $\mathcal{P}$. The simulator simP receives $\texttt{pour}(\mathcal{P}_1, \mathcal{P}_2)$ from $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{cash}})$. However, the simulator does not learn the name of the honest sender $\mathcal{P}$, or the correct values for each input coin val_i (for $i \in \{1, 2\}$). Instead, the simulator uses τ to create a false proof using arbitrary values for these values in the witness.

To generate each serial number sn_i in the witness, the simulator chooses a random element from the codomain of PRF. For each recipient $\mathcal{P}_i$ (for $i \in \{1, 2\}$), the simulator behaves differently depending on whether or not $\mathcal{P}_i$ is corrupted:

Case 1: $\mathcal{P}_i$ *is honest.* The simulator does not know the correct output value, so instead sets $\mathsf{val}'_i := 0$, and computes coin'_i and ct_i as normal. The environment therefore sees a commitment and an encryption of 0, but without $\mathcal{P}_i.\mathsf{esk}$ it cannot distinguish between an encryption of 0 or of the correct value.

Case 2: $\mathcal{P}_i$ *is corrupted.* Since the ideal world recipient would receive $\$\mathsf{val}'_i$ from $\mathcal{F}_{\text{CASH}}$, and since $\mathcal{P}_i$ is corrupted, the simulator learns the correct value $\$\mathsf{val}'_i$ directly. Hence coin_i is a correct encryption of $\$\mathsf{val}'_i$ under $\mathcal{P}_i$'s registered encryption public key.

7.2.3.3 Indistinguishability of Real and Ideal Worlds

To prove indistinguishability of the real and ideal worlds from the perspective of the environment, we will go through a sequence of hybrid games.

Real world. We start with the real world with a dummy adversary that simply passes messages to and from the environment $\mathcal{E}$.

Hybrid 1. Hybrid 1 is the same as the real world, except that now the adversary (also referred to as the simulator) will call $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \mathsf{NIZK}.\widehat{\mathcal{K}}(1^\lambda)$ to perform a simulated setup for the NIZK scheme. The simulator will pass the simulated $\widehat{\mathsf{crs}}$ to the environment $\mathcal{E}$. When an honest party $\mathcal{P}$ publishes a NIZK proof, the simulator will replace the real proof with a simulated NIZK proof before passing it onto the environment $\mathcal{E}$. The simulated NIZK proof can be computed by calling the $\mathsf{NIZK}.\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \cdot)$ algorithm which takes only the statement as input but does not require knowledge of a witness.

Fact 2. It is immediately clear that if the NIZK scheme is computational zero-knowledge, then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 1 from the real world except with negligible probability.

Hybrid 2. The simulator simulates the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{ContractCash}))$ functionality. Since all messages to the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{ContractCash}))$ functionality are public, simulating the contract functionality is trivial. Therefore, Hybrid 2 is identically distributed as Hybrid 1 from the environment $\mathcal{E}$'s view.

Hybrid 3. Hybrid 3 is the same as Hybrid 2 except the following changes. When an honest party sends a message to the contract (now simulated by the simulator $\mathcal{S}$), it will sign the message with a signature verifiable under an honestly generated nym. In Hybrid 3, the simulator will replace all honest parties' nyms and generate these nyms itself. In this way, the simulator will simulate honest parties' signatures by signing them itself. Hybrid 3 is identically distributed as Hybrid 2 from the environment $\mathcal{E}$'s view.

Hybrid 4. Hybrid 4 is the same as Hybrid 3 except for the following changes:

- When an honest party $\mathcal{P}$ produces a ciphertext ct_i for a recipient $\mathcal{P}_i$, and if the recipient is also uncorrupted, then the simulator will replace this ciphertext with an encryption of 0 before passing it onto the environment $\mathcal{E}$.
- When an honest party $\mathcal{P}$ produces a commitment coin, then the simulator replaces this commitment with a commitment to 0.
- When an honest party $\mathcal{P}$ computes a pseudorandom serial number sn, the simulator replaces this with a randomly chosen value from the codomain of PRF.

Fact 3. It is immediately clear that if the encryption scheme is semantically secure, if PRF is a pseudorandom function, and if Comm is a perfectly hiding commitment scheme, then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 4 from Hybrid 3 except with negligible probability.

Hybrid 5. Hybrid 5 is the same as Hybrid 4 except for the following changes. Whenever the environment $\mathcal{E}$ passes to the simulator $\mathcal{S}$ a message signed on behalf of an honest party's nym, if the message and signature pair was not among the ones previously passed to the environment $\mathcal{E}$, then the simulator $\mathcal{S}$ aborts.

Fact 4. Assume that the signature scheme employed is secure; then the probability of aborting in Hybrid 5 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 5 would otherwise be identically distributed as Hybrid 4 modulo aborting.

Hybrid 6. Hybrid 6 is the same as Hybrid 5 except for the following changes. Whenever the environment passes $\mathtt{pour}(\pi, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i, \mathsf{ct}_i\})$ to the simulator (on behalf of corrupted party $\mathcal{P}$), if the proof π verifies under $\mathsf{statement}$, then the simulator will call the NIZK's extractor algorithm $\mathcal{E}$ to extract $\mathsf{witness}$. If the NIZK π verifies but the extracted $\mathsf{witness}$ does not satisfy the relation $\mathcal{L}_{\mathtt{POUR}}(\mathsf{statement}, \mathsf{witness})$, then abort the simulation.

Fact 5. Assume that the NIZK is simulation sound extractable, then the probability of aborting in Hybrid 6 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 6 would otherwise be identically distributed as Hybrid 5 modulo aborting.

Finally, observe that Hybrid 6 is computationally indistinguishable from the ideal simulation $\mathcal{S}$ unless one of the following bad events happens:

- A value val' decrypted by an honest recipient is different from that extracted by the simulator. However, given that the encryption scheme is perfectly correct, this cannot happen.
- A commitment coin is different than any stored in $\mathsf{ContractCash.coins}$, yet it is valid according to the relation $\mathcal{L}_{\mathtt{POUR}}$. Given that the merkle tree MT is computed using collision-resistant a hash function, this occurs with at most negligible probability.

- The honest public key generation algorithm results in key collisions. Obviously, this happens with negligible probability if the encryption and signature schemes are secure.

Fact 6. Given that the encryption scheme is semantically secure and perfectly correct, and that the signature scheme is secure, then Hybrid 6 is computationally indistinguishable from the ideal simulation to any polynomial-time environment $\mathcal{E}$.

7.2.4 Technical Subtleties in Zerocash

In general, a simulation-based security definition is more straightforward to write and understand than ad-hoc indistinguishability games – although it is often more difficult to prove or require a protocol with more overhead. Below we highlight a subtle weakness with Zerocash's security definition [36], which motivates our stronger definition.

The privacy guarantees of Zerocash [36] are defined by a "Ledger Indistinguishability" game (in [36], Appendix C.1). In this game, the attacker (adaptively) generates two sequences of queries, $Q_{\texttt{left}}$ and $Q_{\texttt{right}}$. Each query can either be a raw "insert" transaction (which corresponds in our model to a transaction submitted by a corrupted party) or else a "mint" or "pour" query (which corresponds in our model to an instruction from the environment to an honest party). The attacker receives (incrementally) a pair of views of protocol executions, $V_{\texttt{left}}$ and $V_{\texttt{right}}$, according to one of the following two cases, and tries to discern which case occurred: either $V_{\texttt{right}}$ is generated by applying all the queries in $Q_{\texttt{right}}$ and respectively for $V_{\texttt{right}}$; or else $V_{\texttt{left}}$ is generated by interweaving the "insert" queries of $Q_{\texttt{left}}$ with the "mint" and "pour" queries of $Q_{\texttt{right}}$, and $V_{\texttt{right}}$ is generated y interweaving the "insert" queries of $Q_{\texttt{right}}$ with the "mint" and "pour" queries of $Q_{\texttt{left}}$. The two sequences of queries are constrained to be "publicly consistent", which effectively defines the information leaked to the adversary. For example, the ith queries in both sequences must be of the same type (either "mint", "pour", or "insert"), and if a "pour" query includes an output to a corrupted recipient, then the output value must be the same in both queries.

However, the definition of "public consistency" is subtly overconstraining: it requires that if the ith query in one sequence is an (honest) "pour" query that spends a coin previously created by a (corrupt) "insert" query, then the ith queries in both sequences must spend coins of equal value created by prior "insert" queries. Effectively, this means that if a corrupted party sends a coin to an honest party, then the adversary may be alerted when the honest party spends it.

We stress that this does not imply any flaw with the Zerocash construction itself — however, there is no obvious path to proving their scheme secure under a simulation based paradigm. Our scheme avoids this problem by using an simulation extractable NIZK instead of a zkSNARK.

7.3 Combining Privacy with Programmable Logic

7.3.1 Overview

So far, ContractCash provides roughly the same service as Zerocash [36], only supporting *direct* money transfers between users. In this section, we show how to enable transactional privacy and programmable logic simultaneously.

We consider a "private smart contract" arranged between N parties, $\mathcal{P}_i$ for $1 \leq i \leq N$. The parties to the contract may or may not also be parties that participate (as "miners") in the underlying consensus protocol. The application is defined by user-provided function, ϕ_{priv}. Each of the parties provides a private input to the contract, including both a string and a quantity of private cash. After all the inputs are collected, they are provided as input to the function ϕ_{priv}, which produces an output for each party (also consisting an arbitrary string and quantity of cash). Roughly, we desire the following two requirements: First, the computation should be privacy-preserving, in the sense that it does not reveal information about the inputs or outputs of the parties to each other or to the public. Second, the computation should be performed correctly (i.e., the parties receive the correct outputs) and should satisfy the same invariants as $\mathsf{Ideal}_{\text{cash}}$(e.g., neither creating nor destroying money).

In this section we develop a protocol for such applications, based on zero-knowledge proofs. Our protocol also relies on a semi-trusted party, called the manager, who is responsible for collecting the parties' inputs and producing the zero-knowledge proofs. We provide fine-grained guarantees that restrict how much the manager can influence the protocol. In particular, the manager does not learn any party's input values until after all of the inputs are committed (e.g., the adversary cannot "front-run" in a sealed-bid auction), and the manager cannot produce incorrect outputs. These guarantees are captured in our ideal functionality specification, as explained shortly.

The content of this chapter was first published an Oakland 2016 conference paper [8]. The conference version also includes a practical protocol instantiation and optimized implementation due primarily to Ahmed Kosba. The contribution claimed in this thesis is restricted to the high-level protocol formalism and compiler design.

7.4 Private Smart Contract Specification

We now describe our cryptography abstraction in the form of ideal programs. Ideal programs define the correctness and security requirements we wish to attain by writing a specification assuming the existence of a fully trusted party. We will later prove that our real-world protocols (based on smart contracts) securely emulate the ideal programs.

Overview. With a private ledger specified, we then define the additional Hawk-specific primitives including *freeze*, *compute*, and *finalize* that are essential for enabling transactional privacy and programmability simultaneously.

