Foundations of Decentralised Privacy

Thomas Kerber

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

2021

This thesis is dedicated to my grandmother, ***Jule Schütt****,*
who called me "Doctor" before the first word was written,
and who will no longer be able to witness it.

Diese Doktorarbeit ist meiner Großmutter, ***Jule Schütt****, gewidmet,*
die mich "Doktor" nannte, bevor das erste Wort geschrieben war,
und die es nicht mehr erleben wird.

Abstract

Distributed ledgers, and specifically blockchains, have been an immensely popular investment in the past few years. The heart of their popularity is due to their novel approach toward financial assets: They replace the need for central, trusted institutions such as banks with cryptography, ensuring no one entity has authority over the system. In the light of record distrust in many established institutions, this is attractive both as a method to combat institutional control and to demonstrate transparency. What better way to manage distrust than to embrace it? While distributed ledgers have achieved great things in removing the need to trust institutions, most notably the creation of fully decentralised assets, their practice falls short of the idealistic goals often seen in the field.

One of their greatest shortcomings lies in a fundamental conflict with privacy. Distributed ledgers and surrounding technologies rely heavily on the transparent replication of data, a practice which makes keeping anything hidden very difficult. This thesis makes use of the powerful cryptography of succinct non-interactive zero-knowledge proofs to provide a foundation for re-establishing privacy in the decentralised setting. It discusses the security assumptions and requirements of succinct zero-knowledge proofs at length, establishing a new framework for handling security proofs about them, and reducing the setup required to that already present in commonly used distributed ledgers. It further demonstrates the possibility of privacy-preserving proof-of-stake, removing the need for costly proofs-of-work for a privacy-focused distributed ledger. Finally, it lays out a solid foundation for a smart contract system supporting privacy – putting into the hands of contract authors the tools necessary to innovate and introduce new privacy features.

Acknowledgements

First, and most relevant to this work, I would like to thank my supervisors Aggelos Kiayias and Markulf Kohlweiss for their help and support during my PhD; for listening to my crazy ideas, and for helping me make them happen.

All members of the *cryptosec* research group have my thanks for the many interesting discussions they facilitated. In particular, I will remember with fondness my discussions with Dimitris Karakostas, Orfeas Thyfrontis Litos, Christian Badertscher, Mikhail Volkhov, Lorenzo Martinico, and Aydin Abadi. Though not in the *cryptosec* group, I also enjoyed my discussions with Dionysis Zindros, Matthias Fitzi, and Jamie Gabbay, when the opportunity arose.

I went through some tough personal times during my PhD, and I do not think I would have made it through if it were not for the invaluable support of my dear friends Margus Lind and Dagmara Niklasiewicz, as well as of course the consistent encouragement of my girlfriend, Yuk Shan Cheng. You have helped me press on when I had lost hope, and I am forever in your debt.

I am grateful to have a family who has always been available and supportive, even as I live far from them, and have not been able to see them in person in a while as the Covid-19 pandemic reared its ugly head. The feeling of safety that – if all else failed – I would always find a warm home to return to in Germany is one I cannot express enough appreciation for. A particular thanks to my father Manfred, for taking the time to proofread this behemoth of a document.

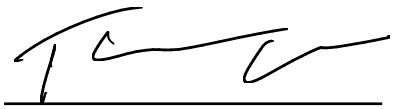
My girlfriend, Yuk Shan Cheng, has struggled through her PhD alongside me, and there is little doubt in my mind that struggling together is much easier – and oftentimes even enjoyable – than taking the road alone. I think I got the better end of the deal here – I enjoy great cooking and her lovely[I] company!

Finally, I would like to thank all friends who I did not explicitly mention. In particular, Paul Crone, who is always great fun to talk with, and who presents my only avenue to any kind of gossip in my life.

[I] Terms and conditions apply.

Declaration

I hereby declare this thesis was written by myself to an overwhelming degree. A minority of the writing was originally composed by co-authors, namely Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas, for joint research papers. Neither this work, nor any part thereof, has been submitted for any other degree or professional qualification.



Thomas Kerber

Lay Summary

Blockchains have reached great popularity recently, as seen by the high market price of Bitcoin. There is more than financial speculation here, as they cut out banks and other trusted institutions. As social tensions rise in our world, and people distrust many institutions more, removing the need for them is attractive. An unintended side-effect of the technology behind blockchains is that – contrary to common misinformation – they have no meaningful privacy, outside a few specialised exceptions. This thesis uses cryptography to re-establish a basis for privacy in blockchain systems by ensuring the cryptography can be properly applied to the problem, and broadening what it can be applied to from previous results.

Table of Contents

1 Introduction

Modern life increasingly relies on online networks and connections. The benefits to modern communications are clear: Actions previously done in-person can be done over vast distances. Actions which might take a long time to resolve in person, such as auctions or matching orders have been vastly sped up, as the increased networking allows people to access information and systems directly, rather than through middlemen.

Despite the great advances brought by the internet, as with any industrial revolution, it has also brought unexpected challenges. Large organisations such as Google, Microsoft, Facebook, and Apple have amassed an immense effective power – power which they are as of the time of writing under legal and legislative scrutiny for [LH20, KM20]. Regardless of the outcome of such investigations, it cannot be denied that these companies each have a great potential for censorship and leverage their power to limit potential competitors.

Prominently, tech giants have been accused of partisan censorship [PW20] and undue influence on the election process [Fol17], from both sides of the US political spectrum. This culminated in the ban of US president Donald Trump from Twitter [WP21] after the Capitol riots of January 2021. Online platforms such as GitHub, Twitter, and Reddit, moderate content opaquely to ambiguous terms of use, and even services considered as infrastructure, such as Cloudflare's denial of service protection and Amazon's AWS cloud computing, the likes of which are considered almost essential for operating a website in today's internet, have withdrawn their service for ideological reasons [Pri17, Nov21].

The popularity of cryptocurrencies, blockchains, and distributed ledgers (the terms often being merely different sides of the same coin), is largely explained through a disillusionment with centralised systems and a promise that without central oversight and dictation, a fairer, next generation network can be built.

This ideal of distributed ledgers empowering a more decentralised future is at odds with the privacy of their users. Somewhat paradoxically, even though

much of the centralised internet relies on the gathering and selling (either directly, or through advertising services) of personal data, the alternative proposed by distributed ledgers has, at least naively, an arguably greater cost: complete transparency.

While technological developments during the past few decades have desensitised many people to the sharing of their personal data, people are prone to overestimating the extent – major privacy scandals such as Cambridge Analytica [CGH18] still shock the general public, and in many areas, such as personal finances, privacy is still very much expected. Even in areas where personal data is frequently gathered and used, such as online purchasing habits and searches, users often expect this privacy violation to be limited – it is one thing for "the Google algorithm" to know what you have been searching for, but quite another for your colleague to.

In their naive form, which overwhelmingly is still their current form, blockchain-based systems are completely transparent. Their operation is primarily based on replication – everyone knows everything and can therefore hold everyone else to account. This is somewhat inherent: To reach agreement on what happened, one much *know* what happened. The challenge lies in the openness of distributed systems – while with a centralised system, such as a bank, you may be able to trust the respective entity with confidentiality (or at least to respect the law on privacy), with a decentralised system any entity could be looking to sell your data at the first opportunity.

Cryptography presents one of the best tools available to resolve this conflict. At its most basic form, simply the usage of public-key cryptography in all cryptocurrencies provides more privacy than their fiat-currency alternatives in one aspect: A person's bank account and credit cards are directly linked to their name and address. A public key is typically linked much less directly; only through third-parties which have gathered this information, usually because of legal requirements. Furthermore, the creation of new public keys, new "identities", is free and trivial – while bank accounts and cards are often also free to open, their creation is sufficient hassle to discourage making use of this fact.

During the early days of cryptocurrencies, the wild-west nature of the field and the pseudonymity provided by public-key cryptography caused a frequent misconception of *anonymity*, rather than pseudonymity. Even users recognising

the difference would often feel that pseudonymity was sufficient – nothing was directly linked to them, after all. Since then, Bitcoin has seen multiple high-profile deanonymisation attacks [MPJ+13, BKP14], disabusing this notion.

More complex cryptography *can* provide better anonymity and privacy: The Monero cryptocurrency [vS13] relies on ring signatures hiding the details of each transaction among a set of possibilities, with external observers not able to tell who made it. Even more powerful is Zerocash [BCG+14], which hides additional information of a transaction, except when it was made[1], using non-interactive zero-knowledge proofs of knowledge (NIZKs) as the basic tool. The details of how this proof works will be explained in Chapter 5, and NIZKs will follow us as a central tool throughout this thesis.

1.1 From Bitcoin to Ethereum and Beyond

The question of consensus was thought largely settled in the literature at the time Bitcoin [Nak08] was presented. n parties could agree, provided fewer than $n/3$ of these behaved adversarially – or in a *Byzantine* manner, as it is often called in the context of consensus. Practical protocols achieving this, notably PBFT (practical Byzantine fault-tolerance) [CL99] existed, and impossibility results for better thresholds were well-known.

Bitcoin [Nak08] threw a spanner in the academic works of consensus. While the basic goal was the same, to agree on something, the details were entirely different and largely novel:

1. The setting was not one of n parties, but the open internet.
2. Parties had influence according to their computational power, rather than a singular vote.
3. The consensus was a continuous, eventual agreement, rather than being immediate.
4. A threshold of adversarial power of 1/2 was achieved.

The argument for these security properties was largely informal, but was later formally proven [GKL15], the details of which we will recap in Section 2.4.

[1] There exist privacy attacks [KYMM18] on Zcash, the cryptocurrency based on Zerocash, however they are more indicative of user behaviour and design decisions than the protocol's security: When given an inconvenient option of privacy, users will ignore it.

2. and 3. are compromises – they are less strict than the traditional requirements of Byzantine consensus and come with their own problems. The former violates a fundamental human desire for fairness, and the latter introduces an uncertainty about when agreement has actually been formed (a fault which great effort has gone into redressing in various cryptocurrencies, rebranded as a novel feature of "finality" [Vuk15]). These relaxations directly enable 1. and 4., and while some distributed ledger protocols have compromised on 4., removing the restriction of a fixed set of parties is crucial for the distributed nature of distributed ledgers[2].

Proof-of-work and stake. Bitcoin's dependence on computational power solved the issue of *Sybil attacks*: In a setting where creating new connections and identities is effectively free, there must be some way to limit which identities get a say and which do not. Bitcoin re-purposed the existing tool of proof-of-work [DGN03]. The basic primitive is the cryptographic hash function (using the random oracle heuristic [BR93]), the output of which is unpredictable. A proof-of-work is simply demonstrating a hash preimage for which the output conforms to some pattern – for instance the first n bits being 0. This demonstrates having – on average – run the hash function 2^{n-1} times, demonstrating either an investment of computational resources, or luck, with the latter smoothing out due to the law of large numbers.

As Bitcoin grew in popularity, it quickly became apparent that proof-of-work has a major downside when directly tied to financial incentives: Rather than merely selecting the users which had the most computational power available, it actively encouraged the development of gigantic server farms with no other purpose than to generate proofs-of-work. Moreover, the difficulty of the proof-of-work adjusts automatically – it provides a competitive environment, but no limits on the competition. As a direct result, the energy consumption and waste generated by the proof-of-work process is immense. According to [Dig20] and, at the time of writing, Bitcoin alone has a carbon footprint comparable to New Zealand, with a yearly energy consumption of 75.50 TWh.

To address this issue, various alternatives have been proposed, with the most

[2] This restriction too has been lifted, for "federated" ledgers, in which a fixed set of parties can participate in consensus. This setting is of no interest for this thesis and while it has seen renewed interest since the development of Bitcoin, the cryptographic foundations of the setting are well explored.

prominent being proof-of-stake. As distributed ledgers are typically accompanied by a currency, the amount each user owns of this can be used to determine their voting power, rather than their computational resources. While proof-of-stake appears a harder problem to solve, solutions have been created and proven secure [BGK+18, GHM+17, DPS19]. There are trade-offs with respect to trust in proof-of-stake – if the initial stakeholders of a proof-of-stake system are not considered of a majority "honest", the system is inherently insecure, placing an unusual amount of power into the process of selecting the initial distribution. By contrast, proof-of-work is more self-regulating, as it does not require "permission" to participate in the consensus process. Given a sufficiently liquid asset, this is no longer a problem, and by contrast the advantages of the immensely reduced energy consumption are obvious.

"Smart" contracts. Bitcoin itself was designed primarily as a cryptographic currency: It supports transfers of funds, which are associated with a public key. In perhaps a surprising level of foresight, Bitcoin came with a limited programmability, in the form of short scripts which need to be satisfied to spend each coin (or more precisely, unspent transaction output, or UTxO). While the expressiveness of this scripting is limited, it allows for instance requiring multiple keys to sign off on a transaction, and locking funds for a fixed time. This expressiveness has been sufficient for some more innovative systems to be deployed in the Bitcoin ecosystem, despite it having few fundamental changes.

Bitcoin was nevertheless too restrictive for many applications users wanted to develop and, even where workarounds were possible, they were cumbersome and hard to develop. Various applications duplicated the consensus of Bitcoin and attached their own semantics for specific applications: NameCoin [KCE+15] providing a distributed alternative to domain names and Bitmessage [War12] providing a distributed ledger-based communication protocol, for instance.

This practice of separating the consensus of each application was not particularly sustainable. Not only did it splinter consensus into many different, small communities, the smaller of which could be attacked with comparative ease, but the effort of maintaining the complex consensus system went beyond what many of these applications offered. Ethereum [Woo14] provided an alternative approach: It aimed to provide a fully programmable blockchain as a platform, allowing users to deploy programmatic rules to govern – in a small part – the

consensus. These small programs have been called "smart contracts" and, as bad as I believe this name to be[3], it has stuck.

Smart contracts have been used for a multitude of applications, from the comparatively simple, such as auctions and subsidiary currencies (usually referred to as tokens), to the complex, such as the (now infamous) DAO (decentralised autonomous organisation). The increased complexity of smart contracts has also increased the attack surface, with Ethereum being the subject to multiple high-profile attacks over the years [CPNX20].

"Layer 2" solutions. After the popularity of distributed ledger exploded in 2017, it became apparent that there was a very real limit to the practical scale of the technology. Simply put, when everyone has to know everything, the amount of data processed grows quadratically. At the height of its popularity, fees for processing transactions soared, as users were forced to compete for the available bandwidth of the system, and many of the smaller, less financially driven applications were laid on the wayside – no longer worth the cost of running.

This quadratic growth of processing (or linear if considered per-person) is a direct result of the setting of mutual distrust, unlike trusted solutions which *decrease* in per-person processing. The most common initial naive "solution" is to split the network into smaller, independent parts (or shards, with the process being often called sharding), has a tremendous hidden cost: It amplifies the effective power of an adversary, who can pick and choose which part of the network to attack.

Recently, so-called "layer 2" solutions have become more prominent, including state channels [DFH18] and the Lightning network [KL20]. These process largely independently of the blockchain, falling back to it only as a dispute resolution mechanism. While this thesis does not concern itself directly with layer 2 solutions, their existence is important for the goal of privacy, as they exhibit privacy characteristics both positive and negative: Layer 2 solutions provide inherent privacy, by relying on less public, user-to-user interaction, but often imply relying on, and interacting with, semi-trusted third-parties, potentially replicating the privacy characteristics of the centralised internet.

[3] I contend: Smart contracts are neither smart, nor contracts. More accurately, they are reactive state machines, as we will be discussed in Chapter 6.

1.2 Disparate Approaches to Privacy

Privacy is one of the fundamental pillars of the field of cryptography. Many tools, such as secure multi-party computation, fully homomorphic encryption, and zero-knowledge provide great privacy guarantees. These are not all directly suitable to the distributed ledger space however and a few tools have seen the most success in preserving the privacy of distributed ledgers.

Statistical techniques. The first, and the one which we will not focus on, relies on statistical security. Privacy is preserved by muddying the paths, making unclear how funds are moving. The earliest conception of this approach lies in coin mixers, such as CoinJoin [Max13] and CoinShuffle [RMK14]. The idea is simple: A set of users deposit equal-value coins into a pot and each withdraws the same value again independently, with all users signing off on the result. Each user retains the same value, but (for an external observer) any output could belong to any user. The Monero cryptocurrency (based on [vS13]) bakes this model into its basic transactions: Using ring signatures, each transaction is placed into an anonymity set obscuring its true origin.

Zero-knowledge. The second prominent direction of privacy has used non-interactive zero-knowledge proofs-of-knowledge (NIZKs), to prove that a transaction is correct, and legally generated some output, while not revealing the details of the output, instead only cryptographically committing to it. This is the basic premise of Zerocoin [MGGR13] and its more notable successor Zerocash [BCG+14], which also relies on a highly efficient class of NIZKs, Succinct Non-interactive ARguments of Knowledge (SNARKs, or zk-SNARKs to emphasise their zero-knowledge property). These SNARKs are well-suited to usage in distributed ledgers: They are fast to verify, meaning it is possible to utilise them for all transactions, and have small proof sizes, independent of the complexity of what is being proven. They can be used as a drop-in replacement for signatures, but being able to authenticate much more complex information than knowledge of the correct secret key.

Despite this tool being powerful, it is still limited. Zero-knowledge alone does not let mutually untrusting users to interact arbitrarily, although it does enable many interactions in the face of adversity. As a result of this limitation

such approaches to privacy in the more complex setting of smart contracts, such as Hawk [KMS+16], re-introduce trust assumptions, approaching the centralised model of privacy to cope with the limitations of the setting. This is a good approach, however makes for a poor foundation for a platform. Had Ethereum stipulated that each contract must nominate a trusted party, interactions within it would almost certainly be much more restricted.

SNARKs come with their own drawbacks: First, they rely on non-falsifiable *knowledge assumptions* which make using them in larger compositional settings more difficult, and second, they require a *trusted setup*: They rely on a randomly sampled Structured Reference String (or SRS), with the process sampling the string being able to retain information which could be used to break the security of the SNARK. These drawbacks are not to be underestimated – a flaw in Zcash's initial reference string design could have enabled illicit printing of vast amounts of funds in the corresponding network [SWB19].

Multi-party computation. Beyond these commonly used approaches to privacy on distributed ledgers, three further approaches are noteworthy. First, among other cryptographic approaches, secure Multi-Party Computation (or MPC) is a very powerful candidate. Its use for privacy in smart contracts has often been proposed, however it is directly opposed to scalability aims: MPC, while practical, takes orders of magnitude longer to run than "native" computation. While it is possible to deploy in the fully distributed setting, by electing representative committees from the participants (as some proof-of-stake protocols, such as Algorand [GHM+17] already do), this committee would either need to perform an infeasible amount of computation, or the ledger would have much reduced throughput.

Layer 2 solutions. MPC still has good applications combined with the second approach: As already mentioned in Section 1.1, layer 2 solutions are inherently less open, with higher levels of privacy. As layer 2 solutions generally concern small sets of semi-trusted parties, running MPC between these can reduce the trust required further, spreading the cost of the overhead, as each user only needs to compute what they are interested in[4].

[4] Incidentally, this is the same reason why SNARKs are attractive: Proving is expensive, but you need prove only what you are interested in.

Trusted hardware. Finally, it is commonly proposed to utilise trusted hardware to bypass the limitations and computational cost of cryptographic tools. This is an especially attractive option for the private sector, as it reduces costs, while being able to offer additional features. Tools such as Intel SGX and AMD SVE are available for immediate use and require relatively little development effort when compared to deploying advanced cryptography. Nevertheless, the implications for privacy are murky: Centralised trust is placed on the design of the processors' secure elements, their production, and less obvious things such as microcode updates. While it can (and should!) be argued that users must trust their machines already – users typically do not trust the machines of *others* and, especially when it comes to privacy, information is leaked at the weakest link. Given a history of side-channel attacks on secure hardware [VMW+18, MOG+20], the real privacy of such systems is questionable and, given the reliance of central trust, it will not be able to resist nation-state actors[5].

1.3 Thesis Outline

This thesis focuses on how to utilise zero-knowledge proofs to provide a better foundation for privacy in the distributed ledger space. The contributions are split into four main parts, each following one of the research papers produced during this thesis, and each addressing a pertinent problem to deploying privacy in distributed ledgers. While the privacy and decentralisation discussed here seem far from the politics at the start of this section, I believe distributed systems to be crucial in opposing centralised structures – and the basics of their privacy are still poorly understood beyond financial transactions. This thesis' primary motivation is then to provide a foundation for constructing and reasoning about distributed and private systems.

- **Composition with Knowledge Assumptions.** In Chapter 3 the shortcomings of existing compositional security techniques in handling knowledge assumptions (and by extension, zk-SNARKs) are outlined and a new approach that permits composable security proofs with minimal changes is presented. This substantiates the folklore belief that the usage of zk-

[5] Few things are when pressed. Given rising geopolitical tensions between the USA and China, who broadly control the software and hardware we use respectively, it is worth considering that both will likely be leveraged in future geopolitical conflicts, however.

SNARKs in larger protocols is secure, although there is still ground for caution: The reuse of knowledge assumptions presents a potential danger, one with real-world consequences, as the same groups are frequently used for various systems. This is a crucial prerequisite for using efficient zero-knowledge proof systems, which the latter part of this thesis relies on.

- **Secure Reference Strings from Consensus.** Chapter 4 handles the issue of the trusted setup required for zk-SNARKs and observes a mechanism to perform the setup from the same trust assumptions of Nakamoto-based blockchains (i.e. honest majority of some resource). The result holds for both proof-of-stake and proof-of-work blockchains, although it is easier to establish for the latter. To simplify, each creator of a block also performs an "update" on the reference string, which relies on this being among a class of "updateable" reference strings. Given existing results on the frequency of honest blocks, an honest update is eventually guaranteed. Again, this is done as a prerequisite for using zk-SNARKs in the final two parts of this thesis.

- **Privacy in Proof-of-Stake.** The benefits of proof-of-stake are apparent and, despite some drawbacks, outweigh their downsides, as discussed in Section 1.1. Proof-of-stake exists in an inherent conflict to privacy on the underlying currency however, with its operation being influenced by ostensibly private data. In Chapter 5 we resolve this conflict as far as possible, following an adaptation of Zerocash [BCG+14] and Ouroboros Genesis [BGK+18] to derive a provably secure and private proof-of-stake protocol, with some concessions being made for network leakage. This can provide assurance that private systems are not inherently more wasteful than public ones – something increasingly becoming a point of contention for Bitcoin, for example.

- **Privacy in Smart Contracts.** The disparate approaches to privacy in smart contract systems and lack of a unified and foundational approach is one of the primary challenges to privacy in the distributed ledger space. Users cannot write a privacy-preserving contract with the same ease as they can write one in Ethereum. This is party inherent, as privacy is hard, but is also explained by there being a lack of a good foundational framework to build from. Chapter 6 presents a foundation of privacy in

smart contracts, based on zero-knowledge proofs of correct state updates and argues why this is useful.

Beyond these chapters, which make up the core of this thesis, Chapter 2 introduces necessary background material and important related work is. Chapter 7 summaries and ties together the core contributions.

2 Background

This chapter presents the foundations required to understand the main body of this thesis, given a general computer science background. We begin in Section 2.2 by introducing basic fundamentals of cryptographic proofs, such as hardness assumptions and reductions, as well as how they relate to the more sophisticated constructions primarily used in this thesis. Notably the introduction of the universal computation and constructive cryptography frameworks, as well as knowledge assumptions are central to the work in Chapter 3.

Section 2.4 covers the origins of distributed ledgers, their properties, and how secure proof-of-stake is constructed and combined with compositional security frameworks. The properties and modelling of distributed ledgers is of importance to Chapters 4 to 6, while the design of secure proof-of-stake protocols, and more specifically the Ouroboros family of protocols, is central to Chapter 5.

The basics of zero-knowledge and zk-SNARKs are presented in Section 2.5. Important are the fundamental characteristics of zero-knowledge proofs, the additional properties of updateability and universality, required for Chapter 4 and Chapter 6, respectively. Section 2.5 also discusses the commonalities of zk-SNARKs, and why they often rely on knowledge assumptions.

Finally, Section 2.6 discusses the design of smart contract systems, starting with the archetypical smart contract system of Ethereum. The section also covers UTxO-based smart contract systems, which Bitcoin's scripting language may be considered a part of, and a few of the more notable privacy focused variations of smart contract systems.

2.1 Mathematical and Programming Notations

This thesis uses common functional programming expressions in various places, including the following for precision:

- Lambda expressions: $(\lambda x\colon 2x)(2) = 4$

- List and tuple literals: $[1, 2]$ and $(1, 2)$.
- List accessors: $[1, 2][0] = 1$
- List head and tail: $\mathsf{head}([1, 2]) = 1$, $\mathsf{tail}([1, 2]) = [2]$
  We consider tail permissive, i.e. $\mathsf{tail}([]) = []$
- Tuple projection: $\mathsf{proj}_1(1, 2) = 1$
- List concatenation: $[1, 2] \parallel [3, 4] = [1, 2, 3, 4]$
- The higher-order function filter: $\mathsf{filter}(\lambda x{:}\, x \equiv 0 \mod 2, [1, 2]) = [2]$
- The higher-order function foldl: $\mathsf{foldl}(+, 0, [1, 2, 3]) = 6$
- Curried functions: $\mathsf{foldl}(+) = \lambda s, l{:}\, \mathsf{foldl}(+, s, l)$

Maps are seen as functions from keys to values, allowing map defaults to be represented by initialising to a constant function, for instance $\lambda x{:}\, 0$. Lists are freely used to represent the set of their elements. The symbols $\bot$ and $\varnothing$ are overloaded, with the former representing both "false" and "error/abort", while the latter represents the empty set, empty map, and in some cases, the initial state. Further, for a map M, $k \in M$ denotes that the map contains the key k. A key is not in the map if and only if $M(k) \coloneqq \bot$. For lists, ε denotes the empty list and $\parallel$ denotes list concatenation. Single non-list items can be interpreted as a singleton list.

The following functions are used throughout the thesis.

- The function $\mathsf{prefix}(L, x)$ returns the shortest prefix of L containing x, or L itself, if no such prefix exists.
- The function $\mathsf{idx}(L, x)$ returns the index of the first occurrence of x in L, or $\bot$ if $x \notin L$.
- The functions $\mathsf{take}(n, L)$ and $\mathsf{drop}(n, L)$ return the first n items in L, and all but the first n items in L respectively.
- The function $\mathsf{last}(L)$ returns the last element in a list.
- The function $\mathsf{reverse}(L)$ reverses the order of a list.
- The function $\mathsf{dedup}(L)$ returns L, with only the first occurrence of any element retained.

The relation $a \prec b$ denotes the (reflexive) list prefix, and $L^{\lceil k}$ is used to denote the prefix of L missing the last k entries.

Finally, **assert** and **abort** are used throughout this thesis. The statement "**assert** x" is equivalent to "**if** $\neg x$ **then abort**". Where it occurs, **abort** is seen to mean "the current experiment outputs $\bot$". Note that this is different from the

machine outputting $\perp$ itself, and in particular applies to the whole simulation experiment we will introduce in Subsection 2.2.2.

2.2 Cryptographic Proof Styles

Cryptographic proofs vary greatly in scale and requirements: From the information-theoretic proof of the security of the one-time pad, to complex high-level proofs of interactive systems relying on many unusual assumptions. This thesis focuses more on the latter, the details of which depend on compositional security frameworks which assume basic knowledge. The core ideas at play are presented in this section.

2.2.1 Basics of Computational Security

At the base of security proofs are statements about security. Typically these are expressed through so-called *security games*, short sequences of interactions with an adversary. This adversary is typically modelled as a set of potential algorithms (or sometimes multiple such algorithms, if the adversary operates in stages). The set most typically has restrictions on the adversary's execution time, which will be discussed shortly, with other restrictions also being possible.

The adversary in the game has some winning condition – for instance, outputting the correct plaintext when it is given a ciphertext. The goal of any security proof is to demonstrate that for any adversary, its probability of success in the game is low. There are different flavours of what "low" means, depending on the game and the setting. As there is typically a way for the adversary to simply guess the correct answer, rather than considering the probability of success directly, often the *advantage* of the adversary is used instead: how much larger any adversary's probability of success is than the probability of a random guess succeeding.

If the supremum of adversarial advantages is zero, security is *perfect*. If not, rather than directly working with concrete probabilities, the success probability is usually expressed in terms of a *security parameter*, for which κ will be used throughout this thesis[1]. The usage of a security parameter allows tuning the level of security required from non-perfect primitives. Typically this is taken

[1] The security parameter is often supplied in unary, written as 1^κ, to make explicit its impact on algorithmic complexity.

a step further: Rather than concerning ourselves with the concrete advantage probabilities, cryptographers consider the asymptotics of these: Denote $\kappa^{-\omega(1)}$ by $\mathsf{negl}(\kappa)$, typically security proofs demonstrate that the probability of adversarial success is *negligible*, defined as it lying in $\mathsf{negl}(\kappa)$. We often abuse notation and write $\varepsilon \leq \mathsf{negl}(\kappa)$ to mean $\varepsilon \in \kappa^{-\omega(1)}$.

The security parameter often serves a double purpose, also limiting the time complexity of the set of adversaries permitted to $\kappa^{\omega(1)}$. This is not necessarily the case, however is considered the default, with more powerful adversaries (for instance unbounded) adversaries usually being explicitly introduced.

While not unique to cryptography, this field has a strong reliance on assumptions – these take many shapes, and we will introduce the more exotic knowledge assumptions in Subsection 2.2.3, the most common are *hardness assumptions*. These are specified as a game themselves, with this game being secure *by assumption*. Often this is used though a reduction: The security of another game is proved by demonstrating how to construct a (victorious) adversary against the hardness assumption from a (victorious) adversary against the game subject to the proof.

2.2.2 Simulation-Based Security

To prove the correctness of implementations of software systems, a common approach is to prove its equivalence to a simpler (perhaps due to it being unoptimised) specification. A similar approach exists for security properties: A complex interactive system can be proven secure by demonstrating its equivalence, in terms of execution semantics, to a corresponding specification of ideal behaviour. Importantly, while real protocols will have as foundational points leaky network infrastructure and mutually distrusting parties, the ideal specification can assume the existence of a trusted third party, with the specification simply being a description of what this party does, when it receives (perfectly hidden) messages from the protocol participants. Cryptographers often refer to the ideal specification as the *ideal world* and the protocol as the *real world*.

Both worlds are usually modelled with an adversary[2], which can influence the world in certain ways. For an encrypted channel, it may be able to eavesdrop on network communications, and for an authenticated channel, inject its own.

[2] It is possible to have multiple, independent adversaries as well. This is not a setting considered in this thesis, which assumes all adversaries are colluding.

The adversary can also be a participant in the protocol, or control many users at once (though if the adversary controls everything, there is no user for whom security should hold). The more power is afforded to the adversary, the larger class of attacks the protocol can withstand.

The adversary has a curious role in the ideal world. Naively, one might think there is no point to an adversary in the specification. The need to establish equivalence and its existence in the real world mandates it however, and forces concessions into the ideal world. At best, any participant the adversary controls in the real world, it also will control in the ideal world. The adversary may be afforded additional powers as well, however. For instance, even in an authenticated, secure channel's ideal specification, the adversary is informed whenever a message is sent, and usually provided with its length.

The secure channel's leakage of the message length does not match the leakage observed in the real world, a full ciphertext. It is here that the "simulation" part of the security statement comes in: An equivalence proof specifies a new ideal-world adversary, called the simulator, for every potential adversary in the real world. This simulator must coerce the ideal world to match the behaviour of the real world, together with the adversary: It must simulate what the adversary would do in this equivalent situation, for which it needs to simulate what leakage should happen in the real world. For the secure channel, it may sample a corresponding random string and hand it to the adversary, pretending it is a ciphertext. Importantly, the simulator must be able to take any actions the real-world adversary wants to take, otherwise the two side-by-side executions will diverge and no longer be equivalent. This is the essence of a simulation security proof: Any attack which works against the real world can be simulated against the ideal specification – where by definition, it is no attack, but intended behaviour.

2.2.3 Hybrid Models and Knowledge Assumptions

Beyond the hardness assumptions discussed in Subsection 2.2.1, assumptions are sometimes made about the existence of a cryptographic primitive. These are often approximated by a construction which cannot be proven secure – its security is then heuristic, rather than proven. The most prominent instance of this kind of assumption is one which features in this thesis at multiple points:

The random oracle. The ideal description of this consists of a third party who receives arbitrary inputs. If they have seen the input before, they output their previous response again. If not, they randomly sample a fixed-length bit string, record it, and output it.

The random oracle is a convenient abstraction of a hash function, although this abstraction is flawed. For any concrete hash function, the abstraction demonstrably *does not* hold [CGH98]. It follows that the usage can only ever be heuristic. A similar situation occurs with the (comparatively simpler) common random string (CRS) model. In this model, a fixed number of bits are randomly sampled at the start of the protocol and made available to all users. Without a trusted party, it is difficult to do this sampling in practice[3]. Nevertheless, there are reasonable approaches to take in the real-world, which work heuristically: Measurements of the physical world can be used to provide entropy. A hash function can be used (as a random oracle), and provided with a distinct input everyone agrees on – perhaps the name of the protocol, or a recent news headline, as was done for Bitcoin.

A further assumption which is often made to enable using a rounds structure of communication is a *discrete clock*, which advances only when all interested parties give it the go-ahead, and allows all parties to read the current time[4]. Unlike with many other ideal functionalities, this thesis uses the clock *globally* – i.e. anyone can access it at any time. Notably this means the environment can talk to it directly and it is present in both worlds.

A much stronger heuristic assumption is made in proof-of-stake[5] protocols, where an assumption over the initial distribution of funds is made – specifically, that most funds are distributed to honest users. This is of course impossible to demonstrate in the real-world, where initial distributions often follow a similar pattern to initial public offerings of companies – whoever is interested and has the funds to buy.

Knowledge assumptions. In zk-SNARKs a class of assumptions known as *knowledge assumptions* are prominently used. At their most basic, these are

[3]Although coin-flipping protocols are a possibility, they are also unreasonable in the setting of a non-fixed set of users, which we will consider.

[4]This has plenty of practical uses, however raises both questions of physical limitations and the reliability of our time-keeping when under active attack.

[5]Proof-of-work protocols also have to make strong assumptions about the frequency of hashes performed. These are not central to this thesis.

implications of the kind "If someone knows x, they must also know y". Most often, y encodes information on how to construct x. This allows extracting information about the *intent* behind an action, which can be used in the proof, or by the simulator, to ensure the action made sense. For example, a knowledge assumption may state that you must know the preimage of a hash to produce this hash (or more likely, either know this, or demonstrate you sampled it randomly in some way). A simulator could use this knowledge in the simulation of a hash-based commitment scheme to retrieve the original input and commit to this in the ideal world.

We will discuss the details more in Chapter 3, however the basic form of the knowledge assumption states that for any algorithm which outputs x, there must exist a corresponding extractor, which outputs the corresponding y. Knowledge assumptions are powerful and seem necessary for succinctness in many cases, as the simulator needs to be able to retrieve the original inputs – something it cannot do information-theoretically without additional help when the output belongs to a smaller domain. The downside of this approach is that they are non-falsifiable: To disprove a knowledge assumption, one would have to prove that for some adversary, *no possible extractor* exists.

2.2.4 Adaptive Security

Subsection 2.2.1 already mentioned it is common for specific parties in a cryptographic protocol to be considered adversarial. This is often also called *static corruption*: The adversary effectively begins the Protocol by deciding which parties it wishes to control, within some limits. In many cases, this is a sufficiently powerful model, however for long-running systems it is worth considering something more powerful: An adversary which can "corrupt" parties *while the protocol is running*. Such an adversary is called *adaptive*, due to its ability to adapt who to corrupt based on information it gathers during the protocol's execution.

This is a particularly important consideration for proof-of-stake systems. In these systems, as funds shift from party to party, it may become easy for an adaptive adversary to corrupt users which hold only a minority of stake now – therefore not violating the honest majority assumption – but *did* hold most stake in the past, enabling them to re-write history. We will discuss this conflict in more detail in Chapter 5. This may appear an unimportant detail, but it comes

with a real-world analogue, making adaptive corruption a considerable threat in this case: A user who has spent their stake in the system, no longer has reason to protect their secret keys and may be careless with them: Installing malware on the machine without as much thought, or selling it second-hand. There is little question that obtaining old secrets is an easier task than obtaining fresh ones.

2.3 Composition

The usage of simpler cryptographic tools in larger constructions is common – Bitcoin would fall apart without digital signatures, and layer 2 solutions like Lightning would fall apart without Bitcoin. In the context of simulation security, it is desirable to specify each in terms of their idealised abstraction: Bitcoin should be secure for any signature primitive, not just its current elliptic curve based solution[6], and Lightning should work for any distributed payments system.

Despite this concept being natural, it is not immediate, with the devil, as usual, lying in the details. The exact notion of equivalence used in the simulation proof matters, as it needs to be powerful enough to allow replacing the ideal primitive with its realisation in the larger system, while still preserving any security proofs. In practice, this requires two things: First, the equivalence notion must be one of semantic equivalence, that is, both the real and the ideal system will output the same values for the same sequence of inputs[7], and second, the proof of equivalence must apply in the presence of the larger protocol and its environment.

Several frameworks exist to facilitate such composable proofs, by requiring security proofs to be with respect to an arbitrary environment, which can make any sequence of interactions it wishes. They typically define an explicit equivalence relation, usually requiring a negligible advantage to distinguishing between the real and ideal world for any of the arbitrary environments. Combined, these two properties ensure that composition is possible.

This approach is adopted both by the Universal Composition [Can01] framework and the Constructive Cryptography [Mau11] framework (which is based on Abstract Cryptography [MR11]), although both differ in their details. While

[6] An especially important point in the face of quantum cryptography.

[7] Technically, they must output a sufficiently close random distribution – occasional failures and different sampling methods can be tolerated.

many of the results in this thesis were first stated in the Universal Composition framework, the inclusion of knowledge assumptions, which is discussed in Subsection 2.2.3, is incompatible with either of these frameworks. The results of Chapter 3 make up for this limitation, with the rest of the thesis being formalised in the related UC model.

2.3.1 Universal Composability

The most popular compositional framework is *Universal Composability* [Can01], commonly referred to as *UC*. This framework is loosely used throughout this thesis, making the basis of the results from Chapters 4 to 6. Its use is "loose", as these results are transferable to other frameworks, but are expressed in matching language and notation.

Interactive Turing Machines. UC makes statements about sets of interactive Turing machines, or ITMs. These are modelled on traditional Turing machines with a few extensions for interaction and stochastic operation. They possess additional input and backdoor tapes which allow ITMs to communicate with each other, and a random tape which is provided an infinite sequence of uniformly sampled bits. ITMs are instantiated with an identifier, with the combination of an encoding of the ITM's behaviour and the identifier making up an *extended ID*. This can be used in a new *external write* operation, which can be used to write to either the input or backdoor tapes of another ITM instance (ITM instances, or ITIs, are independently executing copies of the same underlying machine). Formally, an output tape exists which first the extended ID of the indented recipient is written to, then the message itself. When the external write operation is executed, this tape is erased, and its content instead put on the recipient's tape.

Flow Control. The interaction between ITIs is not free-form, but is formally restricted by a *control function*, which may deny or alter external write operations. In practice, this is used to prevent addressing the internals of a protocol directly. An exception is made for the *adversary*, which is permitted to write onto any backdoor tape. This interaction allows each ITI to define their own adversarial influence. The details of the control function, and when calls are permitted and when not are too long for this section, however the general premise is that – aside from the adversary – all external writes form a tree structure, with protocols being

able to invoke sub-protocols and receive information back from them. Execution is kept serial for simpler analysis – one ITI is active at any time, and when it performs an external write, it becomes inactive, with the recipient being activated instead.

UC-emulation. A statement of security in UC follows from the notion of *UC-emulation*. Before defining this, it is first necessary to formally define protocols in the UC setting, and their idealised equivalents. Roughly speaking, a protocol consists of a number of different ITIs, each representing one of the parties in the protocol, running a program dictated by the protocol's code. Often (especially in the decentralised setting) the code each party runs is identical. They may interact with other ITIs, which can model assumptions (such as the random oracle, or a shared network), and with the adversary (which is often used as a fully subvertable network). In the ideal world, parties are still represented as ITIs, however they simply forward their inputs to a single *trusted* ITI, the specification of which is usually called an *ideal functionality*, and denoted $\mathcal{F}$. These forwarding parties are referred to as *dummy parties*. In both settings, the adversary is assumed to be addressable – it can be sent messages, and send them in turn as well. Beyond this, parties get inputs and supply outputs to something external to them. This can be a larger protocol context, or the end user themselves.

The key to composition in the UC setting is that, in addition to the protocol and adversary, a third component is considered, called the *environment* (written $\mathcal{Z}$). This may be *any* ITI, which interacts with either the real-world protocol, or the ideal-world specification, without knowing which. It instructs the behaviour of the adversary[8] in both worlds, and must attempt to determine which world it resides in. If the environment cannot succeed, the real protocol is as secure as its ideal specification.

Simulation-based security (see Subsection 2.2.2) is formally used, with the ideal-world adversary being a *simulator* (often written $\mathcal{S}$) which can mimic real-world attacks on the ideal protocol, rather than the real-world adversary itself. Additionally there are complexities in ensuring all ITIs run in polytime (in particular, it is difficult to capture with respect to what execution time should by polynomial), however these are not of significant importance to this thesis.

[8]The adversary can be considered as part of the environment without loss of generality. Indeed, this is part of the proof of composition of UC: the adversary is replaced with a "dummy" adversary which simply does as instructed.

Crucial to UC-emulation is the order of quantifiers. Given exec as the distribution of executing a system of ITIs, UC-emulation is given by:

Definition 2.1. A protocol π UC-emulates a protocol ϕ if and only if

$$\forall \mathcal{A}: \exists \mathcal{S}: \forall \mathcal{Z}: \text{exec}(\mathcal{Z}, \mathcal{A}, \pi) \approx \text{exec}(\mathcal{Z}, \mathcal{S}, \phi),$$

where $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{Z}$ denote the adversary, simulator, and environment respectively.

Universal Composition. Universal composition then states that a part of a larger protocol can be replaced with a protocol which realises it. For instance, an authenticated channel protocol ρ may rely on a digital signature functionality $\mathcal{F}_{\text{sig}}$, which in turn is UC-emulated by a protocol π using elliptic curve cryptography. Then ρ with $\mathcal{F}_{\text{sig}}$ replaced with π (written $\rho^{\mathcal{F}_{\text{sig}} \to \pi}$) also UC-emulates ρ. Intuitively, this result is as the rest of the ITIs in ρ can be seen as part of the environment $\mathcal{Z}$. A simplified statement of the UC theorem is:

Theorem 2.1. *For all protocols ρ, ϕ, π, where ϕ is a part of ρ, and π UC-emulates ϕ, $\rho^{\phi \to \pi}$ UC-emulates ρ.*

2.3.2 Constructive Cryptography

A less popular but mathematically simpler compositional framework is *Constructive Cryptography* [Mau11]. While this framework is not used directly in this thesis, the model used in Chapter 3 is closely based on it.

Random Systems. First introduced in [Mau02], a *random system* is intuitively a stateful variant of a randomly distributed function. Formally:

Definition 2.2. An $(\mathcal{X}, \mathcal{Y})$-random system $\mathbf{F}$ is an infinite sequence of conditional probability distributions $P^{\mathbf{F}}_{Y_i \mid X^i Y^{i-1}}$ for $i \geq 1$, where X_i and Y_i distribute over $\mathcal{X}$ and $\mathcal{Y}$ respectively.

Specifically, random systems produce outputs in the domain $\mathcal{Y}$ when given an input in $\mathcal{X}$ and are stateful – their behaviour can depend on prior inputs and outputs. [Mau11] itself works with random systems based on an automaton with internal state; such an automaton can then also be constrained to a reasonable notion of feasibility, such as being limited to a polynomial number of execution steps with respect to some security parameter.

Interfaces, Resources, and Converters. Constructive cryptography begins by defining a set $\mathcal{I}$ of *interfaces*, through which one may interact with a random system. Typically, these represent protocol parties: $\mathcal{I}$ may be $\{A, B, E\}$ for a point-to-point channel, for instance, to represent the parties Alice, Bob, and Eve the eavesdropper. A random system which allows each of these interfaces to interact with it is called a *resource*, typically denoted with a capital letter, such as R. A *converter* is a random system operating only on a single interface – it connects to the interface of some underlying resource, and provides a replacement interface for it. As it is possible to reduce two random systems interacting with each other to a single random system (we will touch on this in Subsection 3.2.3), attaching a converter to a resource on an interface results in another random system. Converters are typically denoted with lowercase Greek letters, for instance α, and attached to a resource R on an interface $i \in \mathcal{I}$ by writing $\alpha^i R$.

Parallel composition is defined for both resources and converters, written $\alpha \parallel \beta$ or $R \parallel S$, although we will not cover the details here.