The formal specification of the ideal program $\mathsf{Ideal}_{\text{hawk}}$ is provided in Figure 7.4. Below, we provide some explanation.

Freeze. In `freeze`, a party tells $\mathsf{Ideal}_{\text{hawk}}$ to remove one coin from the private coins pool Coins, and freeze it in the contract by adding it to $\mathsf{ContractCoins}$. The party's private input denoted in is also recorded in $\mathsf{ContractCoins}$. $\mathsf{Ideal}_{\text{hawk}}$ checks that $\mathcal{P}$ has not called `freeze` earlier, and that a coin $(\mathcal{P}, \mathsf{val})$ exists in Coins before proceeding with the freeze.

Compute. When a party $\mathcal{P}$ calls `compute`, its private input in and the value of its frozen coin val are disclosed to the manager $\mathcal{P}_{\mathcal{M}}$.

Finalize. In `finalize`, the manager $\mathcal{P}_{\mathcal{M}}$ submits a public input $\mathsf{in}_{\mathcal{M}}$ to $\mathsf{Ideal}_{\text{hawk}}$. $\mathsf{Ideal}_{\text{hawk}}$ now computes the outcome of ϕ_{priv} on all parties' inputs and frozen coin values, and redistribute the $\mathsf{ContractCoins}$ based on the outcome of ϕ_{priv}. To ensure money conservation, the ideal program $\mathsf{Ideal}_{\text{hawk}}$ checks that the sum of frozen coins is equal to the sum of output coins.

Interaction with public contract. The $\mathsf{Ideal}_{\text{hawk}}$ functionality is also parameterized by an ordinary public contract ϕ_{pub}, which is included in $\mathsf{Ideal}_{\text{hawk}}$ as a sub-module. During a `finalize`, $\mathsf{Ideal}_{\text{hawk}}$ calls ϕ_{pub}`.check`. The public contract ϕ_{pub} typically serves the following purposes:

- *Check the well-formedness of the manager's input* $\mathsf{in}_{\mathcal{M}}$. For example, in our financial derivatives application (Section 7.5), the public contract ϕ_{pub} asserts that the input corresponds to the price of a stock as reported by the stock exchange's authentic data feed.
- *Redistribute public deposits.* If parties or the manager have aborted, or if a party has provided invalid input (e.g., less than a minimum bet) the public contract ϕ_{pub} can now redistribute the parties' public deposits to ensure financial fairness. For example, in our "Rock, Paper, Scissors" example (see Section 7.5), the private contract ϕ_{priv} checks if each party has frozen the minimal bet. If not, ϕ_{priv} includes that information in out so that ϕ_{pub} pays that party's public deposit to others.

Security and privacy requirements. The $\mathsf{Ideal}_{\text{hawk}}$ program specifies the following privacy guarantees. When an honest party $\mathcal{P}$ freezes money (e.g., a bid), the adversary should not observe the amount frozen. However, the adversary can observe the party's pseudonym $\mathcal{P}$. We note that leaking the pseudonym $\mathcal{P}$ does not hurt privacy, since a party can simply create a new pseudonym $\mathcal{P}$ and pour to this new pseudonym immediately before the freeze.

When an honest party calls `compute`, the manager $\mathcal{P}_{\mathcal{M}}$ gets to observe its input and frozen coin's value. However, the public and other contractual parties do not observe anything (unless the manager voluntarily discloses information).

Finally, during a `finalize` operation, the output out is declassified to the public – note that out can be empty if we do not wish to declassify any information to the public.

It is clear to see that our ideal program $\mathsf{Ideal}_{\text{hawk}}$ satisfies *input independent privacy* and *authenticity* against a dishonest manager. Further, it ensures *posterior privacy* as long as the manager does not voluntarily disclose information. Intuitive explanations of these security/privacy properties were provided in Section 7.5.

Timing and aborts. Our ideal program $\mathsf{Ideal}_{\text{hawk}}$ requires that `freeze` operations are performed by time T_1, and that `compute` operations are performed by time T_2. If a user freezes coins but does not open by time T_2, our ideal program $\mathsf{Ideal}_{\text{hawk}}$ treats $(\mathsf{in}_i, \mathsf{val}_i) := (0, \perp)$, and the user $\mathcal{P}_i$ essentially forfeits its frozen coins. Managerial aborts are not handled inside $\mathsf{Ideal}_{\text{hawk}}$, but by the public portion of the contract ϕ_{pub}.

Simplifying assumptions. For clarity, our basic version of $\mathsf{Ideal}_{\text{hawk}}$ is a stripped down version of our implementation. Specifically, our basic $\mathsf{Ideal}_{\text{hawk}}$ and protocols do not realize refunds of frozen coins upon managerial abort. It is straightforward to accommodate this by adding this behavior to the transparent contract. We also assume that the set of pseudonyms participating in the contract as well as timeouts T_1 and T_2 are hard-coded in the program, while in reality these could easily be chosen dynamically.

7.4.1 Protocol

The protocol is defined in Figure 7.5. We give an explanation of the new activation points.

Freeze. We support a new operation called `freeze`, that does not spend directly to a user, but commits the money as well as an accompanying private input to a smart contract. This is done using a `pour`-like protocol:

- The user $\mathcal{P}$ chooses a private coin $(\mathcal{P}, \mathsf{coin}) \in \mathsf{Coins}$, where $\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$. Using its secret key, $\mathcal{P}$ computes the serial number sn for coin – to be disclosed with the `freeze` operation to prevent double-spending.
- The user $\mathcal{P}$ computes a commitment $(\mathsf{val}||\mathsf{in}||k)$ to the contract where in denotes its input, and k is a symmetric encryption key that is introduced due to a practical optimization explained later in Section 7.4.4.
- The user $\mathcal{P}$ now makes a zero-knowledge proof attesting to similar statements as in a `pour` operation, i.e., that the spent coin exists in the pool Coins, the sn is correctly constructed, and that the val committed to the contract equals the value of the coin being spent. See $\mathcal{L}_{\texttt{FREEZE}}$ in Figure 7.5 for details of the NP statement being proven.

Compute. Next, computation takes place off-chain to compute the payout distribution $\{\mathsf{val}'_i\}_{i\in[N]}$ and a proof of correctness. In Hawk, we rely on a minimally trusted manager $\mathcal{P}_{\mathcal{M}}$ to perform computation. All parties would open their inputs to the manager $\mathcal{P}_{\mathcal{M}}$, and this is done by

encrypting the opening to the manager's public key:

$$\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val} \| \mathsf{in} \| k \| s'))$$

The ciphertext ct is submitted to the smart contract along with appropriate zero-knowledge proofs of correctness. While the user can also directly send the opening to the manager off-chain, passing the ciphertext ct through the smart contract would make any aborts evident such that the contract can financially punish an aborting user. Alternatively, it is also possible to perform the opening off-chain *optimistically*, and allowing the manager to submit a dispute to the contract in the event of a user abort. In this case, the blamed user must submit the opening on-chain within a timeout.

After obtaining the openings, the manager now computes the payout distribution $\{\mathsf{val}'_i\}_{i \in [N]}$ and public output out by applying the private contract ϕ_{priv}. The manager also constructs a zero-knowledge proof attesting to the outcomes.

Finalize. When the manager submits the outcome of ϕ_{priv} and a zero-knowledge proof of correctness to $\mathsf{ContractHawk}$, $\mathsf{ContractHawk}$ verifies the proof and redistributes the frozen money accordingly. Here $\mathsf{ContractHawk}$ also passes the manager's public input $\mathsf{in}_{\mathcal{M}}$ and public output out to a public contract denoted C. The public contract C can be invoked to check the validity of the manager's input, as well as redistribute public collateral deposit.

Subtleties related to the NIZK proofs. Some subtle technicalities arise when we use SNARKs to instantiate NIZK proofs. As mentioned in Theorem 7.2, we assume that our NIZK scheme satisfies simulation sound extractability. Unfortunately, ordinary SNARKs do not offer simulation sound extractability – and our simulator cannot simply use the SNARK's extractor since the SNARK extractor is non-blackbox, and using the SNARK extractor in a UC-style simulation proof would require running the environment in the extractor!

We therefore rely a generic transformation [50] to build a simulation sound extractable NIZK from an ordinary SNARK scheme.

We now prove our main result, Theorem 7.2 (see Section 7.3). Just as we did for private cash in Theorem 7.1, we will construct an ideal-world simulator $\mathcal{S}$ for every real-world adversary $\mathcal{A}$, such that no polynomial-time environment $\mathcal{E}$ can distinguish whether it is in the real or ideal world.

7.4.2 Ideal World Simulator

Our ideal program ($\mathsf{Ideal}_{\mathrm{hawk}}$) and construction ($\mathsf{ContractHawk}$ and $\Pi_{\mathtt{HAWK}}$) borrows from our private cash definition and construction in a non-blackbox way (i.e., by duplicating the relevant behaviors). As such, our simulator program simP also duplicates the behavior of the simulator from Section 7.2.3.2 involving `mint` and `pour` interactions. Hence we will here explain the behavior involving the additional `freeze`, `compute`, and `finalize` interactions.

Init. Same as in Section 7.2.3.

Simulating corrupted parties. The following messages are sent by the environment $\mathcal{E}$ to the simulator $\mathcal{S}(\mathsf{simP})$ which then forwards it on to both the internally simulated contract $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{ContractHawk}))$ and the inner simulator simP.

- Corrupt party $\mathcal{P}$ submits a transaction $\mathtt{freeze}(\pi, \mathsf{sn}, \mathsf{cm})$ to the contract. The simulator forwards this transaction to the contract, but also uses the trapdoor τ to extract a witness from π, including $\$\mathsf{val}$ and in. The simulator then sends $\mathtt{freeze}(\$\mathsf{val}, \mathsf{in})$ to $\mathcal{F}_{\text{HAWK}}$.
- Corrupt party $\mathcal{P}$ sumbits a transaction $\mathtt{compute}(\pi, \mathsf{ct})$ to the contract. The simulator forwards this to the contract and sends $\mathtt{compute}$ to $\mathcal{F}_{\text{HAWK}}$. The simulator also uses τ to extract a witness from π, including k_i, which is used later. These is stored as $\mathsf{CorruptOpen}_i := k_i$.
- Corrupt party $\mathcal{P}_{\mathcal{M}}$ submits a transaction $\mathtt{finalize}(\pi, \mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\})$. The simulator forwards this to the contract, and simply sends $\mathtt{finalize}(\mathsf{in}_{\mathcal{M}})$ to $\mathcal{F}_{\text{HAWK}}$.

Simulating honest parties. When the environment $\mathcal{E}$ sends inputs to honest parties, the simulator $\mathcal{S}$ needs to simulate messages that corrupted parties receive, from honest parties or from functionalities in the real world. The honest parties will be simulated as below:

- Environment $\mathcal{E}$ gives a $\mathtt{freeze}$ instruction to party $\mathcal{P}$. The simulator simP receives $\mathtt{freeze}(\mathcal{P})$ from $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{hawk}})$. The simulator does not have any information about the actual committed values for $\$\mathsf{val}$ or in. Instead, the simulator create a bogus commitment
  $\mathsf{cm} := \mathsf{Comm}_s(0\|\bot\|\bot)$ that will later be opened (via a false proof) to an arbitrary value. To generate the serial number sn, the simulator chooses a random element from the codomain of PRF. Finally, the simulator uses τ to generate a forged proof π and sends $\mathtt{freeze}(\pi, \mathsf{sn}, \mathsf{cm})$ to the contract.
- Environment $\mathcal{E}$ gives a $\mathtt{compute}$ instruction to party $\mathcal{P}$. The simulator simP receives $\mathtt{compute}(\mathcal{P})$ from $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{hawk}})$. The simulator behaves differently depending on whether or not the manager $\mathcal{P}_{\mathcal{M}}$ is corrupted.

Case 1: $\mathcal{P}_{\mathcal{M}}$ *is honest.* The simulator does not know values $\$val$ or in. Instead, the simulator samples an encryption randomness r and generates an encryption of 0, $\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, 0)$. Finally, the simulator uses the trapdoor τ to create a false proof π that the commitment cm and ciphertext ct are consistent. The simulator then passes $\mathtt{compute}(\pi, \mathsf{ct})$ to the contract.

Case 2: $\mathcal{P}_{\mathcal{M}}$ *is corrupted.* Since the manager $\mathcal{P}_{\mathcal{M}}$ in the ideal world would learn $\$\mathsf{val}$, in, and k at this point, the simulator learns these values instead. Hence it samples an encryption randomness r and computes a valid encryption $\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_{\mathcal{M}}.\mathsf{epk}, r, (\$\mathsf{val}\|\mathsf{in}\|k))$. The simulator next uses τ to create a proof π attesting that ct is consistent with cm. Finally, the simulator sends $\mathtt{compute}(\pi, \mathsf{ct})$ to the contract.

- Environment $\mathcal{E}$ gives a `finalize` instruction to party $\mathcal{P}_{\mathcal{M}}$. The simulator simP receives $\texttt{finalize}(\mathsf{in}_{\mathcal{M}}, \mathsf{out})$ from $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{hawk}})$. The simulator generates the output coin'_i for each party $\mathcal{P}_i$ depending on whether $\mathcal{P}_i$ is corrupted or not:
  - $\mathcal{P}_i$ *is honest*: The simulator does not know the correct output value for $\mathcal{P}_i$, so instead creates a bogus commitment $\mathsf{coin}'_i := \mathsf{Comm}_{s'_i}(0)$ and a bogus ciphertext $\mathsf{ct}'_i := \mathsf{SENC}_{k_i}(s'_i \| 0)$ for sampled randomnesses k_i and s'_i.
  - $\mathcal{P}_i$ *is corrupted*: Since the ideal world recipient would receive $\$\mathsf{val}'_i$ from $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{Ideal}_{\text{hawk}})$, the simulator learns the correct value $\$\mathsf{val}'_i$ directly. Notice that since $\mathcal{P}_i$ was corrupted, the simulator has access to $k_i := \mathsf{CorruptOpen}_i$, which it extracted earlier. The simulator therefore draws a randomness s'_i, and computes $\mathsf{coin}'_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$ and $\mathsf{ct}_i := \mathsf{SENC}_{k_i}(s'_i \| \$\mathsf{val}'_i)$.

  The simulator finally constructs a forged proof π using the trapdoor τ, and then passes $\texttt{finalize}(\pi, \mathsf{in}_{\mathcal{M}}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\}_{i \in [N]})$ to the contract.

7.4.3 Indistinguishability of Real and Ideal Worlds

To prove indistinguishability of the real and ideal worlds from the perspective of the environment, we will go through a sequence of hybrid games.

Real world. We start with the real world with a dummy adversary that simply passes messages to and from the environment $\mathcal{E}$.

Hybrid 1. Hybrid 1 is the same as the real world, except that now the adversary (also referred to as the simulator) will call $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \mathsf{NIZK}.\widehat{\mathcal{K}}(1^\lambda)$ to perform a simulated setup for the NIZK scheme. The simulator will pass the simulated $\widehat{\mathsf{crs}}$ to the environment $\mathcal{E}$. When an honest party $\mathcal{P}$ publishes a NIZK proof, the simulator will replace the real proof with a simulated NIZK proof before passing it onto the environment $\mathcal{E}$. The simulated NIZK proof can be computed by calling the $\mathsf{NIZK}.\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \cdot)$ algorithm which takes only the statement as input but does not require knowledge of a witness.

Fact 7. It is immediately clear that if the NIZK scheme is computational zero-knowledge, then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 1 from the real world except with negligible probability.

Hybrid 2. The simulator simulates the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{ContractHawk}))$ functionality. Since all messages to the $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{ContractHawk}))$ functionality are public, simulating the contract functionality is trivial. Therefore, Hybrid 2 is identically distributed as Hybrid 1 from the environment $\mathcal{E}$'s view.

Hybrid 3. Hybrid 3 is the same as Hybrid 2 except the following changes. When an honest party sends a message to the contract (now simulated by the simulator $\mathcal{S}$), it will sign the message with a signature verifiable under an honestly generated nym. In Hybrid 3, the simulator will replace all honest parties' nyms and generate these nyms itself. In this way, the simulator will simulate honest parties' signatures by signing them itself. Hybrid 3 is identically distributed to Hybrid 2 from the environment $\mathcal{E}$'s view.

Hybrid 4. Hybrid 4 is the same as Hybrid 3 except for the following changes:

- When an honest party $\mathcal{P}$ produces a ciphertext ct_i for a recipient $\mathcal{P}_i$, and if the recipient is also uncorrupted, then the simulator will replace this ciphertext with an encryption of 0 before passing it onto the environment $\mathcal{E}$.
- When an honest party $\mathcal{P}$ produces a commitment coin or cm, then the simulator replaces this commitment with a commitment to 0.
- When an honest party $\mathcal{P}$ computes a pseudorandom serial number sn, the simulator replaces this with a randomly chosen value from the codomain of PRF.

Fact 8. It is immediately clear that if the encryption scheme is semantically secure, if PRF is a pseudorandom function, and if Comm is a perfectly hiding commitment scheme, then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 4 from Hybrid 3 except with negligible probability.

Hybrid 5. Hybrid 5 is the same as Hybrid 4 except for the following changes. Whenever the environment $\mathcal{E}$ passes to the simulator $\mathcal{S}$ a message signed on behalf of an honest party's nym, if the message and signature pair was not among the ones previously passed to the environment $\mathcal{E}$, then the simulator $\mathcal{S}$ aborts.

Fact 9. Assume that the signature scheme employed is secure; then the probability of aborting in Hybrid 5 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 5 would otherwise be identically distributed as Hybrid 4 modulo aborting.

Hybrid 6. Hybrid 6 is the same as Hybrid 5 except for the following changes. Whenever the environment passes $\mathtt{pour}(\pi, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i, \mathsf{ct}_i\})$ (or $\mathtt{freeze}(\pi, \mathsf{sn}, \mathsf{cm})$) to the simulator (on behalf of corrupted party $\mathcal{P}$), if the proof π verifies under $\mathsf{statement}$, then the simulator will call the NIZK's extractor algorithm $\mathcal{E}$ to extract $\mathsf{witness}$. If the NIZK π verifies but the extracted $\mathsf{witness}$ does not satisfy the relation $\mathcal{L}_{\mathtt{POUR}}(\mathsf{statement}, \mathsf{witness})$ (or $\mathcal{L}_{\mathtt{FREEZE}}(\mathsf{statement}, \mathsf{witness})$), then abort the simulation.

Fact 10. Assume that the NIZK is simulation sound extractable, then the probability of aborting in Hybrid 6 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 6 would otherwise be identically distributed as Hybrid 5 modulo aborting.

Finally, observe that Hybrid 6 is computationally indistinguishable from the ideal simulation $\mathcal{S}$ unless one of the following bad events happens:

- A value val' decrypted by an honest recipient is different from that extracted by the simulator. However, given that the encryption scheme is perfectly correct, this cannot happen.
- A commitment coin is different than any stored in $\mathsf{ContractHawk.coins}$, yet it is valid according to the relation $\mathcal{L}_{\mathtt{POUR}}$. Given that the Merkle tree MT is computed using collision-resistant a hash function, this occurs with at most negligible probability.
- The honest public key generation algorithm results in key collisions. Obviously, this happens with negligible probability if the encryption and signature schemes are secure.

Fact 11. Given that the encryption scheme is semantically secure and perfectly correct, and that the signature scheme is secure, then Hybrid 6 is computationally indistinguishable from the ideal simulation to any polynomial-time environment $\mathcal{E}$.

Theorem 7.2. *Assuming that the hash function in the Merkle tree is collision resistant, the commitment scheme* Comm *is perfectly binding and computationally hiding, the* NIZK *scheme is computationally zero-knowledge and simulation sound extractable, the encryption schemes* ENC *and* SENC *are perfectly correct and semantically secure, the PRF scheme* PRF *is secure, then, our protocols in Figures 7.2 and 7.5 securely emulates the ideal functionality* $\mathcal{F}(\mathsf{Ideal}_{hawk})$.

7.4.4 Practical Considerations

Our scheme's main performance bottleneck is computing NIZK proofs. Specifically, NIZK proofs must be computed whenever *i)* a user invokes a `pour`, `freeze`, or a `compute` operation; and *ii)* the manager calls `finalize`. In an implementation of Hawk, Kosba et al. [8] use a state-of-the-art simulation extractable NIZKconstructions based on zkSNARKs [47, 50] as explained in Section 2.2.3.

Efficient SNARK circuits. A SNARK prover's performance is mainly determined by the number of multiplication gates in the algebraic circuit to be proven [46, 47]. To achieve efficiency, we use circuits that have been carefully optimmized in two main ways. First, we use cryptographic primitives that are SNARK-friendly, i.e. efficiently realizable as arithmetic circuits under a specific SNARK parametrization. For example, we use a SNARK-friendly collision-resistant hash function [50, 99] to realize the Merkle tree circuit. Second, we build customized circuits directly instead of relying on compilers from generic high-level languages (i.e., we avoid relying on the Pinocchio C compiler [46] for implementing cryptography). For example, our circuits rely also on standard SHA-256, RSA-OAEP encryption, and RSA signature verification (with 2048-bit keys for both), so we use the hand-optimized these components due to Kosba et al. [50] to reduce circuit size.