Distinguishers, Simulators, and Security. A distinguisher D in Constructive Cryptography is yet another random system, which forms the inverse of a resource: It connects to the resource on all available interfaces, and interacts with it, before externally outputting a single bit. This bit is interpreted as a guess as to which of two resources the distinguisher is connected with. Let $\Delta^D(R, S)$ be the statistical distance between DR and DS. Then the distance $d(R, S)$ is defined as the supremum over all possible D (ignoring subtleties such as feasibility notions):

$$d(R, S) \coloneqq \sup_D \Delta^D(R, S).$$

For security statements, we need to distinguish between attacker interfaces and honest interfaces in $\mathcal{I}$. A converter π^i is defined for each of the honest interfaces $i \in \mathcal{I}$, representing the *protocol* being executed. Furthermore, a converter σ^i is defined for each of the attacker interfaces $i \in \mathcal{I}$, representing the simulator. Their key difference is that the protocol is attached in the real-world, while the simulator is attached in the ideal world. We write $\vec{\pi}$ for the combination of all π^i, and $\vec{\sigma}$ for the combination of all σ^i. If:

$$d(\vec{\pi}R, \vec{\sigma}S) \leq \varepsilon,$$

then we can write this as:

$$R \xrightarrow{\vec{\pi}, \varepsilon} S,$$

read as "R constructs S". This constitutes a security proof that S can be securely realised by using R.

Rules of Composition. The construction notion of [Mau11] is composable in the sense that security proofs are preserved under parallel and sequential composition.

Theorem 2.2. *Proofs in the Constructive Cryptography framework are* ***sequentially composable*** (2.1)*,* ***parallelly composable*** (2.2)*, and* ***reflexive*** (2.3)*.*

$$R \xrightarrow{\vec{\pi},\varepsilon} S \wedge S \xrightarrow{\vec{\pi}',\varepsilon'} T \implies R \xrightarrow{\vec{\pi}'\vec{\pi},\varepsilon_1+\varepsilon_2} T \tag{2.1}$$

$$R \xrightarrow{\vec{\pi},\varepsilon} S \wedge R' \xrightarrow{\vec{\pi}',\varepsilon'} S' \implies R \parallel R' \xrightarrow{\vec{\pi}\parallel\vec{\pi}',\varepsilon_1+\varepsilon_2} S \parallel S' \tag{2.2}$$

$$R \xrightarrow{\vec{1},0} R \tag{2.3}$$

2.3.3 Globality, Corruption, and Other Caveats

Globality. A few caveats arise in the practical usage of compositional frameworks, in UC and other similar frameworks. First, it is frequently useful to have multiple high-level protocols interact with the same sub-functionality. A great example of this is the clock sketched in Subsection 2.2.3 – multiple protocols will realistically share the same time units, and not operate vastly out-of-sync with each other. This concept of a functionality being available more broadly is also called *globality*. In UC this is not natively available, however modifications to the framework [CDPW07, BCH+20] enable globality. Intuitively, globality equates to changing the requirement of a tree-structure for internal calls to requiring a directed acyclic graph structure instead. Looking forward to Chapter 3, this is also stricter than necessary – any graph structure can be used.

Corruption. A further question is how to model corruption – often static corruption can simply be modelled as a subset of the parties being controlled by the adversary, that is, it determines their behaviour, and sends messages on their behalf. Adaptive corruptions, as described in Subsection 2.2.4, are more complex. Generally, the methodology in UC is to allow dummy parties to receive a special message CORRUPT, which orders them to switch control from the environment to the adversary. In either case, care must be taken to ensure that the adversary

does not violate limits on corruption, and does not corrupt different users at different levels – for instance, corrupting a user in a signature scheme which is used in an authenticated channel should also corrupt that user of the authenticated channel. Often ideal functionalities also make use of honesty, and specify different behaviour for honest and dishonest users. While there is nothing preventing the same from being done in the real world, there it does not match reality (rather inconveniently; it would make many problems far simpler).

Wrappers. It is occasionally useful to separate a core behaviour of the ideal world or of the real world with a modification that is made to it for practical purposes. This thesis makes use of wrappers at two points: In Chapter 4, an idealised reference string functionality is modified to be available only after a set time, and in Chapter 5, the real-world protocol is modified to deny all adversarial zero-knowledge proofs if it violates the corruption limits. Formally, this is achieved through *wrapper functionalities*, often denoted with $\mathcal{W}$. In practice, these are simply functions which map a (set of) ITIs to a corresponding set with modified behaviour. An early example of this usage is in [BGK+18], with the wrapper in Chapter 5 being closely modelled on this.

Aborts. It is common for cryptographic protocols to abort. In practice this can mean that a party leaves the protocol unfinished. This interpretation makes less sense in the decentralised setting, where leaving the protocol is permissible, and should not affect its execution. Instead, this thesis uses aborts to mark violations of assumptions. This can refer to small violations, such as an input being in the incorrect format, to large assumptions such as honest majority. Semantically, throughout this thesis, the keyword **abort** should be taken to mean that the full current execution exits immediately, and outputs $\perp$. Crucially, in a real/ideal world experiment, both worlds should abort under the same conditions. In the real world this is often simpler to achieve – the simulator can abort whenever it pleases.

"Simple" UC. The description given in Subsection 2.3.1 technically limits itself to a simplified view of UC – in the full framework, ITIs are able to create new ITI instances simply by addressing them directly. For analysis, it is often easier to ignore this feature, and instead focus on static interaction graphs. This thesis

only makes use of such a "static" form of UC, and notably relies on this being closely related to the framework used for knowledge assumption in Chapter 3.

2.3.4 Notational Conventions in This Thesis

The style of writing UC protocols and functionalities differs greatly from author to author. As a result potentially important corner cases may be overlooked, as the exact behaviour of a given functionality is sometimes unclear. This thesis adopts a more explicit style, while at the same time attempting to avoid writing unnecessary information in the definition of the functionalities. While the proofs, protocols and functionalities can be read and understood without explicit knowledge of the notation described in this section, this section defines some of the behaviour left implicit in them.

Flow of execution. Session identifiers are formally used in UC to shield a protocol from external calls, except when allowed by the control function. While they are effectively a technical detail of the description in UC, they are often replicated in the description of functionalities and protocols. Session identifiers are implicit in this thesis. In a similar vein, it is often a convention to replicate (part of) the input to a functionality when returning the result, to ensure that it is clear which query is being answered. This is omitted as well, in favour of simply stating the actual value returned. Both of these are how a protocol would be written in a channel-based communications model, such as that of [Mau11], rather than the tape-based model of UC itself.

When a functionality is processing something, it is always processing *on behalf of some party*, which may be the adversary itself, or may be corrupted. Likewise when a protocol is processing something, it is processing this *on behalf of its owning party*. When a functionality or protocol hands off execution to another entity, by making a query to another functionality, or the adversary, execution *for this party* is suspended, and resumes only when the query returns. Attempts by the environment to make queries to a suspended protocol will be ignored. Likewise, if the environment attempts to query a functionality with a party which is currently suspended, the query will also be ignored. Crucially, the environment may still query a functionality with *another party* while one is suspended, ensuring that parties may still act concurrently. This behaves equally in protocols and functionalities, as the functionality is suspended in the same situation the

corresponding party's protocol is suspended. Finally, we assume that queries will eventually return – this is equivalent to queries which do not return, as we allow the environment and adversary to hold off indefinitely until returning. While this is possible, in practice, due to the implicit suspension mechanism described above, this means disabling a party permanently.

This above mechanism is not a great deviation from UC – it can easily be implemented by having a functionality or protocol record locally the suspension, and reject new queries from the suspended party until it receives an input of a specified form. We simply omit this mechanism when writing our protocols. Responsive environments [CEK+16] are a strictly stronger form of this idea.

We assume the existence of a set of all parties $\mathcal{P}$, of which there is a subset of honest parties $\mathcal{H} \subseteq \mathcal{P}$. We assume $\mathcal{H} \neq \varnothing$. Correspondingly, the set of corrupted parties is $\mathcal{P} \setminus \mathcal{H}$. All functionalities are assumed to have knowledge of these sets. Real world protocols, when they interact with these sets, will assume that, if the party running them is ψ, $\mathcal{P} = \mathcal{H} = \{\psi\}$, that is, they know of no other parties except themselves, at least initially.

Notation. As the adversary may respond arbitrarily to queries, each query includes a well-formedness condition, and a fallback distribution. In particular, **query** $\mathcal{A}$ **with** x and **receive the reply** y, **satisfying** P, else **sampling from** D means the following: Send x to $\mathcal{A}$, then wait for the response y. If $P(y)$ does not hold, instead randomly sample y from D. This allows us to ensure responses are well-formed, while avoiding the common technique of aborting in the ideal world on receiving unexpected input, something we try to avoid, as it effectively permits denial-of-service in the "ideal" world. Finally, the period (".") is used as a membership access operator, to talk about variables of simulated functionalities, or in the proof, to talk about state variables of various functionalities and protocol instances. For instance, $\mathcal{F}.X$ means the state variable X within the (possibly simulated) functionality $\mathcal{F}$.

2.3.5 Commonly Used Functionalities

Subsection 2.2.3 has already introduced some cases where a functionality is assumed. Notably the random oracle $\mathcal{F}_{\mathrm{RO}}$, global clock $\mathcal{G}_{\mathrm{clock}}$, and common reference string functionality $\mathcal{F}_{\mathrm{CRS}}^{D}$ are used at multiple points. For this reason, and

also to familiarise the reader with the notation used for functionalities, they are introduced here.

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

The random oracle functionality $\mathcal{F}_{\text{RO}}$ returns a uniform random value in $\{0,1\}^\kappa$ for each input.

State variables and initialisation values:

Variable	Description
$H \coloneqq \varnothing$	A map from inputs to (fixed) outputs

When receiving a message $(\textsc{query}, x)$ *from a party* ψ*:*

if $x \notin H$ **then let** $H(x) \xleftarrow{*} \{0,1\}^\kappa$
return $H(x)$

Functionality $\mathcal{G}_{\text{clock}}$

The global clock allows parties to agree on some discrete notion of time.

State variables and initialisation values:

Variable	Description
$t \coloneqq 0$	Current time
$T \coloneqq \varnothing$	Timekeepers
$A \coloneqq \varnothing$	Agreements to advance

When receiving a message REGISTER *from a party* ψ*:*

let $T \leftarrow T \cup \{\psi\}$

When receiving a message DEREGISTER *from a party* ψ*:*

let $T \leftarrow T \setminus \{\psi\}$

When receiving a message UPDATE *from a party* ψ*:*

let $A(\psi) \leftarrow \top$
if $\forall \psi \in T\colon A(\psi)$ **then**
  let $t \leftarrow t + 1$; $A \leftarrow \lambda\psi\colon \bot$
  query $\mathcal{A}$ **with** TICK-TOCK

When receiving a message READ *from a party* ψ*:*

return t

Functionality $\mathcal{F}_{\text{CRS}}^{D}$

The CRS functionality $\mathcal{F}_{\text{CRS}}^{D}$ samples random values from D, until one satisfies the adversary. This is made publicly available to anyone.

State variables and initialisation values:

Variable	Description
$s \coloneqq \bot$	The reference string

When receiving a message CRS *from a party ψ:*

if $s = \bot$ **then**
  repeat
    let $s' \xleftarrow{*} D$
    query $\mathcal{A}$ **with** (CONFIRM-CRS, s) and **receive the reply** b
  until $b \vee s \neq \bot$
  let $s \leftarrow s'$
return s

2.4 Distributed Ledgers

Distributed ledgers were conceived in Bitcoin [Nak08] to provide a decentralised mechanism to track ownership of a digital currency. While this is still their primary purpose, the basic primitive is more flexible: Bitcoin achieved this tracing of ownership through a ledger, an ordered record of who sent funds to whom. Anyone can write to this record and insert a transaction[9], with the ledger guaranteeing an *approximately* correct ordering. Two transactions submitted close to each other (typically meaning within a few hours, depending on assumptions made on honesty and the security parameter) they may appear in a different order, while if they are sufficiently far apart, their temporal order is preserved on the ledger. Ledgers are also typically assumed to be readable by all, although Chapter 5 will discuss a partially private variant. In this section we will detail the exact security provided, and how this has been achieved in both proof-of-work and proof-of-stake.

[9] Provided they pay a transaction fee and follow the appropriate format, although these are not crucial for the primitive itself.

2.4.1 Bitcoin and Proof-of-Work

Bitcoin [Nak08] was proposed and implemented without a formal security analysis. It builds on the pre-existing tools of peer-to-peer networking, digital signatures, and proof-of-work. Prior to Bitcoin, proof-of-work had been proposed as a mechanism to deal with email spam [DGN03] and used in prior unsuccessful attempts at digital currencies.

Proof-of-work. The term "proof-of-work" describes a class of primitives consisting of a proving and verifying algorithm. They operate much like signatures, with it being possible to perform a proof-of-work being on any arbitrary message and verifying it with respect to the same message. Much like signatures, proofs-of-work must be binding with respect to the message – another message should not be substitutable after the fact. Unlike signatures, no keys are involved however. They are substituted with a different, interesting property: The *only* way to generate a valid proof-of-work is to run the proving algorithm, which has a known average-case runtime[10]. Thus the verifier succeeding attests to the prover having (on average) devoted this runtime into generating the proof. A closely related concept are *verifiable delay functions* [BBBF18], which differ in that instead of the exponential success distributions involved in the usual proofs of work, they are distributed more consistently around the mean.

A straightforward implementation of proof-of-work relies on repeated application of a random oracle in the random oracle model, or in its heuristic realisation, repeated hashing. To begin, sample a random nonce and concatenate it to the message. Pass this into the random oracle and interpret the result as a binary number. If this is sufficiently small, for an l-bit output, less than 2^{l-k}, for a k dictating the hardness of the "work", the nonce serves as a proof of work. If not, repeat until it does. In the case of less than 2^{l-k}, it takes on average 2^{k-1} attempts before the random oracle outputs a sufficiently small value, therefore attesting to sufficient "work" having been done. Furthermore, verification is independent of the difficulty: Only one random oracle invocation is necessary to check that the work claimed is available. As applying the random oracle with a nonce is a commitment scheme, the commitment properties are also guaranteed.

[10] Assuming all parties have the same computational resources. In practice, these are variable, with the proof-of-work runtime being inversely proportional to the resources available.

"Nakamoto" consensus. Given the basic tool of proof-of-work and a peer-to-peer broadcast network which reliably sends information to all other participants[11], it is possible to construct a distributed ledger given a uniform starting point through the so-called "Nakamoto" consensus algorithm, after the pseudonymous author of Bitcoin. Specifically, this starting point, called a *genesis block* in Bitcoin, ensures that no user has been pre-computing proofs-of-work, by requiring all proofs-of-work to sign information which was not known before the protocol start. This can be modelled as an assumption through a common random string, although in Bitcoin it was achieved by embedding a headline from *The Times*.

From the initial genesis, a difficulty target is chosen – k in the sketch of proof-of-work above. Any user may create a "block", a collection of transactions and some metadata and append it to the already existing data, provided its metadata includes a cryptographic reference to the previous block, in form of its hash and it contains a proof-of-work reaching the difficulty target over the message of the remainder of the block (including the reference to the prior block).

This creates a tree of blocks, with the genesis block as its root, due to blocks being verified recursively until the genesis is reached. Only if each block in the path from genesis to leaf block is correctly formatted and contains a correct proof-of-work is the chain as a whole considered valid. Whenever a new block is created (and therefore a new, longer chain is), its creator broadcasts it to all other users. These attempt to create their own blocks, but always start with the currently longest chain (breaking ties when needed). As honest users keep to this protocol, it is unlikely that they will create a wide tree, as whenever a new, longer chain is available they start trying to extend it instead. The only case when honest users will work on different branches is when ambiguity arises, for instance due to multiple blocks being created in short succession, before the users found out about the other through the network. As the next block after this is likely to disambiguate which chain is longer, Bitcoin's original proposal reasons that chains will reconverge.

While the basic form of the Nakamoto consensus functions with a fixed difficulty for its proof-of-work, Bitcoin rightly anticipated that the work available for the system would fluctuate. As this would lead to an inconsistent frequency

[11] What "reliably" means is a point of great contention. This thesis assumes the semi-synchronous setting, which will be discussed in more detail in Subsection 2.4.5.

of blocks, Bitcoin instead varies the difficult of the proof-of-work at regular (spaced in terms of block, rather than time) intervals, adjusting it up or down depending on self-reported timestamps within these blocks. Bitcoin targets approximately 10 minutes between blocks and this mechanism has been adopted by most[12] proof-of-work based cryptocurrencies.

Informally, the security of Nakamoto consensus relies on a larger amount of honest proofs-of-work being performed. An adversary attempting to break consensus will want to create uncertainty about the true state of the ledger, which can be done if multiple chains of equal length are competing. The further back the last common ancestor is in these chains, the less about the ledger is certain. This situation of competing chains of equal length is called a *fork*, with the distance to the common ancestor being the fork's depth. While an adversary can easily be lucky in the proof-of-work process and create a shallow fork, a deep one is exponentially unlikely, as, if honest users have more mining power, they will eventually break any artificial tie and extend only the longer remaining branch, with the adversary being unable to maintain the other branch as a viable alternative.

Digital money. The usage of the ledger is mostly independent from the consensus mechanism itself in proof-of-work currencies. "Mostly", as the winner of the proof-of-work lottery is typically rewarded with funds in the digital currency built on top of the ledger, to incentivise participation in the consensus mechanism. Incentives are not the focus of this thesis (although they briefly feature in Chapter 4, due to a flaw in the incentives for the naive approach), however it is worth stating that rewarding block creation broadly achieves its goal or encouraging rational miners to behave honestly, provided no great financial interests lie in censorship, or reordering transactions[13]. The basic foundations of building a digital currency on a ledger is simple: A mapping of public keys to funds is maintained, with users being able to create transactions when they spend funds associated with their public key. These transactions indicate a new recipient public key and a fee they are willing to pay for the transaction, which is retrieved by the block creator. Each transaction is digitally signed, therefore ensuring only the owner of funds can spend them.

[12] All to my knowledge, although it would be a full-time occupation to survey the entire field.

[13] This is, of course, a strong assumption, although still weaker than the traditional "honest majority".

2.4.2 Security Properties

Bitcoin, being proposed and implemented without a formal security analysis and not falling into the existing notions of consensus, clearly achieved *something*. It took a while for cryptographers to catch up and articulate clearly the properties achieved in the Bitcoin ledger. The most notable of these formalisms is [GKL15], which defines three properties of Nakamoto consensus: *chain growth*, *chain quality*, and *common prefix*. While the analysis of [GKL15] was of a simplified version of Bitcoin, assuming fixed difficulty and a synchronous network, these results have since been extended [GKL17].

Definition 2.3 (Chain Growth). For parameters $s, \gamma \in \mathbb{N}$, if at time t the honest party ψ has selected a chain of length c, then at time $t + s$, ψ will have selected a chain of length at least $c + \gamma$.

Definition 2.4 (Chain Quality). For parameters $l, \mu \in \mathbb{N}$ and any honest party's selected chain, any consecutive sequence of l blocks in the chain will include at least μ blocks created by an honest party.

Definition 2.5 (Common Prefix). For the parameter $k \in \mathbb{N}$: Given the current chains $\mathcal{C}_1$ and $\mathcal{C}_2$ of two honest parties ψ_1 and ψ_2 at the same point in time, removing the last k blocks from one chain ensures it is a prefix of the other: $\mathcal{C}_1^{\lceil k} \prec \mathcal{C}_2$, where $\mathcal{C}^{\lceil k}$ denotes the chain $\mathcal{C}$ without the k last blocks.

Combined, these properties achieve the broader goals of *persistence* and *liveness*, essentially stating that a confirmed transaction will remain confirmed and that all new transactions will become confirmed eventually.

Definition 2.6 (Persistence). For the parameter $k \in \mathbb{N}$, any transaction more than k blocks from the end of the chain of an honest party will be found at the same location in all honest parties ledgers and will remain there in future.

Definition 2.7 (Liveness). For parameters $u, k \in \mathbb{N}$, any valid transaction broadcast to all honest parties will, after u units of time, be reported as k blocks deep in their local chain by all honest parties.

Combined, persistence and liveness are sufficient to construct a digital currency, and [GKL17] demonstrated that the (simplified) Bitcoin ledger achieves these notions for reasonable parameters (although k may be much longer than most users expect for typical security parameters).

2.4.3 Composability

As distributed ledgers are used as a basis for larger systems, such as currency systems and smart contracts, it is crucial that their security proofs still hold in the context of their greater environment. Initial attempts at simulation-based models of distributed ledgers [CGJ+17] were too powerful, as they provided instantaneous consensus, more akin to the traditional byzantine fault-tolerance notions than the "Nakamoto-style" distributed setting.

A comprehensive modelling of Bitcoin in the universal composability setting was later achieved by [BMTZ17], although the model had many complex parameterisations: A function describing how blocks are constructed from transactions, a function restricting how the adversary is permitted to behave and complex machinery to track the passage of time, to name a few. It is because of this complexity that the work in this thesis is based on a series of smaller, simpler ideal descriptions of ledgers – unlike [BMTZ17], this thesis proposes new constructions, rather than modelling existing ones, so a simpler model is more appropriate. These simplified ledgers are presented here, as their modelling is not a primary contribution to the thesis and they are used throughout the rest of the document. The distributed ledger can be modelled to differing levels of granularity, depending on the amount of information needed.

2.4.3.1 The Simplified Ledger

The simplified ledger captures the essence of the traditional persistence property of ledgers, although it does not capture liveness. Any user may post transactions, which are deemed unconfirmed. The adversary may decide when and which unconfirmed transactions to move to an append-only ledger and may decide how long a prefix of this ledger honest parties see – provided it does not remove anything previously revealed to them.

While the liveness property is not captured by this ledger, due to the large amount of adversarial control, it is straightforward to see – although we will not demonstrate it here – that more complex ledgers, such as those defined in [BMTZ17, BGK+18], UC-emulate $\mathcal{G}_{\mathsf{SimpleLedger}}$, as defined next. In particular, this means that if replaced in the ideal world with such a ledger, which *does* have the liveness property, we also in practice have liveness for our protocol.

Functionality $\mathcal{G}_{\text{SimpleLedger}}$

The simplified interface to $\mathcal{G}_{\text{Ledger}}$ is strictly less powerful than actual ledger implementations, allowing reasoning about a less complex ledger functionality.