An optimization using symmetric encryption. We now briefly explain a performance optimization incorporated into our protocol. Since producing the NIZK proofs is the main performance bottleneck in our prototypes, we focus on minimizing the circuit sizes for the NIZK proofs. In particular, we focus on minimizing the cost of the $O(N)$-sized `finalize` circuit by

trading off expense in the other proofs in our scheme (for `freeze` and `pour`), which are only $O(1)$-sized. During `finalize`, the manager encrypts each party $\mathcal{P}_i$'s output coins to $\mathcal{P}_i$'s key, resulting in a ciphertext ct_i. The ciphertexts $\{\mathsf{ct}_i\}_{i \in [N]}$ would then be submitted to the contract along with appropriate SNARK proofs of correctness. Here, if a public-key encryption is employed to generate the ct_i's, it would result in a relatively large SNARK circuit size. Instead, we rely on a symmetric-key encryption scheme denoted SENC in Figure 7.5. This requires that the manager and each $\mathcal{P}_i$ perform a key exchange to establish a symmetric key k_i. During an `compute`, the user encrypts this k_i to the manager's public key $\mathcal{P}_{\mathcal{M}}.\mathsf{epk}$, and prove that the k encrypted is consistent with the k committed to earlier in cm_i. The SNARK proof during `finalize` now only needs to include commitments and symmetric encryptions instead of public key encryptions in the circuit – the latter much more expensive.

7.5 Hawk Implementation and Examples

7.5.1 Protocol Compiler

Our Hawk compiler consists of several steps, as explained below:

1. *Preprocessing:* First, the input Hawk program is split into its *public contract* and *private contract* components. The public contract is Serpent code, and can be executed directly atop an ordinary cryptocurrency platform such as Ethereum. The private contract is written in a subset of the C language, and is passed as input to the Pinocchio arithmetic circuit compiler [46]. Keywords such as `HawkDeclareParties` are implemented as C preprocessor macros, and serve to define the input (`Inp`) and output (`Outp`) datatypes. Currently, our private contract inherits the limitations of the Pinocchio compiler, e.g., cannot support dynamic-length loops. In the future, we can relax these limitations by employing recursive composition of SNARKs.
2. *Circuit Augmentation:* After compiling the preprocessed private contract code with Pinocchio, we have an arithmetic circuit representing the input/output relation ϕ_{priv}. This becomes a subcomponent of a larger arithmetic circuit, which we assemble using a customized circuit assembly tool (described in [8, 50]). This tool is parameterized by the number of parties and the input/output datatypes, and attaches cryptographic constraints, such as computing commitments and encryptions over each party's output value, and asserting that the input and output values satisfy the balance property.
3. *Cryptographic Protocol:* Finally, the augmented arithmetic circuit is used as input to a state-of-the-art zkSNARK library, `libsnark` [47]. We finally compile an executable program for the parties to compute the libsnark proofs according to our protocol.

7.5.2 Example Application: Sealed Bid Auction

We illustrate our protocol and the role of our compiler by explaining an example application that implements a privacy-preserving sealed bid auction.

Example program. Figure 7.6 shows a Hawk program for implementing a sealed, second-price auction where the highest bidder wins, but pays the second highest price. Second-price auctions are known to incentivize truthful bidding under certain assumptions, [100] and it is important that bidders submit bids without knowing the bid of the other people. Our example auction program contains a private contract ϕ_{priv} that determines the winning bidder and the price to be paid; and a public contract ϕ_{pub} that relies on public deposits to protect bidders from an aborting manager. Furthermore, even after the auction, only the manager learns the bids of the parties. The payment from the winning bidder to the seller, as well as the refunds to all the losing bidders, are all executed as private money transfers; this is in contrast to prior schemes such as [101] that must publicly reveal the bids at the end.

Contractual security requirements. Hawk will compile this auction program to a cryptographic protocol. As mentioned earlier, as long as the bidders and the manager do not voluntarily disclose information, transaction privacy is maintained against the public. Hawk also guarantees the following contractual security requirements for parties in the contract:

- *Input independent privacy.* Each user does not see others' bids before committing to their own. This way, users bids are independent of others' bids. Hawk guarantees input independent privacy even against a malicious manager.
- *Posterior privacy.* As long as the manager does not disclose information, users' bids are kept private from each other (and from the public) even after the auction.
- *Financial fairness.* If a party aborts or if the auction manager aborts, the aborting party should be financially penalized while the remaining parties receive compensation. Such fairness guarantees are not attainable in general by off-chain only protocols such as secure multi-party computation [22, 41]. As explained later, Hawk offers built-in mechanisms for enforcing refunds of private bids after certain timeouts.
  Hawk also allows the programmer to define additional rules, as part of the Hawk contract, that govern financial fairness.
- *Security against a dishonest manager.* We ensure *authenticity* against a dishonest manager: besides aborting, a dishonest manager cannot affect the outcome of the auction and the redistribution of money, even when it colludes with a subset of the users. We stress that to ensure the above, input independent privacy against a faulty manager is a prerequisite. Moreover, if the manager aborts, it can be financially penalized, and the participants obtain corresponding remuneration.

An auction with the above security and privacy requirements cannot be trivially implemented atop existing cryptocurrency systems such as Ethereum [28] or Zerocash [36]. The former allows for programmability but does not guarantee transactional privacy, while the latter guarantees transactional privacy but at the price of even reduced programmability than Bitcoin.

Aborting and timeouts. Aborting are dealt with using timeouts. A Hawk program such as Figure 7.6 declares timeout parameters using the `HawkDeclareTimeouts` special syntax. Three timeouts are declared where $T_1 < T_2 < T_3$:

T_1 : The Hawk contract stops collecting bids after T_1.

T_2 : All users should have opened their bids to the manager within T_2; if a user submitted a bid but fails to open by T_2, its input bid is treated as 0 (and any other potential input data treated as $\perp$), such that the manager can continue.

T_3 : If the manager aborts, users can reclaim their private bids after time T_3.

7.5.3 Other Examples

We provide the Hawk programs for several other example applications.

Crowdfunding: (Figure 7.8 A Kickstarter-style crowdfunding campaign, (also known as an assurance contract in economics literature [102]) overcomes the "free-rider problem," allowing a large number of parties to contribute funds towards some social good. If the minimum donation target is reached before the deadline, then the donations are transferred to a designated party (the entrepreneur); otherwise, the donations are refunded. Hawk preserves privacy in the following sense: a) the donations pledged are kept private until the deadline; and b) if the contract fails, only the manager learns the amount by which the donations were insufficient. These privacy properties may conceivably have a positive effect on the willingness of entrepreneurs to launch a crowdfund campaign and its likelihood of success.

Rock Paper Scissors: (Figure 7.7) A two-player lottery game, and naturally generalized to an N-player version. Our Hawk implementation provides the same notion of financial fairness as in [22, 41] and provides stronger security/privacy guarantees. If any party (including the manager), cheats or aborts, the remaining honest parties receive the maximum amount they might have won otherwise. Furthermore, we go beyond prior works [22, 41] by concealing the players' moves and the pseudonym of the winner to everyone except the manager.

"Swap" Financial Instrument: (Figure 7.9 An individual with a risky investment portfolio (e.g, one who owns a large number of Bitcoins) may hedge his risks by purchasing insurance (e.g., by effectively betting against the price of Bitcoin with another individual). Our example implements a simple swap instrument where the price of a stock at some future date (as reported by a trusted authority specified in the public contract) determines which of two parties receives a payout. The private contract ensures the privacy of both the details of the agreement (i.e., the price threshold) and the outcome.

Crowdfunding example. In the crowdfunding example in Figure 7.8, parties donate money for a kickstarter project. If the total raised funding exceeds a pre-set budget denoted `BUDGET`, then the campaign is successful and the kickstarter obtains the total donations. Otherwise, all donations are returned to the donors after a timeout. In this case, no public deposit is necessary to ensure the incentive compatibility of the contract. If a party does not open after freezing its money, the money is unrecoverable by anyone.

Swap instrument example. In this financial swap instrument, Alice is betting on the stock price exceeding a certain threshold at a future point of time, while Bob is betting on the reverse.

If the stock price is below the threshold, Alice obtains \$20; else Bob obtains \$20. As mentioned earlier in Section 7.5, such a financial swap can be used as a means of insurance to hedge invenstment risks. This swap contract makes use of public deposits to provide financial fairness when either Alice or Bob cheats.

This swap assumes that the manager is a well-known public entity such as a stock exchange. Therefore, the contract does not protect against the manager aborting. In the event that the manager aborts, the aborting event can be observed in public, and therefore external mechanisms (e.g., legal enforcement or reputation) can be leveraged to punish the manager.

ContractCash

Init: crs: a reference string for the underlying NIZK system
Coins: a set of coin commitments, initially $\emptyset$
SpentCoins: set of spent serial numbers, initially $\emptyset$

Mint: Upon receiving $\mathtt{mint}(\$\mathsf{val}, s)$ from some party $\mathcal{P}$,
- $\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$
- assert $(\mathcal{P}, \mathsf{coin}) \notin \mathsf{Coins}$
- assert $\mathsf{ledger}[\mathcal{P}] \geq \$\mathsf{val}$
- $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] - \$\mathsf{val}$
- add $(\mathcal{P}, \mathsf{coin})$ to Coins

Pour: Anonymous receive $\mathtt{pour}(\pi, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i, \mathsf{ct}_i\}_{i\in\{1,2\}})$
- let MT be a merkle tree built over Coins
- $\mathsf{statement} := (\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}_i\}_{i\in\{1,2\}})$
- assert $\mathsf{NIZK.Verify}(\mathcal{L}_{\mathtt{POUR}}, \pi, \mathsf{statement})$
- for $i \in \{1, 2\}$,
  - assert $\mathsf{sn}_i \notin \mathsf{SpentCoins}$
  - assert $(\mathcal{P}_i, \mathsf{coin}_i) \notin \mathsf{Coins}$
  - add sn_i to SpentCoins
  - add $(\mathcal{P}, \mathsf{coin}_i)$ to Coins
  - send $\mathtt{pour}(\mathsf{coin}_i, \mathsf{ct}_i)$ to $\mathcal{P}_i$

Relation $(\mathsf{stmt}, \mathsf{wit}) \in \mathcal{L}_{\mathtt{POUR}}$ is defined as:
parse stmt as $(\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}'_i\}_{i\in\{1,2\}})$
parse wit as $(\mathcal{P}, \mathsf{sk}_{\mathtt{prf}}, \{\mathsf{branch}_i, s_i, \$\mathsf{val}_i, s'_i, r_i, \$\mathsf{val}'_i\})$
assert $\mathcal{P}.\mathsf{pk}_{\mathtt{prf}} = \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(0)$
assert $\$\mathsf{val}_1 + \$\mathsf{val}_2 = \$\mathsf{val}'_1 + \$\mathsf{val}'_2$
for $i \in \{1, 2\}$,
- $\mathsf{coin}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i)$
- assert $\mathsf{MerkleBranch}(\mathsf{MT.root}, \mathsf{branch}_i, (\mathcal{P}\|\mathsf{coin}_i))$
- assert $\mathsf{sn}_i = \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P}\|\mathsf{coin}_i)$
- assert $\mathsf{coin}'_i = \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$

Protocol ProtCash

Init: Wallet: stores $\mathcal{P}$'s spendable coins, initially $\emptyset$

GenNym:
- create encryption keys $(\mathsf{epk}, \mathsf{esk}) \leftarrow \mathsf{Keygen}_{\mathrm{enc}}(1^\lambda)$.
- sample a random seed $\mathsf{sk}_{\mathtt{prf}}$
- $\mathsf{pk}_{\mathtt{prf}} := \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(0)$
- store $\mathsf{sk}_{\mathtt{prf}}$ and esk for later use
- return $(\mathsf{pk}_{\mathtt{prf}}, \mathsf{epk})$

Mint: On input $\mathtt{mint}(\$\mathsf{val})$,
- sample a commitment randomness s
- $\mathsf{coin} := \mathsf{Comm}_s(\$\mathsf{val})$
- store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
- send $\mathtt{mint}(\$\mathsf{val}, s)$ to ContractCash

Pour (as sender): On input $\mathtt{pour}(\$\mathsf{val}_1, \$\mathsf{val}_2, \mathcal{P}_1, \mathcal{P}_2, \$\mathsf{val}'_1, \$\mathsf{val}'_2)$,
- assert $\$\mathsf{val}_1 + \$\mathsf{val}_2 = \$\mathsf{val}'_1 + \$\mathsf{val}'_2$
- for $i \in \{1, 2\}$, assert $(s_i, \$\mathsf{val}_i, \mathsf{coin}_i) \in \mathsf{Wallet}$ for some (s_i, coin_i)
- let MT be a merkle tree over ContractCash.Coins
- for $i \in \{1, 2\}$:
  - remove one $(s_i, \$\mathsf{val}_i, \mathsf{coin}_i)$ from Wallet
  - $\mathsf{sn}_i := \mathsf{PRF}_{\mathsf{sk}_{\mathtt{prf}}}(\mathcal{P}\|\mathsf{coin}_i)$
  - let branch_i be the branch of $(\mathcal{P}, \mathsf{coin}_i)$ in MT
  - sample randomness s'_i, r_i
  - $\mathsf{coin}'_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$
  - $\mathsf{ct}_i := \mathsf{ENC}(\mathcal{P}_i.\mathsf{epk}, r_i, \$\mathsf{val}'_i\|s'_i)$
- $\mathsf{statement} := (\mathsf{MT.root}, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}'_i\}_{i\in\{1,2\}})$
- $\mathsf{witness} := (\mathcal{P}, \mathsf{sk}_{\mathtt{prf}}, \{\mathsf{branch}_i, s_i, \$\mathsf{val}_i, s'_i, r_i, \$\mathsf{val}'_i\})$
- $\pi := \mathsf{NIZK.Prove}(\mathcal{L}_{\mathtt{POUR}}, \mathsf{statement}, \mathsf{witness})$
- $\mathsf{AnonSend}(\mathtt{pour}(\pi, \{\mathsf{sn}_i, \mathcal{P}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in\{1,2\}}))$ to ContractCash

Pour (as recipient): On receive $\mathtt{pour}(\mathsf{coin}, \mathsf{ct})$ from ContractCash):
- let $(\$\mathsf{val}\|s) := \mathsf{DEC}(\mathsf{sk}_{\mathsf{enc}}, \mathsf{ct})$
- assert $\mathsf{Comm}_s(\$\mathsf{val}) = \mathsf{coin}$
- store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
- output $\mathtt{pour}(\$\mathsf{val})$

FIGURE 7.2: ProtCash construction. A trusted setup phase generates the NIZK's common reference string crs. For notational convenience, we omit writing the crs explicitly in the construction. The Merkle tree MT is stored in the contract and not computed on the fly – we omit stating this in the protocol for notational simplicity.

$\mathcal{S}(\mathsf{simP})$

Init. The simulator $\mathcal{S}$ simulates a $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{contract})$ instance internally. Here $\mathcal{S}$ calls $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathsf{contract})$.**Init** to initialize the internal states of the transparent contract functionality.
The simulator simP runs $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \mathsf{NIZK}.\widehat{\mathcal{K}}(1^\lambda)$, and gives $\widehat{\mathsf{crs}}$ to the environment $\mathcal{E}$.

Simulating honest parties.

- **GenNym:** Environment $\mathcal{E}$ sends input `gennym` to an honest party $\mathcal{P}$: simulator $\mathcal{S}$ receives notification `gennym` from the ideal functionality. Simulator $\mathcal{S}$ honestly generates an encryption key $(\mathsf{epk}, \mathsf{esk}) := \mathsf{ENC}.\mathcal{K}(1^\lambda)$. and remembers the corresponding secret keys. The wrapper generates and records the PRF keypair, $(\mathsf{pk}_{\mathsf{PRF}}, \mathsf{sk}_{\mathsf{PRF}})$ and returns $\mathsf{payload} := (\mathsf{epk}, \mathsf{pk}_{\mathsf{PRF}})$.
- **Ledger Operations.** If ideal functionality sends `transfer`($\$\mathsf{val}$, P_r, P_s), then update the ledger in the simulated $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{Contract}))$ instance accordingly.
- **Other activations.** Other messages are forwarded to the inner simulator program simP.

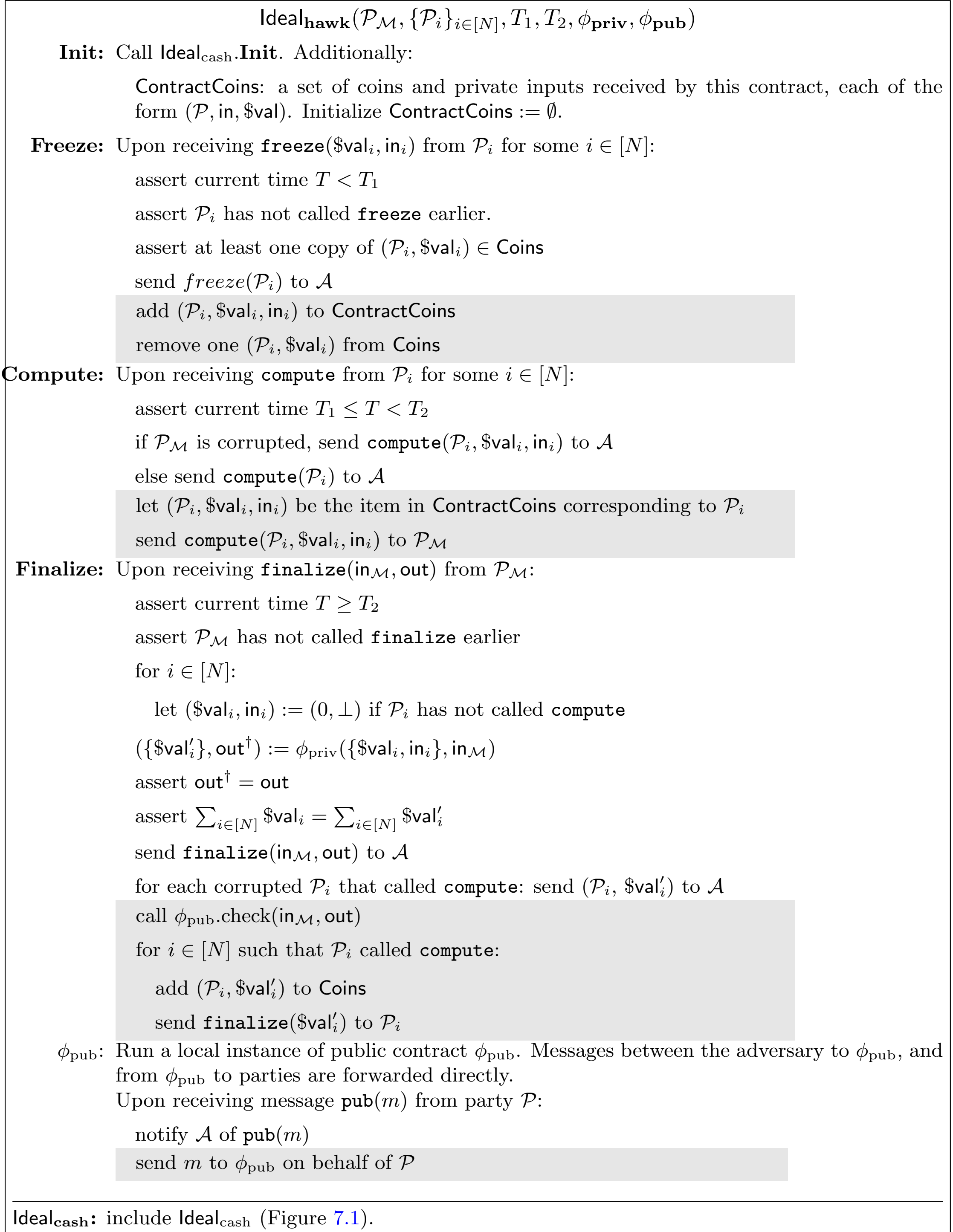
Simulating corrupted parties.

- **Expose.** Upon receiving `exposestate` from the environment $\mathcal{E}$, expose all states of the internally simulated $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{Contract}))$.
- **Pseudonymous send.** Upon receiving `pseudonymous`$(m, \bar{P})$ from the environment $\mathcal{E}$ on behalf of corrupted party P: Forward to internally simulated $\mathcal{F}_{\text{BLOCKCHAIN}}(\mathcal{T}(\mathsf{contract}))$. If the message is of the format `transfer`, $\$\mathsf{val}, P_r, P_s)$, then pass this to the ideal functionality. Otherwise, forward this to simP.
- **Anonymous send.** Upon receiving `anonymous`(m) from the environment $\mathcal{E}$ on behalf of corrupted party P: Forward to internally simulated $\mathcal{F}_{\text{BLOCKCHAIN}}$, and forward to simP.
- **Instructions to the dummy adversary.**

  Deliver. Upon receiving `deliver`(i) from the environment $\mathcal{E}$, pass $\mathsf{deliver}(i)$ to the internally running instance of $\mathcal{F}_{\text{BLOCKCHAIN}}$.

  Get Leaks. Upon receiving `getleaks` from the environment $\mathcal{E}$, return the contents of leaks in the internally running instance of $\mathcal{F}_{\text{BLOCKCHAIN}}$.

FIGURE 7.3: Simulator wrapper.



$\mathsf{Ideal}_{\mathbf{hawk}}(\mathcal{P}_{\mathcal{M}}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\mathbf{priv}}, \phi_{\mathbf{pub}})$

Init: Call $\mathsf{Ideal}_{\text{cash}}$.**Init**. Additionally:

$\mathsf{ContractCoins}$: a set of coins and private inputs received by this contract, each of the form $(\mathcal{P}, \mathsf{in}, \$\mathsf{val})$. Initialize $\mathsf{ContractCoins} := \emptyset$.