State variables and initialisation values:

Variable	Description
$\Sigma \coloneqq \varepsilon$	Authoritative ledger state
$M \coloneqq \lambda\psi.\varepsilon$	Mapping of parties to ledger states

When receiving a message (SUBMIT, tx) *from a party ψ:*

```
// The adversary is not required to ever put
// transactions on the ledger.
// Where it doesn't, the execution is unlikely
// to be interesting, however.
```

query $\mathcal{A}$ **with** (TRANSACTION, tx)

When receiving a message READ *from a party ψ:*

if $\psi = \mathcal{A}$ **then return** Σ

else return $M(\psi)$

When receiving a message (EXTEND, Σ') *from $\mathcal{A}$:*

let $\Sigma \leftarrow \Sigma \parallel \Sigma'$

When receiving a message (ADVANCE, ψ, Σ') *from $\mathcal{A}$:*

if $M(\psi) \prec \Sigma' \prec \Sigma$ **then let** $M(\psi) \leftarrow \Sigma'$.

2.4.3.2 The Delay Ledger

While the simplified ledger $\mathcal{G}_{\text{SimpleLedger}}$ is nice for the analysis of protocols building on it as a global functionality, in practice users would like to take advantage of some liveness guarantees. $\mathcal{G}^{\delta}_{\text{DelayLedger}}$ annotates transactions with a time at which they were received. This time is never returned to parties, however it asserts that every party can see a transaction, once it is δ time slots in the past. This ledger operates under the assumption of a global clock $\mathcal{G}_{\text{clock}}$, described earlier in Subsection 2.3.5. We posit without proof that $\mathcal{G}^{\delta}_{\text{DelayLedger}}$ UC-emulates $\mathcal{G}_{\text{SimpleLedger}}$, by virtue of the latter having far greater adversarial power. This ledger can also be constructed using existing UC-secure ledgers such as [BMTZ17, BGK+18], as these aim to provide the same guarantees.

Functionality $\mathcal{G}^{\delta}_{\text{DelayLedger}}$

The δ-delay ledger adds liveness guarantees to $\mathcal{G}_{\text{SimpleLedger}}$, ensuring that sufficiently old transactions are always visible to honest parties.

State variables and initialisation values:

Variable	Description
$\Sigma \coloneqq \varepsilon$	Authoritative ledger state
$M \coloneqq \lambda\psi.\varepsilon$	Mapping of parties to ledger states
$U \coloneqq \varnothing$	Multiset of unconfirmed transactions

When receiving a message $(\textsc{submit}, \mathsf{tx})$ *from a party* ψ*:*

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t

let $U \leftarrow U \cup \{(\mathsf{tx}, t)\}$

query $\mathcal{A}$ **with** $(\textsc{transaction}, \mathsf{tx}, t)$

When receiving a message READ *from a party* ψ*:*

assert liveness

return $\mathsf{map}(\mathsf{proj}_1, M(\psi))$

When receiving a message $(\textsc{extend}, \Sigma')$ *from* $\mathcal{A}$*:*

if $\Sigma' \subseteq U$ **then**

let $U \leftarrow U \setminus \Sigma'$

let $\Sigma \leftarrow \Sigma \parallel \Sigma'$

When receiving a message $(\textsc{advance}, \psi, \Sigma')$ *from* $\mathcal{A}$*:*

if $M(\psi) \prec \Sigma' \prec \Sigma$ **then**

let $M(\psi) \leftarrow \Sigma'$.

Helper procedures:

procedure liveness

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t

if $\exists(\mathsf{tx}, t') \in U : |t - t'| > \delta$ **then return** $\bot$

else if $\exists(\mathsf{tx}, t') \in \Sigma : |t - t'| > \delta \wedge \exists\psi \in \mathcal{H} : (\mathsf{tx}, t') \notin M(\psi)$ **then return** $\bot$

else return $\top$

2.4.3.3 The Nakamoto Ledger

The basic functionality of this ledger allows the submission of transactions and retrieving each of the following:

- A confirmed prefix of the ledger state.
- A "projection" of the ledger state – that is, what the local state will approach, if there is no chain reorganisation.
- The confirmed "leader state", which models the mechanism used for the SRS generation.

When any of these values is queried, the functionality ensures that liveness and chain quality properties still hold. The adversary further has the power to instruct the creation of a new block, on behalf of any party, and to instruct any party to adopt a different chain. In both cases, the functionality ensures that the common prefix property is preserved. The adversary has full control over the contents of both honest and adversarial blocks, as well as their order.

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

A ledger following a Nakamoto-style consensus, with each party having a *projected* chain, a prefix of which is common to all parties. Common prefix, chain quality and chain growth are guaranteed.

State variables and initialisation values:

Variable	Description
$\Pi \coloneqq \psi \mapsto \varepsilon$	Mapping of parties to projected ledger states
$T \coloneqq \varnothing$	Multiset of submitted transactions
$\mathsf{hon} \coloneqq \varnothing$	Mapping of block ids I if they are honest, or O

When receiving a message (SUBMIT, tx) *from a party ψ:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

let $T \leftarrow T \cup \{(\mathsf{tx}, t)\}$

query $\mathcal{A}$ **with** (TRANSACTION, tx, t)

When receiving a message READ *from a party ψ:*

assert $\mathsf{liveness}(\psi) \wedge \mathsf{chainQuality}(\psi)$

return $\mathsf{map}(\mathsf{proj}_1, \mathsf{txs}(\Pi(\psi)^{\lceil k}))$

When receiving a message PROJECTION *from a party ψ:*

assert $\mathsf{liveness}(\psi) \wedge \mathsf{chainQuality}(\psi)$

return $\mathsf{map}(\mathsf{proj}_1, \mathsf{txs}(\Pi(\psi)))$

When receiving a message LEADER-STATE *from a party ψ:*

assert $\mathsf{liveness}(\psi) \wedge \mathsf{chainQuality}(\psi)$

let $\vec{a} \leftarrow \mathsf{map}(\lambda(\cdot, a, \cdot, t).\,(a, t), \Pi(\psi)^{\lceil k})$

return $\mathsf{foldl}(\mathsf{Apply}, \varnothing, \vec{a})$

When receiving a message $(\textsc{extend}, \psi, B, t, a)$ *from* $\mathcal{A}$*:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t'

let $\mathsf{id} \xleftarrow{*} \{0,1\}^{\kappa}$

if $\psi \in \mathcal{H}$ **then**

let $\vec{a} \leftarrow \mathsf{map}(\lambda(\cdot, a, \cdot, t).\,(a, t), \Pi(\psi))$

let $\sigma \leftarrow \mathsf{foldl}(\mathsf{Apply}, \varnothing, \vec{a})$

let $a \xleftarrow{*} \mathsf{Gen}(\sigma, t')$

let $t \leftarrow t'$ **let** $\mathsf{hon}(\mathsf{id}) \leftarrow 1$

else

let $\mathsf{hon}(\mathsf{id}) \leftarrow 0$

if $t' < t$ **then let** $t \leftarrow t'$

else if $\exists t''.\,(\cdot, \cdot, \cdot, t'') = \mathsf{last}(\Pi(\psi)) \wedge t'' > t$ **then let** $t \leftarrow t''$

let $\Pi(\psi) \leftarrow \Pi(\psi) \parallel (B, a, \mathsf{id}, t)$

assert $\forall \psi' \in \mathcal{P}.\,\Pi(\psi)^{\lceil k} \prec \Pi(\psi')$

return (B, a, id, t)

When receiving a message $(\textsc{advance}, \psi, \Sigma')$ *from* $\mathcal{A}$*:*

assert $\exists \psi' \in \mathcal{P}.\,\Sigma' \prec \Pi(\psi')$

assert $\forall \psi' \in \mathcal{P}.\,\Sigma'^{\lceil k} \prec \Pi(\psi') \wedge \Pi(\psi')^{\lceil k} \prec \Sigma'$

let $\Pi(\psi) \leftarrow \Sigma'$

Helper procedures:

function $\mathsf{txs}(\Pi_\psi)$

let $\vec{B} \leftarrow \mathsf{map}(\mathsf{proj}_1, \Pi_\psi)$

return $\mathsf{concat}(\vec{B})$

procedure $\mathsf{liveness}(\psi)$

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

if $\exists t_0 < t.\,|[\ t_b \mid (\cdot, \cdot, \cdot, t_b) \in \Pi(\psi), t_0 - s \le t_b < t_0\ ]| < \gamma \wedge t_0 - s \ge 0$ **then**

return $\perp$

return $\forall (\mathsf{tx}, t') \in T.\,t' + \lceil (l+k)\gamma^{-1} \rceil s > t \ \vee (\mathsf{tx}, t') \in \mathsf{txs}(\Pi(\psi)^{\lceil k})$

procedure $\mathsf{chainQuality}(\psi)$

let $\vec{\mathsf{id}} \leftarrow \mathsf{map}(\mathsf{proj}_3, \Pi(\psi)^{\lceil k})$

return $\forall i \in \mathbb{Z}_{|\vec{a}|-l}.\,\left(\sum_{j \in \mathbb{Z}_l} \mathsf{ids}(\vec{\mathsf{id}}_{i+j})\right) \ge \mu l$

2.4.3.4 Commutativity of Ledger Realisations

It is of note that since the ledger exists in both the ideal and real world, we would ideally wish to be able to utilise the stronger δ-delay ledger (and others) in the ideal world as well. This is not trivial, however – the UC proof presented in this paper holds for the simple ledger, and while the transitivity and composability of UC proofs implies that the simple ledger can be replaced by the stronger ledger in the real world, this is not the only goal.

In order to enable ideal-world replacement, we consider when UC replacements are commutative. Specifically, consider we have four functionalities, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$, such that: a) $\mathcal{A}$ and $\mathcal{B}$ both have $\mathcal{C}$ as a global functionality, b) $\mathcal{A}$ is UC-emulated by $\mathcal{B}$ with the simulator $\mathcal{S}_{\mathcal{B}}$, and c) $\mathcal{C}$ is UC-emulated by $\mathcal{D}$ with the simulator $\mathcal{S}_{\mathcal{D}}$. Observe that this is a generalisation of our situation, where $\mathcal{A}$ is $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$, $\mathcal{B}$ is Kachina, $\mathcal{C}$ is $\mathcal{G}_{\text{SimpleLedger}}$, and $\mathcal{D}$ is some other ledger $\mathcal{G}_{\text{L}}$.

When can we conclude that $\mathcal{A}$ is still UC-emulated by $\mathcal{B}$ even if the global functionality is replaced by $\mathcal{D}$ in both worlds? That is, when can we perform the inner UC-replacement *first* and still be able to perform the outer one?

Theorem 2.3. *Given $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{S}_{\mathcal{B}}$, $\mathcal{S}_{\mathcal{C}}$ as defined above, if $\mathcal{S}_{\mathcal{B}}$ forwards all adversarial queries to $\mathcal{C}$ unchanged and makes no queries to $\mathcal{C}$, then $\mathcal{A}$ is UC-emulated by $\mathcal{B}$ with the global functionality $\mathcal{D}$ in place of $\mathcal{C}$.*

Proof. We will provide this proof largely visually. The environment has a number of actions it can perform in any given world, in tandem with the dummy adversary. We will represent these as unconnected wires in a circuit representation of the different UC functionalities. Each wire is coloured in accordance with its purpose; these colours serve only to differentiate the wires. We visualise the preconditions of Theorem 2.3 in Figure 2.1 and Figure 2.2. This crucially includes the precondition that $\mathcal{S}_{\mathcal{B}}$ forwards adversarial queries to $\mathcal{C}$, which is represented equivalently by these queries being made directly instead.

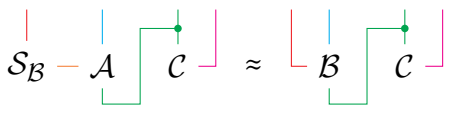


Figure 2.1: The first part of the precondition: $\mathcal{B}$ UC-emulates $\mathcal{A}$.

By the UC emulation theorem, for all environments, executions with the simulator and the ideal-world protocol are equivalent to executions with the real-

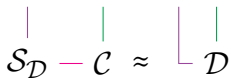


Figure 2.2: The second part of the precondition: $\mathcal{D}$ UC-emulates $\mathcal{C}$.

world protocol. Due to the all-quantification of the environment, we can replace any part of a circuit diagram which matches exactly one of the two sides of the equivalence with the other – this is the foundation of the compositionality in UC.

We first make use of this in the non-standard direction, of making our ideal-world protocol *more abstract*. Specifically, replace the right-hand side of Figure 2.2 with the left. We start similarly to the left-hand side of Figure 2.1, however using $\mathcal{D}$ instead of $\mathcal{C}$. Visually, Figure 2.3 demonstrates the result, which includes two independent simulators.

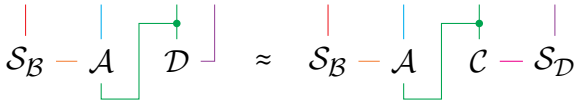


Figure 2.3: Visual equivalence for idealising the sub-protocol $\mathcal{D}$.

From here, we can realise both the ideal-world functionalities, provided it is in the correct order: We must first realise $\mathcal{A}$, as it relies on the presence of $\mathcal{C}$. We can directly apply the equivalence of Figure 2.1, as can be seen in Figure 2.4.

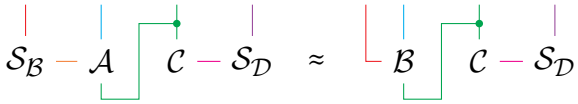


Figure 2.4: Visual equivalence for substituting the main protocol $\mathcal{B}$.

Finally, nothing stands in the way of realising $\mathcal{C}$ with $\mathcal{D}$, using the equivalence of Figure 2.2 again, this time in the more typical direction. As a result, we get in Figure 2.5 the final step, leading us to the intended equivalence and proving the related UC-emulation statement. □

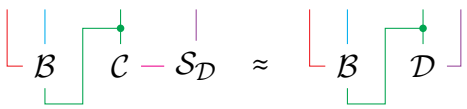


Figure 2.5: Visual equivalence for re-substituting $\mathcal{D}$.

2.4.4 Ouroboros and Proof-of-Stake

As the theory of proof-of-work quickly gave rise to a practice of expensive, purpose built machines dedicated only to "useless" work (not fully useless, but with the purpose only to establish a distribution of power), the question quickly became apparent as to whether this wastefulness was necessary. Proof-of-stake was initially proposed as PPCoin [KN12], which later evolved into the Peercoin cryptocurrency. As with Bitcoin, the early designs' implications were poorly understood, and initially there was a great deal of skepticism that proof-of-stake was possible to do securely.

In the years since, a great deal of research on proof-of-stake has been carried out. Two main streams of securely performing proof-of-stake have emerged from this: First, "Nakamoto-style" proofs of stake, where a randomised process determines which user will be able to create blocks and allows this user to extend a chain, and second, Byzantine fault-tolerant (BFT)-style ledgers, where a randomised process elects a representative committee of users, who perform a traditional BFT consensus protocol to decide on the order of transactions. Examples of the former are "ad-hoc" constructions such as Peercoin and the Ouroboros family of protocols, and examples of the latter are Snow White [DPS19] and Algorand [GHM+17]. This thesis will focus on the Ouroboros family specifically, as Chapter 5 focuses on implementing a privacy-preserving variant of Ouroboros.

The Ouroboros family of proof-of-stake. The Ouroboros family of proof-of-stake protocols consists of the original proof-of-stake design, named just Ouroboros [KRDO17], and various incremental improvements, of which Ouroboros Praos [DGKR18] and Ouroboros Genesis [BGK+18] are relevant to this thesis. This basic design of Ouroboros divides time into lengthy periods across which stake is considered (mostly) stable, called *epochs*, which is subdivided into short periods representing network synchronisation time, called *slots*. At the start of each epoch, the distribution of users' stake at a predefined point during (but not at the end of!) the previous epoch is taken as the distribution according to which new block creators are to be selected.

Choosing a point not at the end of the epoch mitigates an attack known as *grinding*, where an adversary repeatedly sends funds to themselves, or regenerates a public key, until they get one which is eligible to create a block [Jut12].

Separately, during the epoch a randomness generation process takes place, producing the randomness used to sample from this distribution. While the details of the randomness generation and selection process differ, the result is that each slot is assigned a user who is permitted to create a block at this time in the original Ouroboros protocol, with Praos and Genesis differing only in that a slot may have multiple or no eligible block creators (or "leaders").

The original Ouroboros [KRDO17] design used secure multi-party computation [GMW87, CLOS02] to compute randomness at the end of each epoch. This randomness was completely unbiased and is then used to sample one leader for each slot in the next epoch. The cost of the multi-party computation was a limiting factor in this design and the main motivation behind the changes in Ouroboros Praos [DGKR18]. Instead, this relied on each block creator embedding in the block a predetermined random value, dependant on their secret key, the slot number, and the epoch randomness. This is achieved using the primitive of *Verifiable Random Functions* (VRFs), which allow other users to verify that the randomness is indeed associated with the correct input and secret key.

At the end of the epoch, randomness provided by the blocks of this epoch is aggregated, up to a point before end of the epoch to again protect from grinding. A probability analysis shows that this is equivalent to a "clean" source of randomness as in Ouroboros, which the adversary has the opportunity to *reset* a fixed number of times: The adversary can roll the dice again a few times, for instance by withholding its own contribution to the randomness. This influence gives it negligible impact over the distribution of blocks in the next epoch however, not compromising overall security.

In addition to the changes to randomness selection, Praos has a modified process for selecting slot leaders: Instead of publicly sampling a leader for each slot, Praos relies on a process more similar to the probabilistic self-election of proof-of-work. The VRF primitive is also used (in a separate instance) to determine if a user is eligible for creating a block in a specific slot. Each slot, a user evaluates the VRF given the epoch randomness and slot number. If the output value is lower than a threshold which depends on both a difficulty parameter and the proportion of stake the user holds, they are eligible to create the block, with the VRF output authenticating their right to this. In particular this approach enables moving the network model from a synchronous one to being semi-synchronous, the difference of which will be discussed in Subsection 2.4.5.

Ouroboros Genesis [BGK+18] moves from the property-based results of the original Ouroboros and Praos to simulation-based results in the UC framework. In addition to this change, it addresses an issue in proof-of-stake for users newly joining the protocol: Given only the initial distribution of stake, it is possible to create believable, yet false histories, by including only transactions in an alternate history which transfer funds to the adversary. This attack, known as *stake bleeding* [GKR18], or a long range attack, requires that participants are fixed and active in many proof-of-stake protocols, including Ouroboros and Ouroboros Praos. Ouroboros Genesis addresses this deficit by changing the conditions under which new chains are adopted: Rather than only adopting the longest chain, Ouroboros Genesis requires the adopted chain to be sufficiently active *throughout* its history. Specifically, a chain which grows faster *at the point of divergence* is preferred over a slower growing chain. This makes stake bleeding attacks infeasible, as the fork necessarily takes place at a point where the adversary has a minority of stake.

Forkable strings. The core of the security proof of each of the Ouroboros protocols is a stochastic analysis of randomly distributed strings of selected leaders: In Ouroboros, these strings are binary, with ones representing adversarial slots and zeros honest slots, sampled from independant Bernoulli distributions, biased towards honest parties. In Ouroboros Praos and Genesis, as multiple or no user may be elected to any slot, the additional symbol $\perp$ is introduced to denote slots without a leader, and slots with multiple leaders are treated as adversarial. The forkable strings theorem is restated here, with the following caveats: $\mathsf{div}_0(w)$ is not formally presented, though it can informally be thought of as the distance to the latest common ancestor of the longest possible fork. Further, the notation is asymptotic, using exp and Ω notations. Informally, it becomes exponentially unlikely for a fork longer than k to be possible, as the length of the string grows.

Lemma 2.1 (Theorem 3 of [DGKR18]). *Let $\ell, k \in \mathbb{N}$, and $\varepsilon \in (0,1)$. Let $w \in \{0,1\}^\ell$ be sampled from ℓ independant Bernoulli trials, where $\Pr[w_i = 1] = (1-\varepsilon)/2$. The $\Pr[\mathsf{div}_0(w) \geq k] \leq \exp(\ln \ell - \Omega(k))$.*

Lemma 2.1 is combined with a proof that the asynchronous strings (i.e. in $\{0,1,\perp\}^*$) can be safely reduced to synchronous ones, accounting for a maximal network delay Δ.

2.4.5 Assumptions: Network and Honesty

Some of the cryptographic assumptions required for distributed ledgers were already briefly covered in Section 2.2. Two additional assumptions are necessary however: First, a broadcast network is needed, both to broadcast transactions and to broadcast blocks in order to perform consensus (this is assuming a Nakamoto-style consensus mechanism – BFT-style algorithms may additionally require point-to-point channels between the committee members). Second, the assumption of honest majority of either computational power, or stake needs to be made formally. In particular in the composable setting with adaptive security, this restricts *how* the adversary can corrupt. This is primarily relevant in the proof-of-stake case, as in proof-of-work corrupting a party does not necessarily inherit their computational power.

Modelling the network. The most basic form of broadcast network delivers any message to all participants immediately. This is clearly too strong a notion: Real networks have delays, and a network this powerful already almost achieves consensus: Everyone simply observes the messages in order, and treats this as final (this assumes no messages are sent simultaneously, a possibility often deliberately unsupported by cryptographic frameworks to force modelling temporal uncertainty as adversarial influence). The next relaxation of broadcast is to assume delivery after some fixed time, or after a *round*. Cryptography tends to discretise time, with the (assumed) delay of communication often dictating the units. This setting is often referred to as *synchronous*, as every user still receives a message at the same time, although there is not necessarily a guarantee of order for messages submitted in the same round. Ouroboros [KRDO17] was initially proven in the synchronous setting.

The synchronous setting has the disadvantage of relying on the delay to be an *upper bound*. If at any point a message takes longer than this delay to deliver, the security proof is invalidated. This is in contention with a desire for protocols, and consensus in particular, to be as *responsive* as possible, and therefore for the network delay to be set as low as possible. In the real world, network delays are likely to be distributed according to an exponential distribution – packets having a probability of successful transmission on each attempt, with repeated attempts on timeout. It is tempting to set the network delay to the median latency and

throughput, which often assumes no retransmissions.

While a network-level adversary could do worse, even simple network-level attacks such as denial-of-service can vastly increase network delays, making it paramount that they are considered. A fully asynchronous network is not an option either, as the delay cannot be arbitrary for consensus to be achieved. Instead, the *semi-synchronous* setting used in Ouroboros Praos [DGKR18] does have a maximum network delay, but also divides time into smaller units than this, and allows messages to be delivered in a time window between when they are sent and the maximum delay. This allows protocols which operate optimistically, moving faster under good network conditions, but which still have a hard fallback when network delays are increased.

A further power which can be afforded to the adversary is selectively broadcasting messages, or delaying messages partially. Also referred to as *multicasting*, this allows the adversary to present different chains to different users, an attack which assumes a fair amount of network control or good timing. Further, as privacy is the focus of this thesis it is important to stress that anything *broadcast* over a network will be seen by a powerful adversary immediately. Although it makes no difference for some of the results of this thesis, Chapter 6 assumes *sender anonymity* of the transaction broadcast, that is, that a transaction cannot be linked to the user that posted it. This is a powerful requirement, however anonymity is unachievable without it on distributed ledgers. In practice, it can be approximated through techniques such as onion routing [CL05], or mix networks [JJR02], although the former is vulnerable to powerful adversaries and the latter is costly to operate at scale. Formally, we use the following UC hybrid functionality:

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

The sender-anonymous multicast network $\mathcal{F}_{\mathsf{Net}}$ permits any party to broadcast messages to any other. These messages will be delivered within a maximum delay of δ, measured through $\mathcal{G}_{\mathsf{clock}}$. The adversary can deliver messages faster, and can send messages to individual parties. Messages are erased after they are read.

State variables and initialisation values:

Variable	Description
$M \coloneqq \lambda\psi.\varepsilon$	Mapping of parties to messages
$P \coloneqq \lambda\psi.\varnothing$	Mapping of parties to messages pending delivery

When receiving a message (BCAST, m) *from a party* ψ*:*

query $\mathcal{A}$ **with** (BCAST, m)
send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t
let $P(\psi) \leftarrow P(\psi) \cup \{(m, t)\}$
return $\top$

When receiving a message READ *from a party* ψ*:*

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t
for (m, t') **in** $P(\psi)$ **do**
  if $t' + \delta \leq t$ **then abort**
let res $\leftarrow M(\psi); M(\psi) \leftarrow \varepsilon$
return res

When receiving a message (TARGET, ψ, m) *from* $\mathcal{A}$*:*

if $\exists t$: $(m, t) \in P(\psi)$ **then**
  let $P(\psi) \leftarrow P(\psi) \setminus \{(m, t)\}$
let $M(\psi) \leftarrow M(\psi) \parallel m$
return $\top$

Requiring honest majority. How to formally require honest majority differs between proof-of-stake and proof-of-work protocols. In (random oracle based) proof-of-work, a comparatively straightforward approach may be adopted: The random oracle is specified to restrict the number of queries each party can make in a single round, and further limits the adversary to performing fewer queries than honest parties. Proof-of-stake is more complex, as it is related to the corruption model. Once the adversary adaptively corrupts any party, it obtains this parties funds (more precisely, it obtains the parties state and identity, which is sufficient to control the funds). The adversary can attempt to corrupt a party even if this results in it violating the honest majority assumption.

The simplest attempt to circumvent this would be to forbid such corruptions, however as corruptions are a part of the fundamental security model in the universal composition framework[14], Ouroboros Genesis [BGK+18] instead permitted them, but ensured that the security statement is trivial if honest majority is violated. Specifically, the assumption is encoded as a *wrapper* around the VRF functionality and the network. The wrapper observes network traffic to con-

[14] Rather: They were at the time Ouroboros Genesis was written. Newer version of UC are more flexible in their corruption models.

clude if honest majority is violated, and if it is, refuses to carry out any adversarial evaluations of the VRF. While the VRF functionality itself can be constructed, using the wrapped variant instead is done by assumption – the assumption that this violation never occurs. If it does, the adversary effectively shoots themselves in the foot, as its ability to create block *at all* is revoked. A similar approach is taken in Chapter 5, although the wrapper applies to performing zero-knowledge proofs and retrieves the stake distribution by observing the inputs to these proofs.

2.4.6 Limitations

When discussing distributed ledgers and their usage, it is important to bear in mind that they are not a universal solution. Two major issues limit their application, the first of which being a lack of privacy. As the topic of this thesis, this should come as little surprise, but it is important to note that the lack of privacy is (to some level) inherent in the idea of a distributed ledger as a form of consensus among parties. Users must know the data they are agreeing on, even if they need not know its meaning. This enables some usage of cryptography to hide meaning, but only in so far as it is isolated and does not affect the rest of the system. The trade-offs here are discussed in detail in Chapter 6, which focuses on providing a reasonable basis for privacy in the general case of smart contracts.

The second limitation is not the focus of this thesis, but is important to understand as it indicates the direction of this field of study: Distributed ledgers are limited in their scalability. Because of the setting of mutual distrust, greater participation in the consensus mechanism does not lead to less computational burden on each participant, but if anything to more, as each user must interact with more people. Given more users means more information flowing into a distributed ledger, the effort required to maintain this ledger rises linearly for each participant as it grows. During its peak of interest in late 2017, costs of transactions rose dramatically, and the latency of using many blockchains slowed to a crawl due to limited transaction throughput. A lot of research has been done over the years to increase throughput, including Bitcoin-NG [EGSvR16] and a variant of Ouroboros [FGKR20]. Proposals were made to split networks [WSNH19], although this dramatically weakens the security of each part.

To bypass this limitation, layer 2 solutions [DFH18, KL20] optimistically carry out computation off-chain. This must be something which can safely be detached from the primary, on-chain consensus, such as exchanging funds between each other. Typically this involves *locking* the on-chain state in some way, for instance setting aside a fixed amount of funds for trading off-chain in a payments channel. Only in the case that the users wish to end their relationship, either agreeably or in a dispute, is the distributed ledger invoked again and provided with evidence of what happened.

If distributed ledgers become a commonly used tool, their fate is likely to become a decentralised court – mediating disputes, rather than managing micropayments. In this context privacy becomes of even greater importance: What happens off-chain is often protected by partial trust, and can more easily be subjected to powerful cryptography, such as secure multi-party computation. The on-chain resolution is the place where care needs to be taken, and in order for this to be possible in the broadest cases, a good foundation for privacy is needed.

2.5 Zero-Knowledge

Zero-knowledge proofs[15] are an advanced and powerful cryptographic primitive with many applications, specifically to ensuring privacy. The core idea is difficult to convey to a layperson without it sounding like magic: Zero-knowledge proofs allow one person to convince another that a statement is true, without revealing any further information than the fact *that it is true*. This is counter-intuitive as humans have grown used to demonstrating truth through transparency: We give the tax-man our records and their consistency confirms our workings. This is epitomised in the infamous argument "*If you have nothing to hide, you have nothing to fear*", a statement which presupposes a proof of innocence cannot coexist with privacy.

The idea of zero-knowledge also sounds like magic as its limitations are not well-defined. Can a zero-knowledge proof convince someone of a person's trustworthiness, or desire to do good? Clearly not, as the terms themselves are subjec-

[15] Or zero-knowledge arguments, as they are sometimes referred to. The term argument is used for proofs whose soundness relies on computational assumptions, distinguishing them from mathematical, or perfectly correct, proofs.

tive, but in a similar vein they cannot prove who is the legitimate owner of a piece of land, or that a parcel is within its weight limits. Zero-knowledge proofs are mathematical objects and can reason only about mathematics. Specifically, they can make statements about the properties of information. A zero-knowledge proof may prove that a prime exists between 15 and 20 (in this case, it should not be too hard to find), or that a message, when decrypted, contains a specific phrase.

A proof of knowledge additionally demonstrates that the person doing the proving *knows* the information which a property is asserted about. They should know of the primality of either 17 or 19. They should know the decrypted message – and the key, to demonstrate how it relates to the ciphertext.

Subsection 2.5.1 introduces the mathematics behind zero-knowledge proofs, and the common properties they satisfy. The main theoretical object used in this thesis, the non-interactive zero-knowledge proof (or NIZK), is introduced in Subsection 2.5.2, with their most interesting instantiation, the SNARK, being introduced in Subsection 2.5.3. Finally, the universality and updateability features of a select set of SNARKs are discussed in Subsection 2.5.4.

2.5.1 Definition and Sigma Protocols

The basis of a zero-knowledge proof is a relation $\mathcal{R}$, which encodes the information the proof wishes to convey. For instance, in our bounded prime example, the following relation formally describes the problem:

$$((a, b), p) \in \mathcal{R} \iff a < p < b \wedge p \text{ is prime}$$

The left-hand side of zero-knowledge relations can be considered public, while the right-hand side should have no information revealed about it. If Alice proves to Bob that $\exists p\colon ((15, 20), p) \in \mathcal{R}$, Bob should not be able to determine if Alice was using $p = 17$ or $p = 19$, and for other problems, should not be able to determine *any* satisfying value.

The left-hand side of a relation is often denoted by x and referred to as the *statement*. Correspondingly, the right-hand side is denoted by w and referred to as the *witness*. A specific protocol for zero-knowledge proofs applies only for specific types of relations $\mathcal{R}$, although protocols for NP-complete relations allow proving solutions to all problems expressible in NP. Given efficient primality tests, the simplistic primality relation above can also be realised.

Prover		Verifier
$r \xleftarrow{*} \mathbb{F}_q^*$	$\xrightarrow[\text{(commitment)}]{t \coloneqq g^r}$	$c \xleftarrow{*} \mathbb{F}_q^*$
	$\xleftarrow[\text{(challenge)}]{c}$	
	$\xrightarrow[\text{(response)}]{s \coloneqq r + cx}$	**output** $g^s = th^c$

Figure 2.6: The Schnorr proof of knowledge of discrete logarithm sigma protocol.

Many zero-knowledge proofs rely on an interactive proof approach of commitment challenge and response. In the first step, the prover commits to an input, either the witness, or randomness, and transmits this to the verifier. The verifier selects a random challenge and sends this to the prover. The prover must then generate a response depending on both the challenge and the witness – one which must convince the verifier that the response could not have been generated without it satisfying the relation.

The most well-known such protocol (the structure of which is also collectively called a Σ-protocol, or sigma protocol, due to its three-round structure matching the strokes of the Greek letter), is a proof of knowledge of a discrete logarithm from Schnorr [Sch90], which serves as a good example. In this case, for a group $\mathbb{G}$ of prime order q with generator g, the relation is:

$$(h, x) \in \mathcal{R} \iff h = g^x.$$

Using $\mathbb{F}_q^*$ for the multiplicative field modulo q, i.e. $\mathbb{Z}_q \setminus \{0\}$, Schnorr's protocol is described in Figure 2.6. This protocol achieves three properties crucial for zero-knowledge proofs: Correctness, soundness, and zero-knowledge, which are sketched informally here (as this thesis will rely on composable zero-knowledge definitions, rather than these properties directly).

A note on rewinding. Schnorr's security is commonly expressed using *rewinding* – where the proof itself involves (as the name suggests) rewinding the protocol execution and observing an alternate execution with different randomness from the same point on. Although this approach is common for formalising sigma protocols, we explicitly avoid it in this thesis, in favour of *white-box, straightline extraction*. This requires additional assumptions, such as the Algebraic Group Model [FKL18], a powerful knowledge assumption. The primary reason

for this is this thesis' use of compositional security, which does not work well with rewinding proof techniques.

- **Correctness.** The correctness property states that with overwhelming probability, an honest prover will convince an honest verifier. Note that as $g^{r+cx} = g^r(g^x)^c = g^r h^c$, Schnorr's protocol is *perfectly* correct.

- **Soundness**[16]**.** The soundness property requires that a prover who does not know the witness cannot convince the verifier. In Schnorr's proof, for $g^s = g^r h^c$ to hold, $s = r + cx$ must also hold. A prover able to generate such an s can also solve for x, thereby "knowing" it.

- **Zero-knowledge.** The zero-knowledge property states that the verifier learns no information about the witness, except that it satisfies the relation. In the case of sigma protocols, the more relaxed *honest-verifier zero-knowledge* property is often used, which requires that the verifier adheres to the protocol (in the case of Schnorr, c must be distributed according to $\mathbb{F}_q^*$). In practice, this is demonstrated using a simulator, which must be able to generate a transcript of interactions indistinguishable from a real one, given the statement x. Given that the simulator can sample c honestly, it can sample $s \xleftarrow{*} \mathbb{F}_q^*$, and compute $t \coloneqq g^s h^{-c}$. This produces a matching transcript without knowing the discrete logarithm of h.

2.5.2 Non-Interactive Zero-Knowledge

The interactive nature of sigma protocols does not suit the nature of distributed-ledger protocols, in which transactions are typically considered fire-and-forget. If the validity of Alice's transaction required an interactive query with Alice, not only does she need to remain online in case further people attempt to verify it, but worse she will need to repeat proving to each of them. As distributed ledgers rely on many verifiers for their security, this is impractical, even if one ignores the question of how to handle the potential disagreement if some users succeed in verification, but others fail to verify, for instance due to the prover being unavailable.

[16] In a rewinding based setting, *special soundness* typically states that the transcripts of two accepting runs, sharing the same initial message, can be combined to extract a witness.

To solve this issue, the interactive protocol can be constrained to a single round: The prover sends a single message to the verifier, who then either accepts this proof, or rejects it. Impossibility results [GO94] show that this type of protocol requires hybrid assumptions – either a common reference string, or a random oracle. Given these, constructing a non-interactive form of zero-knowledge is possible. This non-interactive zero-knowledge (or NIZK) no longer benefits from a distinction between prover and verifier – key is the content of the message that is sent, which is often referred to as a *proof* (although *argument* is more appropriate) and is denoted by π throughout this document. A NIZK consists of two algorithms, which have access to the same hybrid assumption:

- $\pi \xleftarrow{*} \mathsf{NIZK}_{\mathcal{R}}.\mathsf{Prove}(x, w)$ – Produces a valid proof π if and only if $(x, w) \in \mathcal{R}$.
- $b \leftarrow \mathsf{NIZK}_{\mathcal{R}}.\mathsf{Verify}(x, \pi)$ – Outputs 1 if π is honestly generated and may additionally do so only if knowledge of w can be extracted from the adversary such that $(x, w) \in \mathcal{R}$. In all other cases, output 0.

A fairly simple trick, known as the Fiat-Shamir transform (or heuristic[17]), transforms a sigma protocol into a non-interactive proof: Instead of asking the verifier for a random challenge, the random oracle is queried. As this challenge is equally unpredictable to a challenge coming from the verifier, it is possible to roll all three phases into one. Even outside of sigma protocols, the general approach of transforming an interactive, challenge-based protocol into a non-interactive one through use of random oracles is commonly used, forming one of the basis of SNARKs, which will be discussed in Subsection 2.5.3.

Black-box vs white-box extraction. The Fiat-Shamir transform is secure, however it often relies on white-box extraction assumptions or rewinding. Consider the example Schnorr protocol: The extraction is not black-box, as to compute x, it is necessary not only to know the parts s and c directly embedded in the proof, but it is also necessary to know r, which is internal to the prover and never broadcast.

The primary issue with this is that, in order to retrieve the witness for an adversary's proof, it is necessary to know *everything* about the adversary. In compo-

[17] It is heuristic when used with a hash function, while being statistically secure using a random oracle.

sition frameworks, this is too powerful a statement, as the "adversary" is the distinguishing environment, which also controls honest users' inputs. Chapter 3 discusses how this issue with white-box extraction can be resolved, however it is worth discussing the alternative.

Instead of white-box extraction, composition is possible when the extraction is black-box, that is, when it does not require knowledge of the internals of the prover. Fischlin's transform [Fis05] provides a generic replacement for the Fiat-Shamir transform, which in the random oracle model allows for black-box extraction. It cleverly shifts extraction toward the random oracle, by requiring n multiple independent proofs, and for each of these m multiple challenges to be answered. Of the answered challenges, the full sigma protocol transcript is again passed into the random oracle, and the smallest response's transcript is chosen to represent this one of the n proofs. This must fall under a threshold – practically requiring at least some of the m queries are actually made, with the n independent proofs preventing luck being the reason for passing the threshold in all of them. The upshot is that at least two transcripts for the same initial commitment will have been queried on the random oracle. These transcripts can be extracted and correlated to find the witness.

Composable Definition. Throughout this thesis, the below composable definition for non-interactive zero-knowledge will be used. It permits proof-malleability, that is, for a valid proof to be changed into a different proof of the same statement, and for ease of extraction, tracks witnesses. Proof-malleability is required for several real-world constructions, and does not impede the protocols presented in this thesis. It also tracks *disproven* statements – when it marks a proof as false, it will not later change this assessment.

Functionality $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$

The (proof-malleable) non-interactive zero-knowledge functionality $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$ allows proving of statements in an NP relation $\mathcal{R}$.

State variables and initialisation values:

Variable	Description
$W \coloneqq \varnothing$	Mapping of statement/proof pairs to witnesses
$\Pi \coloneqq \varnothing$	Set of statement/proof pairs
$\overline{\Pi} \coloneqq \varnothing$	Set of known invalid statement/proof pairs

When receiving a message $(\textsc{prove}, x, w)$ *from a party* ψ*:*

if $(x, w) \notin \mathcal{R}$ **then**
  return $\bot$
query $\mathcal{A}$ **with** $(\textsc{prove}, x)$ and **receive the reply** π,
  satisfying $\pi \neq \bot \wedge (x, \pi) \notin \overline{\Pi} \wedge (\cdot, \pi) \notin \Pi$, else **sampling from** $\{0,1\}^\kappa$
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}; W(x, \pi) \leftarrow w$
return π

When receiving a message $(\textsc{verify}, x, \pi)$ *from a party* ψ*:*

if $(x, \pi) \notin \Pi \cup \overline{\Pi} \wedge \pi \neq \bot$ **then**
  query $\mathcal{A}$ **with** $(\textsc{verify}, x, \pi)$ and **receive the reply** R
  if $(x, \pi) \notin \Pi \cup \overline{\Pi}$ **then**
    if $\exists w. R = (\textsc{witness}, w) \wedge (x, w) \in \mathcal{R}$ **then**
      let $\Pi \leftarrow \Pi \cup (x, \pi); W(x, \pi) \leftarrow w$
    else
      let $\overline{\Pi} \leftarrow \overline{\Pi} \cup (x, \pi)$
return $(x, \pi) \in \Pi$

When receiving a message $(\textsc{maul}, x, \pi)$ *from* $\mathcal{A}$*:*

if $\exists \pi' : (x, \pi') \in \Pi \wedge (x, \pi) \notin \overline{\Pi}$ **then**
  let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$

2.5.3 SNARKs

Traditionally, both interactive and non-interactive zero-knowledge protocols are quite costly on both the prover and verifier side, when compared with the cost of a membership test in the relation $\mathcal{R}$. This is especially true as most problems of interest first need to be massaged into an appropriate, typically NP-complete, relation. For distributed ledgers, verification cost dictates the efficiency of the entire chain, making it a far more limiting factor than in other applications.

Furthermore, non-interactive proofs need to be *stored* in distributed pro-

tocols, making their brevity paramount – a 100 MiB proof may be acceptable in many cases, but attached to each transaction on a distributed ledger, will cause the system to fail. Within the last decade, a new wave of research into *succinct* zero-knowledge [Gro10, Lip12, GGPR13, PHGR13, Gro16, GM17, GKM+18, MBKM19, CHM+19, GWC19, CHM+20] brought the technology into the realm of feasibility for distributed ledgers. zk-SNARKs, standing for zero-knowledge Succinct Non-interactive ARguments of Knowledge, typically have proof sizes of at most a few kilobytes and verify in a few milliseconds (for typical circuits). This is very close to the performance characteristics of digital signatures, making them an effective drop-in replacement.

The structure of a SNARK. SNARKs often share many of the same components, although each part has seen optimisation in subsequent works. This section gives a loose intuition for the Sonic [MBKM19] zk-SNARK, although the same ideas apply more broadly. Firstly, the zero-knowledge relation is one of *arithmetic circuit* satisfiability. A fixed number of variables are constrained through addition and multiplication gates with each other. Some of these variables are declared as public inputs. The arithmetic circuit (and notably, the values assigned to it) is then transformed into a corresponding constraint between polynomials. One of these polynomials is publicly computable, and represents the public inputs and (potentially) the structure of the constraints itself.

Many SNARKs begin as interactive protocols, to which the Fiat-Shamir-like transform is applied. First, the prover commits to each of the relevant polynomials, using a specialised cryptographic primitive. After the verifier provides a random challenge, the polynomials are opened at this point. The verifier can test that the public (also called target) polynomial relates to the openings in an expected way – for instance, demonstrating that a polynomial is divisible by it, if it can be multiplied with a third polynomial's opening to reach the same value. This relies on the Schwartz-Zippel lemma, which states evaluating any two polynomials at random points over a domain much larger than their degree d will have a very small probability of resulting in equal outputs.

Circuit satisfiability and polynomials. Various zk-SNARKs differ slightly in their circuit representation, depending on the details of the relation enforced

between polynomials. As a point of commonality, they tend to be easily reducible to the problem of arithmetic circuit satisfiability. An arithmetic circuit over the prime field $\mathbb{F}_p$ consists of a number of variables $v_1, \ldots, v_n$, and arithmetic constraints modulo p between these, taking the form of $v_a + v_b = v_c$ or $v_a \cdot v_b = v_c$. SNARKs differ in regard to how efficiently some of these constraints can be represented – Pinocchio [PHGR13] can make use of linear combinations in multiplication constraints, effectively making additions "free", while Plonk [GWC19] limits the number of constraints each variable can be used in.

An assignment *satisfies* the circuit if each of the addition and multiplication constraints holds for the assignment. A subset of variables are usually marked as public inputs; part of the statement in the relation. The circuit is then transformed into an equivalent statement about relationships between polynomials. For instance, to express a vector of constraints $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$ (i.e. $a_i \cdot b_i = c_i$ for $i \in 1, \ldots, m$), each of $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ is expressed as a polynomial, with different powers representing separate variables, such as:

$$\mathbf{A}(x) \coloneqq \sum_{i=1}^{m} a_i x^i$$

Then the constraint becomes $\mathbf{A} \circ \mathbf{B} = \mathbf{C}$, which by the Schwartz-Zippel lemma can be efficiently checked by testing $x \xleftarrow{*} \mathbb{F}_p$; $\mathbf{A}(x) \cdot \mathbf{B}(x) = \mathbf{C}(x)$. In order to hide the structure of the polynomial, an additional random masking polynomial is added, ensuring that all values appearing in the proof are uniformly distributed.