Freeze: Upon receiving $\texttt{freeze}(\$\mathsf{val}_i, \mathsf{in}_i)$ from $\mathcal{P}_i$ for some $i \in [N]$:

assert current time $T < T_1$

assert $\mathcal{P}_i$ has not called `freeze` earlier.

assert at least one copy of $(\mathcal{P}_i, \$\mathsf{val}_i) \in \mathsf{Coins}$

send $freeze(\mathcal{P}_i)$ to $\mathcal{A}$

add $(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ to $\mathsf{ContractCoins}$

remove one $(\mathcal{P}_i, \$\mathsf{val}_i)$ from Coins

Compute: Upon receiving `compute` from $\mathcal{P}_i$ for some $i \in [N]$:

assert current time $T_1 \leq T < T_2$

if $\mathcal{P}_{\mathcal{M}}$ is corrupted, send $\texttt{compute}(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ to $\mathcal{A}$

else send $\texttt{compute}(\mathcal{P}_i)$ to $\mathcal{A}$

let $(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ be the item in $\mathsf{ContractCoins}$ corresponding to $\mathcal{P}_i$

send $\texttt{compute}(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$ to $\mathcal{P}_{\mathcal{M}}$

Finalize: Upon receiving $\texttt{finalize}(\mathsf{in}_{\mathcal{M}}, \mathsf{out})$ from $\mathcal{P}_{\mathcal{M}}$:

assert current time $T \geq T_2$

assert $\mathcal{P}_{\mathcal{M}}$ has not called `finalize` earlier

for $i \in [N]$:

  let $(\$\mathsf{val}_i, \mathsf{in}_i) := (0, \perp)$ if $\mathcal{P}_i$ has not called `compute`

$(\{\$\mathsf{val}'_i\}, \mathsf{out}^\dagger) := \phi_{\text{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}, \mathsf{in}_{\mathcal{M}})$

assert $\mathsf{out}^\dagger = \mathsf{out}$

assert $\sum_{i\in[N]} \$\mathsf{val}_i = \sum_{i\in[N]} \$\mathsf{val}'_i$

send $\texttt{finalize}(\mathsf{in}_{\mathcal{M}}, \mathsf{out})$ to $\mathcal{A}$

for each corrupted $\mathcal{P}_i$ that called `compute`: send $(\mathcal{P}_i, \$\mathsf{val}'_i)$ to $\mathcal{A}$

call $\phi_{\text{pub}}.\text{check}(\mathsf{in}_{\mathcal{M}}, \mathsf{out})$

for $i \in [N]$ such that $\mathcal{P}_i$ called `compute`:

  add $(\mathcal{P}_i, \$\mathsf{val}'_i)$ to Coins

  send $\texttt{finalize}(\$\mathsf{val}'_i)$ to $\mathcal{P}_i$

ϕ_{pub}: Run a local instance of public contract ϕ_{pub}. Messages between the adversary to ϕ_{pub}, and from ϕ_{pub} to parties are forwarded directly.
Upon receiving message $\mathsf{pub}(m)$ from party $\mathcal{P}$:

notify $\mathcal{A}$ of $\mathsf{pub}(m)$

send m to ϕ_{pub} on behalf of $\mathcal{P}$

$\mathsf{Ideal}_{\mathbf{cash}}$: include $\mathsf{Ideal}_{\text{cash}}$ (Figure 7.1).

FIGURE 7.4: Definition of $\mathsf{Ideal}_{\text{hawk}}$. Notations: $\mathsf{ContractCoins}$ denotes frozen coins owned by the contract; Coins denotes the global private coin pool defined by $\mathsf{Ideal}_{\text{cash}}$; and $(\mathsf{in}_i, \mathsf{val}_i)$ denotes the input data and frozen coin value of party $\mathcal{P}_i$.

ContractHawk$(\mathcal{P}_\mathcal{M}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\text{priv}}, \phi_{\text{pub}})$

Init: See $\mathsf{Ideal}_{\text{hawk}}$ (Figure 7.4) for description of parameters
Call ContractCash.**Init**.

Freeze: Upon receiving $\texttt{freeze}(\pi, \mathsf{sn}_i, \mathsf{cm}_i)$ from $\mathcal{P}_i$:
- assert current time $T \le T_1$
- assert this is the first freeze from $\mathcal{P}_i$
- let MT be a merkle tree built over Coins
- assert $\mathsf{sn}_i \notin$ SpentCoins
- statement := $(\mathsf{MT.root}, \mathsf{sn}_i, \mathsf{cm}_i)$
- assert NIZK.Verify$(\mathcal{L}_{\texttt{FREEZE}}, \pi, \mathsf{statement})$
- add sn_i to SpentCoins and store cm_i for later

Compute: On receive $\texttt{compute}(\pi, \mathsf{ct})$ from $\mathcal{P}_i$:
- assert $T_1 \le T < T_2$ for current time T
- assert NIZK.Verify$(\mathcal{L}_{\texttt{COMPUTE}}, \pi, (\mathcal{P}_\mathcal{M}, \mathsf{cm}_i, \mathsf{ct}))$
- send $\texttt{compute}(\mathcal{P}_i, \mathsf{ct})$ to $\mathcal{P}_\mathcal{M}$

Finalize: On receiving $\texttt{finalize}(\pi, \mathsf{in}_\mathcal{M}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$ from $\mathcal{P}_\mathcal{M}$:
- assert current time $T \ge T_2$
- for every $\mathcal{P}_i$ that has not called compute, set $\mathsf{cm}_i := \perp$
- statement := $(\mathsf{in}_\mathcal{M}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$
- assert NIZK.Verify$(\mathcal{L}_{\texttt{FINALIZE}}, \pi, \mathsf{statement})$
- for $i \in [N]$:
  - assert $\mathsf{coin}'_i \notin$ Coins
  - add coin'_i to Coins
  - send $\texttt{finalize}(\mathsf{coin}'_i, \mathsf{ct}_i)$ to $\mathcal{P}_i$
- Call ϕ_{pub}.check$(\mathsf{in}_\mathcal{M}, \mathsf{out})$

ContractCash: include ContractCash
ϕ_{pub} : include user-defined public contract ϕ_{pub}

Relation $(\mathsf{stmt}, \mathsf{wit}) \in \mathcal{L}_{\texttt{FREEZE}}$ defined as:
- parse stmt as $(\mathsf{MT.root}, \mathsf{sn}, \mathsf{cm})$
- parse wit as $(\mathcal{P}, \mathsf{coin}, \mathsf{sk}_{\texttt{prf}}, \mathsf{branch}, s, \$\mathsf{val}, \mathsf{in}, k, s')$
- coin := $\mathsf{Comm}_s(\$\mathsf{val})$
- assert MerkleBranch$(\mathsf{MT.root}, \mathsf{branch}, (\mathcal{P}\|\mathsf{coin}))$
- assert $\mathcal{P}.\mathsf{pk}_{\texttt{prf}} = \mathsf{sk}_{\texttt{prf}}(0)$
- assert $\mathsf{sn} = \mathsf{PRF}_{\mathsf{sk}_{\texttt{prf}}}(\mathcal{P}\|\mathsf{coin})$
- assert $\mathsf{cm} = \mathsf{Comm}_{s'}(\$\mathsf{val}\|\mathsf{in}\|k)$

Relation $(\mathsf{stmt}, \mathsf{wit}) \in \mathcal{L}_{\texttt{COMPUTE}}$ defined as:
- parse stmt as $(\mathcal{P}_\mathcal{M}, \mathsf{cm}, \mathsf{ct})$
- parse wit as $(\$\mathsf{val}, \mathsf{in}, k, s', r)$
- assert $\mathsf{cm} = \mathsf{Comm}_{s'}(\$\mathsf{val}\|\mathsf{in}\|k)$
- assert $\mathsf{ct} = \mathsf{ENC}(\mathcal{P}_\mathcal{M}.\mathsf{epk}, r, (\$\mathsf{val}\|\mathsf{in}\|k\|s'))$

Relation $(\mathsf{stmt}, \mathsf{wit}) \in \mathcal{L}_{\texttt{FINALIZE}}$ defined as:
- parse stmt as $(\mathsf{in}_\mathcal{M}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$
- parse wit as $\{s_i, \$\mathsf{val}_i, \mathsf{in}_i, s'_i, k_i\}_{i\in[N]}$
- $(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out}) := \phi_{\text{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}_{i\in[N]}, \mathsf{in}_\mathcal{M})$
- assert $\sum_{i\in[N]} \$\mathsf{val}_i = \sum_{i\in[N]} \$\mathsf{val}'_i$
- for $i \in [N]$:
  - assert $\mathsf{cm}_i = \mathsf{Comm}_{s_i}(\$\mathsf{val}_i\|\mathsf{in}_i\|k_i))$ $\vee(\$\mathsf{val}_i, \mathsf{in}_i, k_i, s_i, \mathsf{cm}_i) = (0, \perp, \perp, \perp, \perp)$
  - assert $\mathsf{ct}_i = \mathsf{SENC}_{k_i}(s'_i\|\$\mathsf{val}'_i)$
  - assert $\mathsf{coin}'_i = \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$

Protocol ProtHawk$(\mathcal{P}_\mathcal{M}, \{\mathcal{P}_i\}_{i\in[N]}, T_1, T_2, \phi_{\textbf{priv}}, \phi_{\textbf{pub}})$

Init: Call ProtCash.**Init**.

Protocol for a party $\mathcal{P} \in \{\mathcal{P}_i\}_{i\in[N]}$:

Freeze: On input $\texttt{freeze}(\$\mathsf{val}, \mathsf{in})$ as party $\mathcal{P}$:
- assert current time $T < T_1$
- assert this is the first freeze input
- let MT be a merkle tree over Contract.Coins
- assert that some entry $(s, \$\mathsf{val}, \mathsf{coin}) \in$ Wallet for some (s, coin)
- remove one $(s, \$\mathsf{val}, \mathsf{coin})$ from Wallet
- $\mathsf{sn} := \mathsf{PRF}_{\mathsf{sk}_{\texttt{prf}}}(\mathcal{P}\|\mathsf{coin})$
- let branch be the branch of $(\mathcal{P}, \mathsf{coin})$ in MT
- sample a symmetric encryption key k
- sample a commitment randomness s'
- $\mathsf{cm} := \mathsf{Comm}_{s'}(\$\mathsf{val}\|\mathsf{in}\|k)$
- statement := $(\mathsf{MT.root}, \mathsf{sn}, \mathsf{cm})$
- witness := $(\mathcal{P}, \mathsf{coin}, \mathsf{sk}_{\texttt{prf}}, \mathsf{branch}, s, \$\mathsf{val}, \mathsf{in}, k, s')$
- $\pi :=$ NIZK.Prove$(\mathcal{L}_{\texttt{FREEZE}}, \mathsf{statement}, \mathsf{witness})$
- send $\texttt{freeze}(\pi, \mathsf{sn}, \mathsf{cm})$ to ContractHawk
- store in, cm, \$val, s', and k to use later (in compute)

Compute: On input compute as party $\mathcal{P}$:
- assert current time $T_1 \le T < T_2$
- sample encryption randomness r
- $\mathsf{ct} := \mathsf{ENC}(\mathcal{P}_\mathcal{M}.\mathsf{epk}, r, (\$\mathsf{val}\|\mathsf{in}\|k\|s'))$
- $\pi :=$ NIZK.Prove$((\mathcal{P}_\mathcal{M}, \mathsf{cm}, \mathsf{ct}), (\$\mathsf{val}, \mathsf{in}, k, s', r))$
- send $\texttt{compute}(\pi, \mathsf{ct})$ to ContractHawk