Polynomial commitments and reference strings. A key part of applying the Schartz-Zippel lemma is the sequence of committing to the polynomial and then having to reveal a random point of it. If the point is known beforehand, the polynomial can be selected to give whichever value one wishes. This order of messages is not always necessary: In Pinocchio [PHGR13], it is possible to send the challenge *first*, provided it is not sent in cleartext. Instead of sending a challenge $s \in \mathbb{F}_p$, evaluations depending on s are sent, such as g^s, g^{s^2} etc. These evaluations are sufficient to construct an evaluation of a polynomial at s, however insufficient to *determine* s. Pinocchio evaluates a polynomial "in the exponent" of the group generator at s (i.e., evaluates $g^{p(s)}$). It then relies on a knowledge assumption known as the Knowledge of Exponent Assumption (KEA) [Dam92, HT98] to guarantee that this exponentially evaluated polynomial actually implies knowledge – it could not have been created without knowing the polynomial itself.

Sonic [MBKM19] has an adjusted form of this, based on [KZG10], which allows it to evaluate polynomials not only on the challenge point s, but on any point, which it later uses to evaluate at a Fiat-Shamir-style chosen point. The general idea remains the same however – an evaluation of the polynomial at a pre-determined (and secret!) point acts as a commitment to it. This crucially means that the reference string, which provides different bases of s must be securely generated, or the commitments do not possess their crucial property of being binding.

2.5.4 Universality and Updateability

A recent family of zk-SNARKs, notably Sonic [MBKM19], and the more efficient Marlin [CHM+20] and Plonk [GWC19] derived from it, achieve two very desirable properties which this thesis makes direct use of: The universality and updatability of their reference string. Although very different properties, they are related, both being properties of the *reference string*. Universality roughly states that the same SNARK can be efficiently adapted for any NP relation $\mathcal{R}$ (within some size bounds). This notably means that the reference string must be sharable between different relations, a feature not present in many zk-SNARKs.

Updatability is more straightforwardly attributable to the reference string, although it also requires the proof system to be secure *despite* it. The idea is simply that any user can transform a reference string, producing a new one which is "more secure": The updated reference string must be secure if either the prior one was, or the user updating it did so honestly.

Universal relations. Consider the Sonic relation as an example of a universal SNARK. The trick of it lies in the relation itself encoding information about constraints, with the constrain information being embedded in the statement x. While Sonic removes some of the overhead due to the increased statement size through a "helped" mode, this is not of particular interest in this thesis. Ignoring this aspect, the proof effectively permits constraining which variables are subject to addition constraints and which are not.

Sonic has variables $a_i, b_i, c_i \in \mathbb{F}_p$ for $i \in \{1, \dots, n\}$, represented with the vectors $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{F}_p^n$. The circuit is constrained through additional vectors $\mathbf{u}_q, \mathbf{v}_q, \mathbf{w}_q \in \mathbb{F}_p^n$, and the scalars $k_q \in \mathbb{F}_p$ for $q \in \{1, \dots, Q\}$. Q represents the number of addition

constraints, which are typically sparse and compressed into a single polynomial.

The NP relationship Sonic produces proofs over is

$$\mathbf{a} \circ \mathbf{b} = \mathbf{c}$$
$$\mathbf{a} \cdot \mathbf{u}_q + \mathbf{b} \cdot \mathbf{v}_q + \mathbf{c} \cdot \mathbf{w}_q = k_q,$$

where $\circ$ denotes element-wise multiplication and $\cdot$ the dot-product. By using values in $\{-1, 0, 1\}$ for $u_{q,i}$, $v_{q,i}$, and $w_{q,i}$ values it is possible to control whether they are constrained in addition or not. It also permits constraining individual variables to be equal to others and equal to specific constants, allowing the statement to be determined entirely through the choice of $\mathbf{u}_q$, $\mathbf{v}_q$, $\mathbf{w}_q$, and k_q.

Updateable reference stings. A polynomial commitment scheme, used in Sonic to commit to monomials, is the primary usage of the reference string. While this thesis does not discuss polynomial commitments in depth, it relies primarily on evaluating $g^{p(x)}$ and $g^{\alpha p(x)}$ for a polynomial of degree d (potentially including negative exponents) where α and x are (secret) trapdoor values of the reference string. This is possible to construct from the coefficients given values g^{x^i} and $g^{\alpha x^i}$ for all necessary values of d. Specifically, given the trapdoor $(\alpha, x) \xleftarrow{*} \mathbb{F}_p^2$, the structure function

$$S((\alpha, x)) = \left(\left\{ g^{x^i}, h^{x^i}, h^{\alpha x^i} \right\}_{i=-d}^{d}, \left\{ g^{\alpha x^i} \right\}_{i=-d, i\neq 0}^{d} \right)$$

is sufficient to construct the "in exponent" evaluations necessary for Sonic.

This reference string is universal, in the sense that it is possible to "multiply" the trapdoor by a randomly sampled field element, by exponentiating each component of the reference string:

$$S((\alpha\beta, xy)) = \left(\left\{ \left(g^{x^i}\right)^{y^i}, \left(h^{x^i}\right)^{y^i}, \left(h^{\alpha x^i}\right)^{\beta y^i} \right\}_{i=-d}^{d}, \left\{ \left(g^{\alpha x^i}\right)^{\beta y^i} \right\}_{i=-d, i\neq 0}^{d} \right)$$

This does not permit an adversary to remove entropy from the reference string, as the adversary cannot find (α, x), but permits honest users to inject more entropy. This is the focus of Chapter 4, where it is described more formally and applied in practice.

2.6 Smart Contracts

The basic premises of a distributed ledger is merely to agree on a sequence of transactions, not to assign any meaning to these transactions. This is a useful theoretical view, however in practice transactions are taxing on the network: they must be replicated, distributed and stored (potentially forever!). This suggests an immediate and very real danger of denial-of-service attacks. What is to prevent a malicious user from flooding the network with useless transactions, preventing it from processing anything useful?

To combat this, the practice of blockchains is different from the idealised transaction-orderer, requiring not only transactions to adhere to specific formats, but imposing a toll on network usage: Each transaction must pay a fee for inclusion. Denial-of-service is much less feasible as a result, however it relies on being able to accurately asses how much strain each transaction takes on the network. As the same currency which the transactions define operations over pays for fees, testing if they are satisfied requires taking a step further and assigning meaning to each transaction, eventually boiling down to the following, ostensibly simple, question: "*Does the user making this transaction have the funds to pay for its fees?*"

Ensuring that transactions which cannot pay for their fees are rapidly dismissed is paramount to resisting denial of service attacks. Bitcoin [Nak08] relies on short, quickly verifiable scripts as a result, written in a language so basic as to lack loops or jump statements. As a result, the time it takes to run these scripts is predictable ahead of time – at most as long as their size.

As different people began attempting to use Bitcoin for purposes which did not fit into the limited expressiveness, new custom-built protocols, such as NameCoin [KCE+15], a distributed domain name registration protocol, and Bitmessage [War12], a ledger-based communications protocol, arose. These used their own separate blockchain with modified semantics for transactions, better suited for their purpose. An obvious problem with this approach is that, even though the Bitcoin source code can be copied arbitrarily often, the Bitcoin community of software developers and miners cannot, and hence such systems are typically not sustainable. Smart contracts, originally posited as a form of reactive computation [Sza97], were popularised by Ethereum [Woo14], solving these problems by providing a uniform and standardised approach for

deploying decentralised computation over the same back-end infrastructure.

2.6.1 Ethereum

Ethereum [Woo14] envisioned arbitrarily programmable semantics for transactions. This brings with it a few immediate questions: Who gets to set the rules for which interactions? How is it possible to prevent denial-of-service when each individual transaction may invoke a large amount of computation (and due to the halting problem, there is no way to determine if it is finite, let alone how expensive the computation is). These problems were solved quite elegantly, a testament to which is the continuing popularity and usage of Ethereum (this is not to say Ethereum does not have flaws – considering off-chain computation as "out-of-scope" being potentially the most serious of these, which will feature in depth in Chapter 6).

The basic semantics of Ethereum. Ethereum has a basic currency transfer system, not unlike Bitcoin as its basis. It differs slightly in that unlike Bitcoin, Ethereum encourages the reuse of public keys, facilitating this and protecting against replay attacks, by associating each public key with a nonce. Transactions must include the current nonce, ensuring a new transaction needs to be signed every time. Although it is often cited as a major difference between Ethereum and Bitcoin, the difference is rather subtle and the designs are in fact isomorphic [Zah18]. More notable is Ethereum permitting two additional types of transactions: Smart contract creations and smart contract invocations. In a smart contract creation, a user submits a program, written in Ethereum's native bytecode language, which gets assigned a unique address. The submitting user pays storage fees for the program and can submit any program they wish, without restriction[18]. This contract can hold its own funds and in addition to its code, can store additional data (the storage of which also requires payment).

The contract's code only governs interactions users directly make with the contract – when a user *invokes* it (with a transaction specifically for this purpose, as mentioned above), they provide an input to the contract's program, as well as optionally providing it with funds. The contract's program executes on this input, as well as auxiliary information including information about the current

[18] In practice, contracts are restricted in size, although this can also be bypassed if necessary by splitting a contract into multiple parts.

block, the address (public key) of the caller, and the funds provided to it. The contract may also invoke other contracts and initiate funds transfers on its own behalf (but not on behalf of its caller!). In some ways the contract is an autonomous, trusted third party, although it is limited in its agency.

Preventing denial-of-service through "gas". The question remains how to limit the impact of denial-of-service attacks. The first part of the solution lies in Ethereum's support for a basic transfer protocol – it is possible to, just as in Bitcoin, quickly test that a user has the funds to pay for a transaction in this basic system, even though managing funds of contracts can require more complex computations. This leaves the question of how to accurately judge how much a transaction *should* cost. As the halting problem hints at the impossibility of this problem, Ethereum instead bypasses it: Instead of attempting to calculate what a transaction should cost, it asks the creator of each transaction to give their own estimate. If a user cannot estimate it (for instance, due to it running too long), the transaction would almost certainly be too expensive for the network to execute as well. Further, there is no need to trust a user on their estimate: If they claim a computation takes five steps to compute and it is not finished after these, Ethereum simply marks the transaction as failed. Crucially however, it is still *valid*, in the sense that it is included in the blockchain – and its fees are taken.

The Ethereum Virtual Machine (EVM). Achieving consensus about the execution of contracts is a subtle matter – and slight difference in the execution semantics between different users can lead to disagreement and the divergence of their state, breaking consensus. Seemingly minor differences, such as bugs in the implementation of floating-point arithmetic, or undefined behaviour of operations, can lead to a collapse of distributed consensus. It is primarily for this reason that Ethereum adopted a virtual machine, designed specifically to be well-defined in all corner cases (although it also benefited from providing application-specific primitives, such as signature verification). Subsequent smart contract systems have largely followed this approach, designing their own atomic "virtual machine" languages with well-defined execution.

A high-level language, Solidity, was also designed to allow developers to write smart contracts in a simpler manner (there are a few other languages targeting the EVM, however Solidity is by far the most popular). This is reminiscent

of object-oriented programming, although it is more akin to a message-passing language, as "objects" represent actual semi-autonomous entities, rather than being a tool for abstraction.

Off-chain execution and Web3. Ethereum's vision involves a distributed "evolution" of the world-wide web. As a means toward this, its primary mode of interacting with smart contracts is via JavaScript, which is assumed to be able to interact with a low-level Inter-Process Communication (IPC) socket to talk with a local (or remote) Ethereum client. This JavaScript interface is called "Web3" as a result of its ambitions – either optimistically, or arrogantly – depending on your point of view.

While ideally this API would be limited to tying together HTML front-end user interfaces with the underlying smart contract, in practice the JavaScript portion of the program also performs pre-computations, and acts more as a part of the overall smart contract, rather than a front-end *to* it. A more holistic approach to on- and off-chain computation is part of some competing smart contract designs, such as Plutus [CKM+19]. It mars the modular, interactive nature of on-chain smart contracts, and as will be discussed in Chapter 6, does not lend itself for privacy-focused smart contract design as a result.

2.6.2 UTxO-Based

Ethereum's approach to smart contracts takes one which is consistent with their interpretation in this thesis, modelling smart contracts as state machines, their state being replicated and reproduced via a distributed ledger. For completeness, it is worth sketching the main alternative approach to modelling smart contracts, which embraces the transactional nature of the distributed ledger.

Rather than smart contracts being their own entities, as in Ethereum, the approach of UTxO-based smart contract system says that *assets* are their own entities, with code dictating who can use them. This is the design underlying Bitcoin – Bitcoin Script controls who can spend a particular transaction's output, and although it is typically simply a public key test, more complex logic can be applied. Another example of this is Plutus [CKM+19], which also extends the model of unspent transaction outputs to allow them to carry data, enabling more complex behaviours including state evolution. The privacy-preserving

Zexe [BCG+20] also falls into a similar category, with the caveat of the UTxO structure itself being hidden.

Ethereum's approach is largely isomorphic to the extended UTxO model of Plutus, although without state-passing it is clearly more powerful. Even state-passing gives a slight advantage to Ethereum's modelling, in that it supports concurrent interactions better, a distinction described in detail in Chapter 6. The increased expressiveness comes at a cost to clarity however – numerous bugs in Ethereum have lead to great financial losses, making the case for a simpler model.

2.6.3 Privacy Focused Variations

A body of research has looked into addressing the privacy shortcomings of smart contracts. These take a variety of different approaches, the most notable of which are briefly discussed here. This work is especially relevant to Chapter 6, which presents as alternative approach to modelling privacy in smart contracts.

Zexe. Zerocash [BCG+14] is a well-known privacy-preserving payment system, allowing direct private payments on a public ledger. Zexe [BCG+20] extends its expressiveness by allowing arbitrary scripts, reminiscent of Bitcoin-scripts, to be evaluated in zero-knowledge in order to spend coin outputs. It is a major improvement in expressiveness over Zerocash, which only permits a few types of transactions. Combined with the extended UTxO approach mentioned in Subsection 2.6.2 of carrying state over from transaction output to transaction output, Zexe would form a very expressive smart contract systems, although its limitations remain uncertain.

zkay. zkay [SBG+19] extends Ethereum smart-contracts with types for private data. It allows users to share encrypted data on-chain, and prove that data is correctly encrypted and correctly used in subsequent interactions. These proofs are managed through the ZoKrates [ET18] framework, which compiles Ethereum contracts into NIZK-friendly circuits. Its usage is limited to fixed size pieces of private data.

Hawk. One of the earliest works on privacy in smart contracts, Hawk [KMS+16] is also one of the most general. It describes how to compile private variants of

smart contracts, given that all participants of the contract trust the same party with its privacy. This party, the "manager", can break the contract's privacy guarantees if they are corrupt, however they cannot break the correctness of the contract's rules. The construction used in Hawk for the manager party relies on zero-knowledge proofs of correct contract execution.

Zether. A lot of work on privacy in smart contracts has focused on retro-fitting privacy into existing systems. Zether [BAZB19], for instance, constructs a privacy-preserving currency within Ethereum, which can be utilised for a number of more private applications, such as hidden auctions. As with most retro-fitted systems, Zether is constrained by the system it is built for and does not generalise to many applications.

Enigma. There are two forms of Enigma: A paper discussing running secure multi-party computation for smart contracts [ZNP15], and a system of the same name designed to use Intel's SGX enclave to guarantee privacy [EPT19]. The former has a lot of potential advantages, but is severely limited by the efficiency of general-purpose MPC protocols. The latter is a practical construction and can claim much better performance than any cryptography-based protocol. The most obvious drawbacks are the reliance on an external trust assumption and the poor track record of secure enclaves against side-channel attacks [BMW+18].

Arbitrum. Using a committee-based approach, Arbitrum [KGC+18] describes how to perform and agree on off-chain executions of smart contracts. A committee of managers is charged with execution, and, in the optimistic case, simply posts commitments to state updates on-chain. In the case of a dispute, an on-chain protocol can resolve the dispute with a complexity logarithmic in the number of computation steps taken. Arbitrum provides correctness guarantees even in the case of $n - 1$ out of n corrupt committee members, however relies on a fully honest committee for privacy.

State channels. State channels, such as those discussed in [DFH18], occupy a similar space to Arbitrum, due to their reliance on off-chain computation and on-chain dispute resolution. The dispute resolution process is different, more aggressively terminating the channel, and typically it considers only participants on the channel that interact with each other. The privacy given

is almost co-incidental, due to the interaction being local and off-chain in the optimistic case.

Piperine. Piperine [LNS20] uses a similar model and approach as presented in Chapter 6, relying on zero-knowledge proofs of correct state transitions, and modelling smart contracts as replicated state machines. Piperine focuses on efficiency gains from this approach, rather than privacy gains, which it does not capture, while our work does not account for the benefit of transaction batching. The notion of state oracles presented in Chapter 6 can be seen as a generalisation of the state interactions presented in [LNS20].

3 Composition with Knowledge Assumptions

This chapter is based on "Composition with Knowledge Assumptions" [KKK21a], first published at the Crypto 2021 conference, primarily authored by Thomas Kerber, and co-authored by Aggelos Kiayias and Markulf Kohlweiss.

Knowledge assumptions, discussed briefly in Subsection 2.2.3, are especially useful in cases where both succinctness and extractability are required. This is notably the case for zk-SNARKs, as discussed in Subsection 2.5.3, which typically rely on either a knowledge-of-exponent assumption [Dam92], the Algebraic Group Model (AGM) [FKL18], or the even stronger Generic Group Model (GGM) [Sho97]. Their importance for zk-SNARKs makes them particularly relevant for this thesis, as Chapter 5 and Chapter 6 rely on a composable non-interactive zero-knowledge functionality, which would ideally be implemented using zk-SNARKs. Nevertheless, the composition of knowledge assumptions has applications outside of zk-SNARKs, for instance in extractable functions [CD08, CD09, BCCT12], and the modelling in this chapter even lends itself to a novel interpretation of random oracles.

Proving the security of SNARKs under composition would typically involve using a compositional framework (see Section 2.3), such as Universal Composability [Can01] or Constructive Cryptography [Mau11], specifying an ideal behaviour for the primitive and constructing a simulator which coerces the ideal behaviour to mimic that of the actual protocol. This simulator will naturally need to make use of the extraction properties, often to infer the exact ideal intent behind adversarial actions. It is in this that the conflict between extraction and compositional frameworks arises: As the extraction is white-box, the simulator requires the input of its counter-party – the environment, or distinguisher, of the simulation experiment. This cannot be allowed however, as it would give the simulator access to *all* information in the system[1], not just that of the adversary.

[1] Recall that the simulator *is* the ideal-world adversary and should by definition not have

This conflict has been observed before, for instance in [KZM+15]. Often, the remedy is to extend the original protocol with additional components to enable the simulator to extract "black-box", i.e. without the original inputs. For example, the Fischlin transform [Fis05] uses multiple queries to a random oracle to bypass the inability to extract from the commitment phase of an underlying Sigma protocol, which would allow using the simpler Fiat-Shamir transform [FS87] instead. C∅C∅ [KZM+15] extends zk-SNARKs with an encryption of the witness and a proof of correctness of this encryption to a public key the simulator can control.

A theme of these approaches is that succinctness is usually lost – size being limited by the information-theoretic reality of black-box extraction. Thus C∅C∅ proofs are longer than their witnesses and UC-secure commitments [CF01] are longer than the message domain.

This limitation can often be bypassed by using a local random oracle, as this *does* permit extraction. Restricting the model to allow the adversary to perform only specific computations on knowledge-implying objects, could be one way to generalise this approach. Just as a random oracle functionality would abstract over extractable hash functions, a generic group functionality would abstract over knowledge of exponent type assumptions. This would constitute a far stronger assumption however, running counter to recent developments to relax assumptions, such as the Algebraic Group Model [FKL18], which aim for a more faithful representation of knowledge assumptions.

In this chapter, a different approach is taken by defining the concept of knowledge-respecting distinguishing environments, or distinguishers (to be consistent with the terminology of Constructive Cryptography). The Constructive Cryptography framework [Mau11] serves as an orientation point for this work, due to its relative simplicity compared to the many moving parts of UC [Can01], making it easier to re-establish composition after making sweeping changes to the model.

Similar to an algebraic algorithm, distinguishers in our model need to explain how they computed each knowledge-implying object they output. The compositional framework is extended by giving the simulator access to these explanations.

Furthermore, this chapter discusses the conditions under which it is rea-

access to secrets the distinguisher holds.

sonable to assume knowledge-respecting distinguishers. To this end, stronger versions of knowledge assumptions are defined that depend on auxiliary and knowledge-implying inputs. These assumptions suffice to extend a distinguisher with an extractor providing said explanations.

Within this setting we are able to establish not only an impossibility result on full general composition, but more interestingly a positive result on the composition of systems relying on different knowledge assumptions. Intuitively: You can use a knowledge assumption only once, or you need to ensure the various uses do not interfere with each other (specifically, the simulators of both invocations cannot provide any advantage due to extraction, as shown in the example in Section 3.4). This result has the immediate effect of enabling the usage of primitives relying on knowledge assumptions in larger protocols – provided the underlying assumption is not used in multiple composing proofs.

3.1 Modelling Knowledge Assumptions

We formally define knowledge assumptions over a type of *knowledge-implying objects* X. When an object of the type X is produced, the assumption states that whoever produced it must know a corresponding witness of the type W. The *knowledge of exponent* assumption is an example of this, where X corresponds to pairs of group elements and W is an exponent. A relation $\mathcal{R} \subseteq X \times W$ defines which witnesses are valid for which knowledge-implying objects.

In the case of the knowledge of exponent assumption, it roughly states that given a generator and a random power s of the generator, the only way to produce a pair of group elements, where one is the sth power of the other, is to exponentiate the original pair and in so doing implying knowledge of this exponent. There is one extra item needed: The initial exponent s needs to be sampled randomly. Indeed, this is true for *any* knowledge assumption: The all-quantification over potential distinguishers implies the existence of distinguishers which "know" objects in X without knowing their corresponding witness. To avoid this pre-knowledge, we assume X itself is randomly selected at the start of the protocol. For this purpose, we will assume a distribution init, which given a source of public randomness (such as a global common random string), produces *public parameters* pp, which parameterise the knowledge assumption. In the case of knowledge of exponent, this needs to sample

an exponent s and output the pair (g, g^s). For this particular setup, public randomness is insufficient.

Beyond this, users do not operate in isolation: If Alice produces the pair (g^x, g^{xs}), knowing x and transmits this to Bob, he can produce (g^{xy}, g^{xys}) *without* knowing xy. This does not mean that the knowledge assumption does not hold, however it is more complex than one might originally imagine: One party can use knowledge-implying objects from another user as (part of) their own witnesses. Crucially this needs to be limited to objects the user actually received: Bob *cannot* produce (g^{sy}, g^{s^2y}) for instance, as he never received (g^s, g^{s^2}) and does not know s. This setting also lends itself more to some interpretations of knowledge assumptions than others. For instance, the classical knowledge-of-exponent assumption [Dam92] does not allow linear combinations of inputs, while the t-knowledge-of-exponent assumption [HT98] does. When used composably, the latter is more "natural", in much the same way that IND-CCA definitions of encryption fit better into compositional frameworks than IND-CPA ones, due to them already accounting for part of the composable interaction.