Finalize: Receive $\texttt{finalize}(\mathsf{coin}, \mathsf{ct})$ from ContractHawk:
- decrypt $(s\|\$\mathsf{val}) := \mathsf{SDEC}_k(\mathsf{ct})$
- store $(s, \$\mathsf{val}, \mathsf{coin})$ in Wallet
- output $\texttt{finalize}(\$\mathsf{val})$

Protocol for manager $\mathcal{P}_\mathcal{M}$:

Compute: On receive $\texttt{compute}(\mathcal{P}_i, \mathsf{ct})$ from ContractHawk:
decrypt and store $(\$\mathsf{val}_i\|\mathsf{in}_i\|k_i\|s_i) := \mathsf{DEC}(\mathsf{epk}, \mathsf{ct})$
store $\mathsf{cm}_i := \mathsf{Comm}_{s_i}(\$\mathsf{val}_i\|\mathsf{in}_i\|k_i)$
output $(\mathcal{P}_i, \$\mathsf{val}_i, \mathsf{in}_i)$
If this is the last compute received:
- for $i \in [N]$ such that $\mathcal{P}_i$ has not called compute, $(\$\mathsf{val}_i, \mathsf{in}_i, k_i, s_i, \mathsf{cm}_i) := (0, \perp, \perp, \perp, \perp)$
- $(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out}) := \phi_{\text{priv}}(\{\$\mathsf{val}_i, \mathsf{in}_i\}_{i\in[N]}, \mathsf{in}_\mathcal{M})$
- store and output $(\{\$\mathsf{val}'_i\}_{i\in[N]}, \mathsf{out})$

Finalize: On input $\texttt{finalize}(\mathsf{in}_\mathcal{M}, \mathsf{out})$:
assert current time $T \ge T_2$
for $i \in [N]$:
- sample a commitment randomness s'_i
- $\mathsf{coin}'_i := \mathsf{Comm}_{s'_i}(\$\mathsf{val}'_i)$
- $\mathsf{ct}_i := \mathsf{SENC}_{k_i}(s'_i\|\$\mathsf{val}'_i)$

statement := $(\mathsf{in}_\mathcal{M}, \mathsf{out}, \{\mathsf{cm}_i, \mathsf{coin}'_i, \mathsf{ct}_i\}_{i\in[N]})$
witness := $\{s_i, \$\mathsf{val}_i, \mathsf{in}_i, s'_i, k_i\}_{i\in[N]}$
$\pi :=$ NIZK.Prove(statement, witness)
send $\texttt{finalize}(\pi, \mathsf{in}_\mathcal{M}, \mathsf{out}, \{\mathsf{coin}'_i, \mathsf{ct}_i\})$ ContractHawk

ProtCash: include ProtCash.

FIGURE 7.5: ProtHawk construction.

```
1   HawkDeclareParties(Seller,/* N parties */);
2   HawkDeclareTimeouts(/* hardcoded timeouts */);

3     // Private contract φpriv
4   private contract auction(Inp &in, Outp &out) {
5     int winner = -1;
6     int bestprice = -1;
7     int secondprice = -1;

8     for (int i = 0; i < N; i++) {
9       if (in.party[i].$val > bestprice) {
10        secondprice = bestprice;
11        bestprice = in.party[i].$val;
12        winner = i;
13      } else if (in.party[i].$val > secondprice) {
14        secondprice = in.party[i].$val;
15      }
16    }

17    // Winner pays secondprice to seller
18    // Everyone else is refunded
19    out.Seller.$val = secondprice;
20    out.party[winner].$val = bestprice - secondprice;
21    out.winner = winner;
22    for (int i = 0; i < N; i++) {
23      if (i != winner)
24        out.party[i].$val = in.party[i].$val;
25    }
26  }

27    // Public contract φpub
28  public contract deposit {
29    // Manager deposits $N
30    def check():
31      send $N to Manager
32    def managerTimeOut():
33      for (i in range($N)):
34        send $1 to party[i]
35  }
```

FIGURE 7.6: Hawk contract for a second-price sealed auction. Code described in this paper is an approximation of our real implementation. In the public contract, the syntax "send $N to P" corresponds to the following semantics in our cryptographic formalism: $\mathsf{ledger}[P] := \mathsf{ledger}[P] + \$N$ – see Section 6.3.

```
1  typedef enum {ROCK, PAPER, SCISSORS} Move;
2  typedef enum {DRAW, WIN, LOSE} Outcome;
3  typedef enum {OK, A_CHEAT, B_CHEAT} Output;

4  // Private Contract parameters
5  HawkDeclareParties(Alice, Bob);
6  HawkDeclareTimeouts(/* hardcoded timeouts */);
7  HawkDeclareInput(Move move);

8  Outcome outcome(Move a, Move b) {
9    return (a - b) % 3;
10 }
11 private contract game(Inp &in, Outp &out) {
12   if (in.Alice.$val != $1)  out.out = A_CHEAT;
13   if (in.Bob.$val != $1)    out.out = B_CHEAT;
14   Outcome o = outcome(in.Alice.move, in.Bob.move);
15   if   (o == WIN)  out.Alice.$val = $2;
16   else if (o == LOSE)  out.Bob.$val  = $2;
17   else out.Alice.$val = out.Bob.$val = $1;
18 }

19 public contract deposit() {
20   // Alice and Bob each deposit $2
21   // Manager deposits $4
22   def check(Output o):
23     send $4 to Manager
24     if (o == A_CHEAT): send $4 to Bob
25     if (o == B_CHEAT): send $4 to Alice
26     if (o == OK):
27       send $2 to Alice
28       send $2 to Bob
29   def managerTimedOut():
30     send $4 to Bob
31     send $4 to Alice
32 }
```

FIGURE 7.7: Hawk program for a rock-paper-scissors game. This program defines both a private contract and a public contract. The private contract guarantees that only Alice, Bob, and the Manager learn the outcome of the game. Public collateral deposits are used to guarantee financial fairness such that if any of the parties cheat, the remaining honest parties receive monetary compensation.

```
1   // Raise $10,000 from up to N donors
2   #define BUDGET $10000

3   HawkDeclareParties(Entrepreneur, /* N Parties */);
4   HawkDeclareTimeouts(/* hardcoded timeouts */);

5   private contract crowdfund(Inp &in, Outp &out) {
6     int sum = 0;
7     for (int i = 0; i < N; i++) {
8       sum += in.p[i].$val;
9     }
10    if (sum >= BUDGET) {
11      // Campaign successful
12      out.Entrepreneur.$val = sum;
13    } else {
14      // Campaign unsuccessful
15      for (int i = 0; i < N; i++) {
16        out.p[i].$val = in.p[i].$val; // refund
17      }
18    }
19  }
```

FIGURE 7.8: Hawk contract for a kickstarter-style crowdfunding contract. No public portion is required. An attacker who freezes but does not open would not be able to recover his money.

```
1   typedef enum {OK, A_CHEAT, B_CHEAT} Output

2   HawkDeclareParties(Alice, Bob);
3   HawkDeclareTimeouts(/* hardcoded timeouts */);
4   HawkDeclarePublicInput(int stockprice,
                           int threshold[5]);
5   HawkDeclareOutput(Output o);

6   int threshold_comm[5] = {/* harcoded */};

6   private contract swap(Inp &in, Outp &out) {
7     if (sha1(in.Alice.threshold) != threshold_comm)
        out.o = A_CHEAT;
7     if (in.Alice.$val != $10) out.o = A_CHEAT;
8     if (in.Bob.$val != $10) out.o = B_CHEAT;
8
9     if (in.stockprice < in.Alice.threshold[0])
        out.Alice.$val = $20;
10    else out.Bob.$val = $20;
11  }

12  public contract deposit {
13    def receiveStockPrice(stockprice):
14      // Alice and Bob each deposits $10
15      // Assume the stock price authority is trusted
16      // to send this contract the price
17      assert msg.sender == StockPriceAuthority
18      self.stockprice = stockprice
19    def check(int stockprice, Output o):
20      assert stockprice == self.stockprice
21      if (o == A_CHEAT): send $20 to Bob
22      if (o == B_CHEAT): send $20 to Alice
23      if (o == OK):
24        send $10 to Alice
25        send $10 to Bob
26  }
```

FIGURE 7.9: Hawk program for a risk-swap financial instrument. In this case, we assume that the manager is a well-known entity such as a stock exchange, and therefore the contract does not protect against the manager defaulting. An aborting manager (e.g., a stock exchange) can be held accountable through external means such as legal enforcement or reputation, since aborting is observable by the public.

Chapter 7

Conclusion

The goal of this thesis has been to construct a provably-secure cryptocurrency protocol, also layer-by-layer. Our contributions are based mainly on two new abstractions. The first abstraction is a new variant of computational puzzles, called scratch-off puzzles (SOPs). This definition generalizes the puzzle used in Bitcoin, but captures its essential security requirements; indeed in Chapter 2 we generically showed that the Nakamoto consensus protocol can securely be instantiated with any SOP. Besides modeling existing protocols, this abstraction also provides an effective way to design upgrades for cryptocurrencies: by instantiating Nakamoto consensus with alternative puzzles that have beneficial side effects. In Chapter 3 we constructed scratch-off puzzles that offer improved incentives and discourage harmful coalitions from forming. In Chapter 4 we constructed a scratch-off puzzle that provides archival data storage as a secondary function, recycling some of the computational effort. These puzzle constructions address important problems facing the cryptocurrency ecosystem today. We use our framework to make formal arguments about their security.

The second abstraction (Chapter 5) is a "blockchain contract," which represents an application built on top of a public transaction log. This abstraction includes both "transparent contracts," which can be implemented directly on top of a public network like Bitcoin and Ethereum because they have no hidden state; and "private contracts," which we implement (in Chapter 6) using a novel cryptographic protocol, Hawk, based on zero-knowledge proofs. While existing cryptocurrency systems provide either programmability (like Ethereum [28]) or transaction privacy (like Zerocash [7]), Hawk is the first protocol to provide both. Hawk is also practical, and includes a correct-by-construction compiler for user-defined applications.

Our work demonstrates several virtues of the provable security approach. First, by building formal models, we can better understand even complex and emergent systems. We showed that Bitcoin embodies a legitimately novel protocol design in the field of distributed systems, circumventing prior impossibility results for anonymous systems. This novelty provides a partial explanation for its surprising success. Second, by seeking general abstractions, we find natural places to extend the protocol, gaining additional functionality and additional security.

Bibliography

[1] The rise and rise of bitcoin. Documentary, 2014.

[2] Garrick Hileman. State of blockchain q1 2016: Blockchain funding overtakes bitcoin. http://www.coindesk.com/state-of-blockchain-q1-2016/, May 2016.

[3] Andrew Miller and Joseph J. LaViola, Jr. Anonymous byzantine consensus from moderately-hard puzzles: A model for bitcoin. UCF Tech Report. CS-TR-14-01.

[4] Andrew Miller, Ahmed Kosba, Jonathan Katz, and Elaine Shi. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *ACM CCS*, pages 680–691, 2015.