Definition 3.1 (Knowledge Assumption). A knowledge assumption $\mathfrak{K}$ is defined by a tuple $(\mathsf{init}, X, W, \mathcal{R})$ consisting of:

1. init, a private-coin distribution to sample public parameters pp from, which the others are parameterised by.
2. X_{pp}, the set of all objects which imply knowledge.
3. W_{pp}, the set of witnesses, where $\forall x \in X_{\mathsf{pp}} \colon (\textsc{input}, x) \in W_{\mathsf{pp}}$.
4. $\mathcal{R}_{\mathsf{pp}} \colon (I \subseteq X_{\mathsf{pp}}) \to (Y \subseteq (X_{\mathsf{pp}} \times W_{\mathsf{pp}}))$, the relation new knowledge must satisfy, parameterised by input objects, where

$$\forall x, y \in X_{\mathsf{pp}}, I \subseteq X_{\mathsf{pp}} \colon (x, (\textsc{input}, y)) \in \mathcal{R}_{\mathsf{pp}}(I) \iff x = y \wedge x \in I.$$

Furthermore, $\mathcal{R}_{\mathsf{pp}}$ must be monotonically increasing:

$$\forall I \subseteq J \subseteq X_{\mathsf{pp}} \colon \mathcal{R}_{\mathsf{pp}}(I) \subseteq \mathcal{R}_{\mathsf{pp}}(J).$$

The inclusion of $(\textsc{input}, x)$ in W_{pp} and $\mathcal{R}_{\mathsf{pp}}$ for all $x \in X_{\mathsf{pp}}$ ensures that parties are permitted to know objects they have received as inputs, without needing to know corresponding witnesses. Importantly, this is possible *only* for inputs and not for

other objects. For each knowledge assumption $\mathfrak{K}$, the assumption it describes is in a setting of computational security, with a We state the assumption itself in a setting of computational security, with a security parameter κ. Broadly, the assumption states that, for a restricted class of "$\mathfrak{K}$-respecting" adversaries, it is possible to compute witnesses for each adversarial output, given the same inputs.

Assumption 3.1 ($\mathfrak{K}$-Knowledge). *The assumption corresponding to the tuple $\mathfrak{K} = (\mathsf{init}, X, W, \mathcal{R})$ is associated with a set of probabilistic polynomial time (PPT) algorithms, $\mathsf{Resp}_{\mathfrak{K}}$. We will say an algorithm is $\mathfrak{K}$-respecting if it is in $\mathsf{Resp}_{\mathfrak{K}}$. This set should contain all adversaries and protocols of interest. The $\mathfrak{K}$-knowledge assumption itself is then that, for all $\mathcal{A} \in \mathsf{Resp}_{\mathfrak{K}}$, there exists a PPT extractor $\mathcal{X}$, such that:*

$$\Pr\left[\begin{array}{l}\mathsf{pp} \xleftarrow{*} \mathsf{init};\\ \exists I \subseteq X_{\mathsf{pp}}, \mathsf{aux} \in \{0,1\}^*:\\ \mathsf{Game}\ 3.1(\mathcal{A}_r, \mathcal{X}_r, \mathsf{pp}, I, \mathsf{aux})\end{array}\right] \leq \mathsf{negl}(\kappa),$$

where $\mathcal{A}_r$ and $\mathcal{X}_r$ are $\mathcal{A}$ and $\mathcal{X}$ supplied with the same random coins r (as such, they behave deterministically within Game 3.1).

While it is trivial to construct adversaries which are not $\mathfrak{K}$-respecting by encoding knowledge-implying objects within the auxiliary input, these trivial cases are isomorphic to an adversary which *is* $\mathfrak{K}$-respecting and which receives such encoded objects directly. We therefore limit ourselves to considering adversaries which communicate through the "proper" channel, rather than covertly. In this way, we also bypass existing impossibility results employing obfuscation [BP15]: We exclude by assumption adversaries which would use obfuscation.

Game 3.1 (Knowledge Extraction). *The adversary $\mathcal{A}_r$ wins the knowledge extraction game if and only if it outputs a series of objects in X_{pp}, for which the extractor $\mathcal{X}_r$ fails to output the corresponding witness:*

$$\mathsf{let}\ \vec{x} \leftarrow \mathcal{A}_r(I, \mathsf{aux}), \vec{w} \leftarrow \mathcal{X}_r(I, \mathsf{aux})\ \mathsf{in}\ \vec{x} \in X_{\mathsf{pp}}^* \wedge \bigvee_{i=1}^{|\vec{x}|}(x_i, w_i) \notin \mathcal{R}_{\mathsf{pp}}(I).$$

Crucial for composition are the existential quantifications, which combined state that we assume extraction *for all* of the following:

- Algorithms in $\mathsf{Resp}_{\mathfrak{K}}$
- Input objects I
- Auxiliary inputs aux

This makes knowledge assumptions following Assumption 3.1 stronger than their typical property-based definitions. It is also non-standard as a result, as it relies on quantifiers *within* a probability experiment. While the adversarial win condition is well-defined, it is not necessarily computable. Nevertheless, quantifications are required for their usage in composable proofs.

3.1.1 Examples of Knowledge Assumptions

To motivate this definition, we demonstrate that it can be applied to various commonly used knowledge assumptions, including the knowledge of exponent assumption, the Algebraic Group Model and variants, and even to random oracles. We detail our flavour of these here. Witnesses naturally seem to form a restricted expression language describing how to construct a knowledge-implying object. A more natural way to express the relation $\mathcal{R}$ is often an evaluation function over witnesses, returning a knowledge-implying object.

Knowledge of Exponent Assumption. The t-knowledge-of-exponent assumption [Dam92, HT98] depends on a (possibility pre-selected) group $\mathbb{G}$ of order p and generator $g \in \mathbb{G}$. It selects a random exponent s and provides the pair (g, g^s) as public parameters. Any pair of group elements where the second is the sth power of the first must provide an exponent to match.

A curiosity of this knowledge assumption is that there is no simple membership test to apply. As a result, we permit pairs *not* related in this way to be members of X_{pp}, which do not require witnessing, capturing the possibility of transmitting unrelated group elements.

$$
\begin{aligned}
\mathfrak{K}_{\mathsf{KEA}} &\coloneqq (\mathsf{init}, X, W, \mathcal{R}) \\
\mathsf{init} &\coloneqq s \xleftarrow{*} \mathbb{F}_p^*; (g, g^s) \\
X &\coloneqq \mathbb{G}^2
\end{aligned}
\qquad
\begin{aligned}
W \coloneqq\ & \{(\textsc{base})\} \cup \\
& \{\, (\textsc{input}, i) \mid i \in X \,\} \cup \\
& \{\, (\textsc{exp}, b, e) \mid b \in W, e \in \mathbb{F}_p^* \,\} \cup \\
& \{\, (\textsc{mul}, b, c) \mid b, c \in W \,\} \cup \\
& \{\, (\textsc{free}, g, h) \mid g, h \in \mathbb{G} \,\}
\end{aligned}
$$

$$\mathsf{eval}(I, w) \coloneqq \begin{cases} (g, g^s) & \text{if } w = \textsc{base} \\ i & \text{if } w = (\textsc{input}, i) \wedge i \in I \\ (a^e, b^e) & \text{if } w = (\textsc{exp}, c, e) \wedge (a, b) = \mathsf{eval}(I, c) \\ (a \circ c, b \circ d) & \text{if } w = (\textsc{mul}, e, f) \wedge (a, b) = \mathsf{eval}(I, e) \\ & \qquad\qquad \wedge (c, d) = \mathsf{eval}(I, f) \\ (g, h) & \text{if } w = (\textsc{free}, g, h) \wedge g^s \neq h \end{cases}$$

$$(x, w) \in \mathcal{R}(I) \iff x = \mathsf{eval}(I, w)$$

The Algebraic Group Model. Assuming a distribution groupSetup providing a group $\mathbb{G}$ and a generator g, we can recreate the Algebraic Group Model [FKL18] as a knowledge assumption fitting Definition 3.1:

$$\mathfrak{K}_{\mathsf{AGM}} \coloneqq (\mathsf{init}, X, W, \mathcal{R}) \qquad W \coloneqq \{\, (\textsc{op}, a, b) \mid a, b \in W \,\} \cup$$
$$\mathsf{init} \coloneqq \mathsf{groupSetup} \qquad \{\, (\textsc{input}, i) \mid i \in X \,\} \cup$$
$$X \coloneqq \mathbb{G} \qquad \{\, \textsc{generator} \,\}$$

$$\mathsf{eval}(I, w) \coloneqq \begin{cases} \mathsf{eval}(g) \circ \mathsf{eval}(h) & \text{if } w = (\textsc{op}, g, h) \\ i & \text{if } w = (\textsc{input}, i) \wedge i \in I \\ g & \text{if } w = \textsc{generator} \end{cases}$$

$$(x, w) \in \mathcal{R}(I) \iff x = \mathsf{eval}(I, w)$$

The Bilinear Algebraic Group Model with Random Sampling. Assuming a distribution groupSetup providing groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, a bilinear pairing operation $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, and generators $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$, we define the corresponding knowledge assumptions of a bilinear Algebraic Group Model below. Random sampling of group elements in $\mathbb{G}_1$ and $\mathbb{G}_2$ is permitted by providing random permutations[2] $\vec{\mathbb{G}_1}$ of $\mathbb{G}_1$ and $\vec{\mathbb{G}_2}$ of $\mathbb{G}_2$ as part of the public parameters – we assume machines have random memory access and can therefore easily query random elements in these vectors, but cannot search for specific elements,

[2] We write S_X for the set of permutations of X and the corresponding uniform distribution.

due to the exponential size of the group. Note that this assumes public parameter selection is free from computational feasibility constraints.

$$\mathfrak{K}_{\mathsf{bAGM}} \coloneqq (\mathsf{init}, X, W, \mathcal{R})$$

$$\mathsf{init} \coloneqq \mathsf{let}\ (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, h) \xleftarrow{*} \mathsf{groupSetup}; \vec{\mathbb{G}_1} \xleftarrow{*} S_{\mathbb{G}_1}; \vec{\mathbb{G}_2} \xleftarrow{*} S_{\mathbb{G}_2}$$
$$\mathsf{in}\ (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, h, \vec{\mathbb{G}_1}, \vec{\mathbb{G}_2})$$

$$X \coloneqq \mathbb{G}_1 \uplus \mathbb{G}_2 \uplus \mathbb{G}_T$$

$$\begin{aligned} W \coloneqq\ & \{\ (\textsc{op}, a, b) \mid a, b \in W\ \} \cup \\ & \{\ (\textsc{pairing}, a, b) \mid a, b \in W\ \} \cup \\ & \{\ (\textsc{input}, i) \mid i \in X\ \} \cup \\ & \{\ (\textsc{index}, i, x) \mid x \in \{1, 2\}\}, i \in \mathbb{Z}_{|\mathbb{G}_i|}\ \} \cup \\ & \{\ (\textsc{generator}, x) \mid x \in \{1, 2\}\ \} \end{aligned}$$

$$\mathsf{eval}(I, w) \coloneqq \begin{cases} \mathsf{eval}(a) \circ \mathsf{eval}(b) & \text{if } w = (\textsc{op}, a, b) \\ e(\mathsf{eval}(a), \mathsf{eval}(b)) & \text{if } w = (\textsc{pairing}, a, b) \\ i & \text{if } w = (\textsc{input}, i) \wedge i \in I \\ (\vec{\mathbb{G}_x})_i & \text{if } w = (\textsc{index}, i, x) \\ g & \text{if } w = (\textsc{generator}, 1) \\ h & \text{if } w = (\textsc{generator}, 2) \end{cases}$$

$$(x, w) \in \mathcal{R}(I) \iff x = \mathsf{eval}(I, w)$$

Notably $\circ$ and e may be undefined – $g \circ h$ is not defined, for instance. In this case eval is also not defined: $\mathsf{eval}(I, (\textsc{op}, g, h))$ is undefined.

Random Oracles. Somewhat surprisingly, (global, non-programmable) random oracles can be seen as a somewhat unique knowledge assumption, using a similar technique for randomness as is used to sample random group elements above. Unlike the above, the public parameters need to encode an *infinite* sequence of random values – effectively publicly describing the entire random oracle. Again an assumption of random access to these public parameters implies a limited number of possible "queries" to this random oracle, with each query simply reading the nth random value in the sequence, where n is a numerical encoding of the query.

In practice, this assumption has similarities to that of an extractable hash

function [KLT16] and global random oracles [CJS14, CDG+18]. It fits the former in that for every "hash" produced by an algorithm, an extractor must be able to output its preimage, and the latter if this operation is viewed as a black-box query to a global random oracle functionality.

$$
\begin{aligned}
\mathfrak{R}_{\mathsf{RO}} &\coloneqq (\mathsf{init}, X, W, \mathcal{R}) \\
\mathsf{init} &\coloneqq (\{0,1\}^\kappa)^\infty \\
X &\coloneqq \{0,1\}^\kappa \\
(x, w) \in \mathcal{R}(I) &\iff x = \mathsf{eval}(I, w)
\end{aligned}
\qquad
\begin{aligned}
W &\coloneqq \{\, (\textsc{input}, i) \mid i \in X \,\} \cup \\
&\quad \{\, (\textsc{index}, i) \mid i \in \mathbb{N} \,\} \\
\mathsf{eval}(I, w) &\coloneqq \begin{cases} i & \text{if } w = (\textsc{input}, i) \\ & \quad \wedge\, i \in I \\ \mathsf{pp}_i & \text{if } w = (\textsc{index}, i, x) \end{cases}
\end{aligned}
$$

3.2 Typed Networks of Random Systems

While it is not our goal to pioneer a new composable security framework, existing frameworks do not quite fit the needs of this chapter. Notably, Universal Composability [Can01] has many moving parts, such as session IDs, control functions and different tapes which make the core issues harder to grasp. Constructive Cryptography [Mau11] does not have a well-established notion of globality and fixes the number of interfaces available, which makes the transformations we will later perform more tricky.

Furthermore, the analysis of knowledge assumptions benefits from a clear type system imposed on messages being passed – knowing which parts of messages encode objects of interest to knowledge assumptions (and which do not) makes the analysis more straightforward. Due to both of these reasons, we construct a compositional framework sharing many similarities with Constructive Cryptography (see Subsection 2.3.2), however using graphs (networks) of typed random systems as the basic unit instead of random systems themselves. Crucially, when we establish composition within this framework, we do so with respect to sets of valid distinguishers. This will allow us to permit only distinguishers which respect the knowledge assumption.

Our definitions can embed existing security proofs in Constructive Cryptography and, due to the close relation between composable frameworks, likely also those in other frameworks, such as UC. In particular, our results directly imply

that primitives proven using knowledge assumptions under this framework can be directly used in place of hybrids in systems proven in Constructive Cryptography.

We will not go into depth on modelling computational security, as it is not the primary focus of this chapter, however we will assume the existence of a feasibility notion of this type. We follow the approach of [LM20] and consider random systems as equivalence classes over probabilistic systems. We make a minor tweak to the setting of [Mau11] as well and use random-access machines instead of automata[3], to enable the use of super-polynomial parameters as laid out in Subsection 3.1.1.

3.2.1 Type Definition

We introduce a rudimentary type system for messages passed through the network. It consists of a unit type $\mathbb{1}$, empty type $\mathbb{0}$, sum and product types $\tau_1 + \tau_2 / \tau_1 \times \tau_2$, and the Kleene star τ^*. This type system was chosen to be minimal while still:

1. Allowing existing protocols to be fit within it. As most of cryptography operates on arbitrary length strings, $(\mathbb{1}+\mathbb{1})^*$, or finite mathematical objects, $\mathbb{1} + \ldots + \mathbb{1}$, these can be embedded in the type system.

2. Allowing new types to be embedded in larger message spaces. The inclusion of sum types enables optional inclusion, while product types enables inclusion of multiple instances of a type alongside auxiliary information.

We stress that this type system may be (and will!) extended and that a richer system may make sense in practice. Types follow the grammar:

$$\tau \equiv \mathbb{0} \mid \mathbb{1} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau^*,$$

and the corresponding expression language follows the grammar

$$E \equiv \top \mid \mathsf{inj}_1(E) \mid \mathsf{inj}_2(E) \mid (E_1, E_2) \mid \varepsilon \mid E_1 :: E_2.$$

[3]Specifically, we assume each of the following to be of time complexity $\Theta(1)$: 1. receiving and sending messages of any length, 2. (de)constructing sum and product types, 3. accessing a given index in a bit string for reading or writing, 4. copying objects of any size, which is assumed to be done through copy-on-write references.

We will also use $\mathbb{2}$ to represent $\mathbb{1}+\mathbb{1}$, and 0 and 1 for $\mathsf{inj}_1(\top)$ and $\mathsf{inj}_2(\top)$ respectively. Formally, the typing rules are:

$$\vdash \top : \mathbb{1} \qquad \frac{\vdash x : \tau_1}{\vdash \mathsf{inj}_1(x) : \tau_1 + \tau_2} \qquad \frac{\vdash x : \tau_2}{\vdash \mathsf{inj}_2(x) : \tau_1 + \tau_2}$$

$$\frac{\vdash x : \tau_1 \qquad \vdash y : \tau_2}{\vdash (x, y) : \tau_1 \times \tau_2} \qquad \vdash \varepsilon : \tau^* \qquad \frac{\vdash x : \tau \qquad \vdash \vec{x} : \tau^*}{\vdash x :: \vec{x} : \tau^*}$$

Note that there is no means to construct the empty type $\mathbb{0}$.

Knowledge assumptions. We expand this basic type system by allowing objects to be annotated with a knowledge assumption. Specifically, given a knowledge assumption $\mathfrak{K} = (\mathsf{init}, X, W, \mathcal{R})$, where init returns $\mathsf{pp} : \tau$, and for all pp in the domain of init, both X_{pp} and W_{pp} are valid types, there are two additional types present:

1. The type of knowledge-implying objects in $\mathfrak{K}$: $[\mathfrak{K}_{\mathsf{pp}}]$ (equivalent to X_{pp})
2. The type of witnessed objects in $\mathfrak{K}$ with respect to an input set of knowledge I: $\forall I \subseteq X_{\mathsf{pp}} : \langle \mathfrak{K}_{\mathsf{pp}}, I \rangle$ (equivalent to $X_{\mathsf{pp}} \times W_{\mathsf{pp}}$)

Formally then, we define $\mathfrak{K}$ types through the grammar

$$\tau \equiv \mathbb{0} \mid \mathbb{1} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau^* \mid [\mathfrak{K}_{\mathsf{pp}}] \mid \langle \mathfrak{K}_{\mathsf{pp}}, I \rangle,$$

with the corresponding expression grammer being

$$E \equiv \top \mid \mathsf{inj}_1(E) \mid \mathsf{inj}_2(E) \mid (E_1, E_2) \mid \varepsilon \mid E_1 :: E_2 \mid [E]_{\mathfrak{K}_{\mathsf{pp}}} \mid \langle E \rangle^I_{\mathfrak{K}_{\mathsf{pp}}}.$$

Crucially, the types of messages may depend on prior interactions. This is particularly obvious with the set of input knowledge I, which will be defined as the set of all previously received $x : [\mathfrak{K}_{\mathsf{pp}}]$, however it also applies to pp itself, which may be provided from another component of the system. This allows for the secure sampling of public parameters, or delegating this to a common reference string (CRS). The typing rules are extended with the following two rules, where X_{pp} and W_{pp} are type variable:

$$\frac{\vdash x : X_{\mathsf{pp}} \qquad \vdash w : W_{\mathsf{pp}} \qquad (x, w) \in \mathcal{R}_{\mathsf{pp}}(I)}{\vdash \langle x, w \rangle^I_{\mathfrak{K}_{\mathsf{pp}}} : \langle \mathfrak{K}_{\mathsf{pp}}, I \rangle} \qquad \frac{\vdash x : X_{\mathsf{pp}}}{\vdash [x]_{\mathfrak{K}_{\mathsf{pp}}} : [\mathfrak{K}_{\mathsf{pp}}]}$$

3.2.2 Efficiently Indexable Sums and Products

In order to allow representing infinite randomness in public parameters, such as for the random oracle, infinite products with efficient indexing are required. We define the sum and product operator here to divide the domain into sets of increasingly large powers of two, which can be arranged as a binary tree. The final construction links these in a basic sequence, ensuring that any index i can be accessed in $\Theta(\log(i))$ operations.

$$\sum_{x \in X} f(x) \coloneqq \mathsf{effAgg}(X, f, +, \mathbb{0}, 0)$$
$$\prod_{x \in X} f(x) \coloneqq \mathsf{effAgg}(X, f, \times, \mathbb{1}, 0)$$

$$\begin{aligned}
\mathsf{effAgg}(\ \varepsilon, \cdot,\ \cdot, \tau,\ \cdot) &\coloneqq \tau \\
\mathsf{effAgg}(\ X, f, \diamond, \tau,\ i) &\coloneqq \mathsf{aggTree}(\mathsf{take}(2^i, X), f, \diamond, \tau, i) \diamond \\
&\qquad\quad \mathsf{effAgg}(\mathsf{drop}(2^i, X), f, \diamond, \tau, i + 1) \\
\mathsf{aggTree}([x], f,\ \cdot, \cdot, 0) &\coloneqq f(x) \\
\mathsf{aggTree}(\ \varepsilon, \cdot,\ \cdot, \tau,\ \cdot) &\coloneqq \tau \\
\mathsf{aggTree}(\ X, f, \diamond, \tau,\ i) &\coloneqq \mathsf{aggTree}(\mathsf{take}(2^{i-1}, X), f, \diamond, \tau, i - 1) \diamond \\
&\qquad\quad \mathsf{aggTree}(\mathsf{drop}(2^{i-1}, X), f, \diamond, \tau, i - 1),
\end{aligned}$$

where $\mathsf{take}(i, X)$ returns the sequence containing only the first i elements of X (or X itself, if $|X| \leq i$), $\mathsf{drop}(i, X)$ returns the sequence containing all other elements of X, such that $\mathsf{take}(i, X) \parallel \mathsf{drop}(i, X) = X$, and $\diamond$ stands in for one of $+$ and $\times$.

3.2.3 Typed Networks

We will consider networks of random systems (which can be considered as labelled graphs) as our basic object to define composition over.