[5] Andrew Miller, Ari Juels, Elaine Shi, Bryan Parno, and Jonathan Katz. Permacoin: Repurposing Bitcoin Work for Data Preservation. In *IEEE Symposium on Security and Privacy*, May 2014.

[6] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 136–145. IEEE, 2001.

[7] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on. IEEE*. IEEE, 2014.

[8] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. *IEEE Security and Privacy*, 2015.

[9] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In *CRYPTO*, 1990.

[10] Berry Schoenmakers. Security aspects of the EcashTM payment system. *State of the Art in Applied Cryptography*, 1998.

[11] Ronald L Rivest. Peppercoin micropayments. In *Financial Cryptography*, 2004.

[12] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to better – how to make bitcoin a better currency. In *Financial Cryptography and Data Security*, pages 399–414. Springer, 2012.

[13] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In S&P, 2015.

[14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in bitcoin p2p network. In *ACM CCS*, pages 15–29, 2014.

[15] Alex Biryukov and Ivan Pustogarov. Bitcoin over Tor isn't a good idea. In *IEEE Symposium on Security and Privacy*, 2015.

[16] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P*, 2013.

[17] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. 2015.

[18] Andrew Miller, James Litton, Andrew Pachulski, Neal Gupta, Dave Levin, Neil Spring, and Bobby Bhattacharjee. Discovering bitcoin's public topology and influential nodes (preprint). http://cs.umd.edu/projects/coinscope, 2015.

[19] Andrew Miller and Rob Jansen. Shadow-bitcoin: Scalable simulation via direct execution of multi-threaded applications. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, 2015.

[20] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. Cryptology ePrint Archive, Report 2016/454, 2016. http://eprint.iacr.org/2016/454.

[21] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Eurocrypt*, 2015.

[22] Iddo Bentov and Ranjit Kumaresan. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO*, 2014.

[23] Meni Rosenfeld. Analysis of bitcoin pooled mining reward systems. *arXiv preprint arXiv:1112.4980*, 2011.

[24] Forrest Voight. p2pool: Decentralized, dos-resistant, hop-proof pool. https://bitcointalk.org/index.php?topic=18313.0, June 2011.

[25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[26] Vitalik Buterin. Bitcoin network shaken by blockchain fork. http://bitcoinmagazine.com/3668/bitcoin-network-shaken-by-blockchain-fork/, 2013.

[27] Jon Matonis. The bitcoin mining arms race: Ghash.io and the 51% issue. http://www.coindesk.com/bitcoin-mining-detente-ghash-io-51-issue/, July 2014.

[28] Gavin Wood. Ethereum: A secure decentralized transaction ledger. http://gavwood.com/paper.pdf, 2014.

[29] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. https://eprint.iacr.org/2015/460, 2015.

[30] Dorit Ron and Adi Shamir. Quantitative Analysis of the Full Bitcoin Transaction Graph. In *Financial Cryptography*, 2013.

[31] 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 *IMC*, 2013.

[32] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Security and Privacy in Social Networks*, pages 197–223. Springer, 2013.

[33] Philip Koshy, Diana Koshy, and Patrick McDaniel. An analysis of anonymity in bitcoin using p2p network traffic. In *International Conference on Financial Cryptography and Data Security*, pages 469–485. Springer, 2014.

[34] Tomas Sander and Amnon Ta-Shma. Auditable, anonymous electronic cash. In *Annual International Cryptology Conference*, pages 555–572. Springer, 1999.

[35] Tomas Sander, Amnon Ta-Shma, and Moti Yung. Blind, auditable membership proofs. In *International Conference on Financial Cryptography*, pages 53–71. Springer, 2000.

[36] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy*, 2014.

[37] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *IEEE Security and Privacy*, 2013.

[38] Nicolas van Saberhagen. Cryptonote v 2.0. *https://cryptonote.org/whitepaper.pdf*, 2013.

[39] George Danezis, Cedric Fournet, Markulf Kohlweiss, and Bryan Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *PETShop*, 2013.

[40] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369. ACM, 1986.

[41] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure Multiparty Computations on Bitcoin. In *S&P*, 2013.

[42] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 705–734. Springer, 2016.

[43] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *Theory of cryptography*, pages 477–498. Springer, 2013.

[44] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Kuesters, and Daniel Rausch. Universal composition with responsive environments. Cryptology ePrint Archive, Report 2016/034, 2016. http://eprint.iacr.org/2016/034.

[45] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology–EUROCRYPT 2013*, pages 626–645. Springer, 2013.

[46] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *S&P*, 2013.

[47] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, 2013.

[48] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE, 2015.

[49] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In *Annual International Cryptology Conference*, pages 566–598. Springer, 2001.

[50] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. How to use snarks in universally composable protocols. Cryptology ePrint Archive, Report 2015/1093, 2015. http://eprint.iacr.org/2015/1093.

[51] Jacob Goldstein. The island of stone money. Planet Money. http://www.npr.org/sections/money/2011/02/15/131934618/the-island-of-stone-money, February 2011.

[52] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[53] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[54] John R Douceur. The sybil attack. In *International Workshop on Peer-to-Peer Systems*, pages 251–260. Springer, 2002.

[55] Michael Okun. *Distributed Computing Among Unacquainted Processors in the Presence of Byzantine Failures*. PhD thesis, Hebrew University of Jerusalem, 2005.

[56] Jeffrey Considine, Matthias Fitzi, Matthew Franklin, Leonid A Levin, Ueli Maurer, and David Metcalf. Byzantine agreement given partial broadcast. *Journal of Cryptology*, 18 (3):191–217, 2005.

[57] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Anne-Marie Kermarrec, Eric Ruppert, et al. Byzantine agreement with homonyms. In *Proceedings of the 30th*

annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing, pages 21–30. ACM, 2011.

[58] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, 1996.

[59] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[60] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. pages 139–147, 1993.

[61] Ben Laurie and Richard Clayton. Proof-of-work proves not to work. In *Workshop on Economics and Information Security*, 2004.

[62] Bogdan Groza and Bogdan Warinschi. Cryptographic puzzles and DoS resilience, revisited. *Designs, Codes and Cryptography*, 2013.

[63] Ari Juels and John G. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*, volume 99, pages 151–165, 1999.

[64] Nikita Borisov. Computational puzzles as sybil defenses. In *Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 171–176. IEEE, 2006.

[65] James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged Byzantine impostors. Technical report, Yale, 2005.

[66] I. Eyal and E. Gun Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *FC*, 2014.

[67] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 305–320. IEEE, 2016.

[68] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. *arXiv preprint arXiv:1507.06183*, 2015.

[69] Jonathan Katz, Andrew Miller, and Elaine Shi. Pseudonymous secure computation from time-lock puzzles. Cryptology ePrint Archive, Report 2014/857, 2014.

[70] Marcin Andrychowicz and Stefan Dziembowski. Pow-based distributed cryptography with no trusted setup. In *Annual Cryptology Conference*, pages 379–399. Springer, 2015.

[71] Tibor Jager. How to build time-lock encryption. *IACR Cryptology ePrint Archive*, 2015: 478, 2015.

[72] Jia Liu, Saqib A. Kakvi, and Bogdan Warinschi. Extractable witness encryption and timed-release encryption from bitcoin. Cryptology ePrint Archive, Report 2015/482, 2015. http://eprint.iacr.org/2015/482.

[73] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1993.

[74] Liqun Chen, Paul Morrissey, Nigel P Smart, and Bogdan Warinschi. Security notions and generic constructions for client puzzles. In *Advances in Cryptology–ASIACRYPT 2009*, pages 505–523. Springer, 2009.

[75] Douglas Stebila, Lakshmi Kuppusamy, Jothi Rangasamy, Colin Boyd, and Juan Gonzalez Nieto. Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols. In *Topics in Cryptology–CT-RSA 2011*, pages 284–301. Springer, 2011.

[76] Bogdan Groza and Bogdan Warinschi. Cryptographic puzzles and dos resilience, revisited. *Designs, Codes and Cryptography*, pages 1–31, 2013.

[77] Melissa Chase and Anna Lysyanskaya. On signatures of knowledge. In *Annual International Cryptology Conference*, pages 78–96. Springer, 2006.

[78] Dario Fiore and Anca Nitulescu. On the (in) security of snarks in the presence of oracles. Technical report, Cryptology ePrint Archive, Report 2016/112, 2016.

[79] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[80] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[81] L Addario-Berry and BA Reed. Ballot theorems, old and new. In *Horizons of combinatorics*, pages 9–35. Springer, 2008.

[82] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, and Emin Gün. On scaling decentralized blockchains. In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[83] Danny Bradbury. Alydian targets big ticket miners with terahash hosting. http://www.coindesk.com/alydian-targets-big-ticket-miners-with-terahash-hosting/, August 2013.

[84] Joshua A Kroll, Ian C Davey, and Edward W Felten. The economics of bitcoin mining or, bitcoin in the presence of adversaries. *WEIS*, 2013.

[85] myVBO. ziftrcoin : A cryptocurrency to enable commerce. Whitepaper http://www.ziftrcoin.com/docs/ziftrcoin-whitepaper.pdf, October 2014.

[86] Spreadcoin. Whitepaper http://spreadcoin.net/files/SpreadCoin-WhitePaper.pdf, October 2014.

[87] Joseph Bonneau, Arvind Narayanan, Andrew Miller, Jeremy Clark, Joshua A Kroll, and Edward W Felten. Mixcoin: Anonymity for Bitcoin with accountable mixes. In *Financial Cryptography*, March 2014.

[88] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and cryptocurrency technologies*. Princeton University Pres, 2016.

[89] Adam L Beberg, Daniel L Ensign, Guha Jayachandran, Siraj Khaliq, and Vijay S Pande. Folding@ home: Lessons from eight years of volunteer distributed computing. In *Parallel & Distributed Processing (IPDPS)*, pages 1–8, 2009.

[90] Ari Juels and Burton S Kaliski Jr. PORs: Proofs of retrievability for large files. In *ACM CCS*, pages 584–597. ACM, 2007.

[91] R. K.L. Ko, S. S.G. Lee, and V. Rajan. Understanding cloud failures. *IEEE Spectrum*, 49 (12), 28 Nov. 2012.

[92] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[93] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35 (11):190–201, 2000.

[94] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[95] Gilad Asharov, Amos Beimel, Nikolaos Makriyannis, and Eran Omri. Complete characterization of fairness in secure two-party computation of boolean functions. In *Theory of Cryptography Conference (TCC)*, pages 199–228, 2015.

[96] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In *ACM CCS*, pages 195–206, 2015.

[97] Ranjit Kumaresan and Iddo Bentov. How to Use Bitcoin to Incentivize Correct Computations. In *CCS*, 2014.

[98] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. On the Malleability of Bitcoin Transactions. In *Workshop on Bitcoin Research*, 2015.

[99] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, 2014.

[100] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of finance*, 1961.

[101] Sebastian Angel and Michael Walfish. Verifiable auctions for online ad exchanges. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 195–206. ACM, 2013.

[102] Mark Bagnoli and Barton L. Lipman. Provision of public goods: Fully implementing the core through private contributions. *The Review of Economic Studies*, 1989.