Definition 3.2 (Cryptographic Networks). A typed cryptographic network is a set of nodes N, satisfying the following conditions:

1. Each node $n \in N$ is a tuple $n = (I_n, O_n, \tau_n, R_n, A_n)$ representing:
    - I_n a set of available input interfaces.
    - O_n a set of available output interfaces.

- τ_n: $I_n \cup O_n \to T$, a mapping from interfaces to their types.
- R_n, a $\left(\sum_{i \in I_n} \tau_n(i), \sum_{o \in O_n} \tau_n(o)\right)$ random system.
  (see Subsection 3.2.2 for a detailed description of sums over types)
- $A_n \subseteq I_n \cup O_n$, the subset of interfaces which behave adversarially.

2. Both input and output interfaces are unique within the network:

$$\forall a, b \in N : a \neq b \implies I_a \cap I_b = \varnothing \wedge O_a \cap O_b = \varnothing.$$

3. Matching input and output interfaces define directed channels in the implied network graph. Therefore, where $a, b \in N, i \in O_a \cap I_b$:

   - The interface types match: $\tau_a(i) = \tau_b(i)$.
   - The edges have a consistent adversariality: $i \in A_a \iff i \in A_b$.

We denote the set of all valid cryptographic networks by $*$.

This corresponds to a directed network graph whose vertices are nodes and whose edges connect output interfaces to their corresponding input interface.

Composing multiple such networks is a straightforward operation, achieved through set union. While the resulting network is not necessarily valid, as it may lead to uniqueness of interfaces being violated, it can be used to construct any valid network out of its components. We also make use of a disjoint union, $A \uplus B$, by which we mean the union of A and B, while asserting that A and B are disjoint. Due to the frequency of its use, we will allow omitting the disjoint union operator, that is, we write AB to denote $A \uplus B$.

Definition 3.3 (Unbound Interfaces). In a typed cryptographic network N, the sets of unbound input and output interfaces, written $I(N)$ and $O(N)$, respectively, are defined as the set of all tuples (i, τ) for which there exists $a \in N$ and $i \in I_a$ (resp. $i \in O_a$), where for all $b \in N$, $i \notin O_b$ (resp. $i \notin I_b$), with τ being defined as its type, $\tau_a(i)$. Furthermore, $IO_{\mathcal{H}}(N)$ is defined as the unbound honest interfaces: all $(i, \cdot) \in I(N) \cup O(N)$, where i is honest, that is, where $\forall a \in N : i \notin A_a$.

We can define a straightforward token-passing execution mechanism over typed cryptographic networks, which demonstrates how each network behaves as a single random system[4]. We primarily operate with networks instead of

[4]Termination is an issue here, in so far as the network may loop infinitely using message passing. We consider a non-terminating network to return the symbol $\perp$, although this might render the output uncomputable.

reducing them to a single random system to preserve their structure: It allows easily applying knowledge assumptions to each part and enables sharing components in parallel composition, a requirement for globality.

Definition 3.4 (Execution). A typed cryptographic network N, together with an ordering of $I(N)$ and $O(N)$ defines a random system through token-passing execution, with the input and output domains $\sum_{(\cdot,\tau)\in I(N)} \tau / \sum_{(\cdot,\tau)\in O(N)} \tau$, respectively. Execution is defined through a stateful passing of messages – any input to N will be targeted to some $(i,\cdot) \in I(N)$. The input is provided to the random system R_a, for which $i \in I_a$. Its output will be associated with an $o \in O_a$. If there exists a $b \in N$ such that $o \in I_b$, it is forwarded to R_b, continuing in a loop until no such node exists. At this point, the output is associated with $(o,\cdot) \in O(N)$ (note that, if $O(N) = \varnothing$, the corresponding random system cannot be defined, as it has an empty output domain) and is encoded to the appropriate part of the output domain.

In order to help with preventing interface clashes, we introduce a renaming operation.

Definition 3.5 (Renaming). For a cryptographic network N, renaming interfaces $a_1, \ldots, a_n$ to $b_1, \ldots, b_n$, is denoted by:

$$N[a_1/b_1, \ldots, a_n/b_n] \coloneqq \{\, m \in N \mid m[a_1/b_1, \ldots, a_n/b_n] \,\}.$$

Where, for $m = (I_m, O_m, \tau_m, \cdot, A_m)$, $m[a_1/b_1, \ldots, a_n/b_n]$ is defined by replacing each occurrence of a_i in the sets I_m, O_m and A_m with the corresponding b_i, as well as changing the domain of τ_m to accept b_i instead of a_i, with the same effect.

To ensure renaming does not introduce unexpected effects, we leave it undefined when any of the output names b_i are present in the network N and are not themselves renamed (i.e. no a_j exists such that $a_j = b_i$). Likewise, we prohibit renaming where multiple output names are equal. For a set of cryptographic networks, the same notation denotes renaming on each of its elements.

When talking about valid distinguishers, these are sets of cryptographic networks closed under internal renaming.

Definition 3.6 (Distinguisher Set). A set of distinguishers $\mathfrak{D} \subseteq *$ is any subset of $*$ which is closed under internal renaming: For any $D \in \mathfrak{D}, \vec{n} = a_1/b_1, \ldots, a_n/b_n$, where no a_i or b_i are in $I(D)$ or $O(D)$, $D[\vec{n}] \in * \implies D[\vec{n}] \in \mathfrak{D}$.

Composition is also defined for distinguisher sets. Given a set of networks $\mathfrak{D}$ and a network A, $\mathfrak{D}A$ is defined as the closure under internal renaming of $\{ DA \mid D \in \mathfrak{D} : DA \in * \}$. Observe that $*$ is closed under composition and therefore $*A \subseteq *$ for any $A \in *$. Renaming for distinguisher sets is defined similarly, allowing distinguisher sets to give special meaning to some *external* interfaces, but not to internal ones.

3.2.4 Observational Indistinguishability

Now that we have established the semantics of cryptographic networks, we can reason about their observational indistinguishability, defined through the statistical distances of their induced random systems combined with arbitrary distinguishers. The indistinguishability experiment is visualised in Figure 3.1.

Definition 3.7 (Observational Indistinguishability). Two cryptographic networks A and B are observationally indistinguishable with advantage ε with respect to the set of valid distinguishers $\mathfrak{D}$, written $A \overset{\varepsilon,\mathfrak{D}}{\sim} B$, if and only if:

- Their unbound inputs and outputs match: $I(A) = I(B) \wedge O(A) = O(B)$.
- For any network $D \in \mathfrak{D}$ for which DA and DB are both in $*$, with $I(DA) = I(DB) = (\cdot, \mathbb{1})$ and $O(DA) = O(DB) = (\cdot, 2)$, the statistical distance $\delta^{\mathfrak{D}}(A, B)$ is at most ε, where

$$\delta^{\mathfrak{D}}(A, B) \coloneqq \sup_{D \in \mathfrak{D}} \Delta^{D}(A, B)$$

$$\Delta^{D}(A, B) \coloneqq |\Pr(DA = 1) - \Pr(DB = 1)|.$$

  To simplify some corner cases, where $\forall D \in \mathfrak{D} : DA \notin * \vee DB \notin *$, we consider $\delta^{\mathfrak{D}}(A, B)$ to be 0 – in other words, we consider undefined behaviours indistinguishable.

The $\mathfrak{D}$ term is omitted if it is clear from the context.

Observe that observational indistinguishability claims can be weakened:

$$A \overset{\varepsilon,\mathfrak{D}_1}{\sim} B \wedge \mathfrak{D}_2 \subseteq \mathfrak{D}_1 \implies A \overset{\varepsilon,\mathfrak{D}_2}{\sim} B \tag{3.1}$$

Lemma 3.1 (Observational Renaming). *Observational indistinguishability is* closed under interface renaming*:*

$$\forall A, B \in *, \mathfrak{D} \subseteq *, \varepsilon, \vec{n} : A[\vec{n}], B[\vec{n}] \in * \wedge A \overset{\varepsilon,\mathfrak{D}}{\sim} B \implies A[\vec{n}] \overset{\varepsilon,\mathfrak{D}[\vec{n}]}{\sim} B[\vec{n}]$$

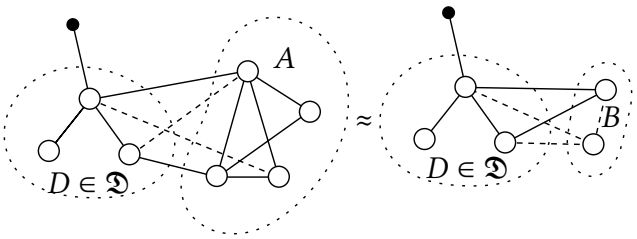


Figure 3.1: A visual representation of an example $A \overset{\mathfrak{D}}{\sim} B$ experiment, with solid lines representing honest interfaces and dashed representing adversarial interfaces.

Proof. By precondition, we know that $I(A) = I(B) \wedge O(A) = O(B)$, that $\delta^{\mathfrak{D}}(A, B) \leq \varepsilon$ and that $\mathfrak{D}$ is closed under renaming. As renaming is restricted by definition to not create any new connections, $I(A[\vec{n}]) = I(A)[\vec{n}] = I(B)[\vec{n}] = I(B[\vec{n}])$ and likewise for O. As $\mathfrak{D}$ remains unchanged, it remains to show that $\sup_{D \in \mathfrak{D}} |\Pr(DA[\vec{n}] = 1) - \Pr(DB[\vec{n}] = 1)| \leq \varepsilon$.

Consider how, for $D \in \mathfrak{D}$, $(DA)[\vec{n}]$ and $(DB)[\vec{n}]$, are related to $D'(A[\vec{n}])$ and $D'(B[\vec{n}])$. If $(DA)[\vec{n}]$ is well-defined, then for $D' = D[\vec{n}]$, then $(DA)[\vec{n}] = D'(A[\vec{n}])$. Moreover, for any $D' \in \mathfrak{D}$, there exists some internal renaming $\vec{m}$ such that $(D'[\vec{m}]A)[\vec{n}]$ and $(D'[\vec{m}]B)[\vec{n}]$ *are* well-defined, as the renaming $\vec{m}$ can remove the potential name clashes introduced by $\vec{n}$. As $\mathfrak{D}$ is closed under renaming, it is therefore sufficient to show that $\sup_{D \in \mathfrak{D}} |\Pr((DA)[\vec{n}] = 1) - \Pr((DB)[\vec{n}] = 1)| \leq \varepsilon$. As the execution semantics of $(DA)[\vec{n}]$ and $(DB)[\vec{n}]$ does not use interface names, this is equivalent to $\sup_{D \in \mathfrak{D}} |\Pr(DA = 1) - \Pr(DB = 1)| = \delta^{\mathfrak{D}}(A, B) \leq \varepsilon$. □

Lemma 3.2 (Observational Equivalence). *Observational indistinguishability is an equivalence relation: It is* ***transitive***[5] (3.2)*,* ***reflexive*** (3.3)*, and* ***symmetric*** (3.4)*. For all* $A, B, C \in *, \mathfrak{D} \subseteq *, \varepsilon_1, \varepsilon_2 \in \mathbb{R}$*:*

$$A \overset{\varepsilon_1,\mathfrak{D}}{\sim} B \wedge B \overset{\varepsilon_2,\mathfrak{D}}{\sim} C \implies A \overset{\varepsilon_1+\varepsilon_2,\mathfrak{D}}{\sim} C \tag{3.2}$$

$$A \overset{0,\mathfrak{D}}{\sim} A \tag{3.3}$$

$$A \overset{\varepsilon_1,\mathfrak{D}}{\sim} B \iff B \overset{\varepsilon_1,\mathfrak{D}}{\sim} A \tag{3.4}$$

Proof. We prove each part independently, given the well-known fact that statistical distance forms a pseudo-metric [Mau11].

[5] Technically, due to the error terms, the relation is not transitive, but obeys a triangle inequality and as a result it is also not an equivalence relation. We view this as a weak transitivity instead, as in practice, for negligible error terms, it behaves as such.

Transitivity. The equality of the input and output interfaces can be established by the transitivity of equality. The statistical distance is established through the triangle equality. Specifically, for all $D \in \mathfrak{D}$, $\Delta^D(A, C) \leq \Delta^D(A, B) + \Delta^D(B, C) \leq \varepsilon_1 + \varepsilon_2$. The only case where this is not immediate is if $DB \notin *$, which occurs in the case of an internal interface name collision – resolvable with renaming and use of Lemma 3.1. □

Reflexivity. By the reflexivity of equality for input and output interfaces and $\delta^{\mathfrak{D}}(A, A) = 0$ being established for pseudo-metrics. □

Symmetry. By the symmetry of equality and pseudo-metrics. □

Lemma 3.3 (Observational Subgraph Substitution). *Observational indistinguishability is closed under subgraph substitution.*

$$\forall A, B, C \in *, \mathfrak{D} \subseteq *, \varepsilon \in \mathbb{R}: A \overset{\varepsilon, \mathfrak{D}C}{\sim} B \iff CA \overset{\varepsilon, \mathfrak{D}}{\sim} CB$$

Proof. The equality of outgoing interfaces is trivially preserved under substitution, as the outgoing interfaces of A and B are the same by assumption.

We know that $\forall D \in \mathfrak{D}C: \Delta^D(A, B) \leq \varepsilon$. Suppose there existed a distinguisher $D \in \mathfrak{D}$ such that $\Delta^D(CA, CB) \geq \varepsilon$. Then, we can define $D' \in \mathfrak{D}C$ as DC, redrawing the boundary between distinguisher and network. By definition, $D' \in \mathfrak{D}C$, allowing us to conclude $\exists D' \in \mathfrak{D}C: \Delta^{D'}(A, B) \geq \varepsilon$, arriving at a contradiction. The proof runs analogously in the opposite direction. □

Corollary 3.1. *For $\mathfrak{D} = *$, observational indistinguishability has the following, simpler statement for closure under subgraph substitution:*

$$\forall A, B, C \in *, \varepsilon: A \overset{\varepsilon, *}{\sim} B \implies CA \overset{\varepsilon, *}{\sim} CB$$

3.2.5 Composably Secure Construction of Networks

(Composable) simulation-based security proofs are then proofs that there exists an extension to one network connecting only on adversarial interfaces, such that it is observationally indistinguishable to another. We visualise and provide an example of construction in Figure 3.2.

Definition 3.8 (Network Construction). A network $A \in *$ constructs another network $B \in *$ with respect to a distinguisher class $\mathfrak{D}$ with simulator $\alpha \in *$ and

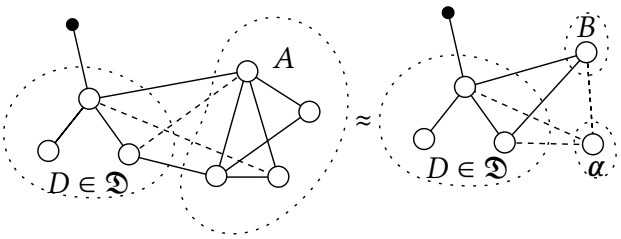


Figure 3.2: A visual representation of the $A \overset{\alpha,\mathfrak{D}}{\rightarrowtail} B$ experiment.

error $\varepsilon \in \mathbb{R}$, written $A \overset{\varepsilon,\alpha,\mathfrak{D}}{\rightarrowtail} B$, if and only if $A \overset{\varepsilon,\mathfrak{D}}{\sim} \alpha B$ and α and B have disjoint honest interfaces: $IO_{\mathcal{H}}(\alpha) \cap IO_{\mathcal{H}}(B) = \varnothing$. The $\mathfrak{D}$ term may be omitted when it is clear from the context, the α term may be omitted when it is of no interest, and the ε term may be omitted when it is negligible.

As with observational indistinguishability, network construction statements can be arbitrarily weakened. Furthermore, it is directly implied by indistinguishability:

$$A \overset{\varepsilon,\alpha,\mathfrak{D}_1}{\rightarrowtail} B \wedge \mathfrak{D}_2 \subseteq \mathfrak{D}_1 \implies A \overset{\varepsilon,\alpha,\mathfrak{D}_2}{\rightarrowtail} B \tag{3.5}$$

$$A \overset{\varepsilon,\mathfrak{D}}{\sim} B \implies A \overset{\varepsilon,\varnothing,\mathfrak{D}}{\rightarrowtail} B \tag{3.6}$$

Theorem 3.1 (Generalised Composition). *Network construction is composable, in that is satisfies* ***transitivity*** (3.7)*,* ***subgraph substitutability*** (3.8)*, and* ***renameability*** (3.9)*. For all* $A, B, C, \alpha, \beta \in *, \mathfrak{D} \subseteq *, \varepsilon_1, \varepsilon_2 \in \mathbb{R}, \vec{n}$*:*

$$A \overset{\varepsilon_1,\alpha,\mathfrak{D}}{\rightarrowtail} B \wedge B \overset{\varepsilon_2,\beta,\mathfrak{D}\alpha}{\rightarrowtail} C \wedge \alpha\beta C \in * \implies A \overset{\varepsilon_1+\varepsilon_2,\alpha\beta,\mathfrak{D}}{\rightarrowtail} C \tag{3.7}$$

$$A \overset{\varepsilon_1,\alpha,\mathfrak{D}C}{\rightarrowtail} B \wedge IO_{\mathcal{H}}(C) \cap IO_{\mathcal{H}}(\alpha B) = \varnothing \implies CA \overset{\varepsilon_1,\alpha,\mathfrak{D}}{\rightarrowtail} CB \tag{3.8}$$

$$A[\vec{n}], \alpha[\vec{n}]B[\vec{n}] \in * \wedge A \overset{\varepsilon_1,\alpha,\mathfrak{D}}{\rightarrowtail} B \implies A[\vec{n}] \overset{\varepsilon_1,\alpha[\vec{n}],\mathfrak{D}[\vec{n}]}{\rightarrowtail} B[\vec{n}] \tag{3.9}$$

Proof. We will prove each of the three properties separately.

Transitivity. By assumption, we know that $A \overset{\varepsilon_1,\mathfrak{D}}{\sim} \alpha B$ and $B \overset{\varepsilon_2,\mathfrak{D}\alpha}{\sim} \beta C$. By Lemma 3.3, we can conclude that $\alpha B \overset{\varepsilon_2,\mathfrak{D}}{\sim} \alpha\beta C$. By transitivity (Lemma 3.2), we conclude that $A \overset{\varepsilon_1+\varepsilon_2,\mathfrak{D}}{\sim} \alpha\beta C$.

Observe that β and C, as well as α and B have disjoint honest interfaces by assumption. As $B \overset{\varepsilon_2,\mathfrak{D}}{\sim} \beta C$, they have the same public-facing interfaces. As $\alpha\beta C$ is well-defined and as α and B have disjoint honest interfaces, so does α and βC.

From each of α, β, and C's honest interfaces being disjoint, we conclude that so are $\alpha\beta$ and C's. □

Closure under subgraph substitution. By assumption, we know $A \overset{\varepsilon_1,\mathfrak{D}C}{\sim} \alpha B$. By Lemma 3.3, we can conclude that $CA \overset{\varepsilon_1,\mathfrak{D}}{\sim} C\alpha B$. As composition is a disjoint union, it is commutative and therefore $C\alpha B = \alpha CB$. The interface disjointness requirement is satisfied by the precondition. □

Closure under renaming. By assumption, we know $A \overset{\varepsilon_1,\mathfrak{D}}{\sim} \alpha B$. By Lemma 3.1, we conclude that $A[\vec{n}] \overset{\varepsilon_1,\mathfrak{D}[\vec{n}]}{\sim} (\alpha B)[\vec{n}] = \alpha[\vec{n}]B[\vec{n}]$. As $\alpha[\vec{n}]B[\vec{n}] \in *$, both $\alpha[\vec{n}]$ and $B[\vec{n}]$ are in $*$. As the honesty of edges remains unaffected by subgraph substitution, name collisions are not introduced, the disjointness requirement is also satisfied. Combined, this implies network construction in the renamed setting. □

From the generalised composition theorem, which notably relies on modifying the distinguisher set (e.g. from $\mathfrak{D}$ to $\mathfrak{D}\alpha$ in (3.7)), we can infer operations similar to sequential and parallel composition in Constructive Cryptography, given $\mathfrak{D} = *$. For any $\mathfrak{D}$, identity also holds, due to the identity of indistinguishability, and indistinguishability lifting to construction.

Corollary 3.2 (Traditional Composition). *For $\mathfrak{D} = *$, honest network construction has the following, simpler statements for* ***universal transitivity*** (3.10) *and* ***universal closure under subgraph substitution*** (3.11). ***Identity*** (3.12) *holds regardless of $\mathfrak{D}$. For all $A, B, C, \alpha, \beta \in *, \varepsilon_1, \varepsilon_2 \in \mathbb{R}, \mathfrak{D} \subseteq *$:*

$$A \xrightarrow{\varepsilon_1,\alpha,*} B \wedge B \xrightarrow{\varepsilon_2,\beta,*} C \wedge \alpha\beta C \in * \implies A \xrightarrow{\varepsilon_1+\varepsilon_2,\alpha\beta,*} C \tag{3.10}$$

$$A \xrightarrow{\varepsilon_1,\alpha,*} B \wedge IO_{\mathcal{H}}(C) \cap IO_{\mathcal{H}}(\alpha B) = \varnothing \implies CA \xrightarrow{\varepsilon_1,\alpha,*} CB \tag{3.11}$$

$$A \xrightarrow{0,\varnothing,\mathfrak{D}} A \tag{3.12}$$

3.3 The Limited Composition of $\mathfrak{K}$-Networks

Having established a composition system which allows restricting the domain of permissible distinguishers and, having formalised the general notion of knowledge assumptions, we can now establish the main contribution in this chapter: Permitting extraction from knowledge assumptions within a composable setting.

We use a similar idea to that of "algebraic adversaries" in the Algebraic Group Model [FKL18], requiring random systems (recall Subsection 2.3.2) to output not only knowledge-implying objects, but also their corresponding witness. We then add new nodes to the network which gather all data extracted in this way in a central repository of knowledge for each knowledge assumption. Crucially, while the distinguisher supplies witnesses for all knowledge-implying objects it outputs, it is not capable of retrieving witnesses from other parts of the system.

Simulators are provided with read access to this repository, allowing the simulator to extract the knowledge it requires, but not any more about the behaviour of honest parties. The composition of constructions using knowledge assumptions is proven, provided the parts being composed do not both utilise the same knowledge assumption. In such a case, Theorem 3.1 provides a fallback for what needs to be proven, namely that the simulator of one system does not permit distinguishing in the other system. At a technical level, modifications to Definition 3.2 are needed to allow types to depend on previously transmitted values. We note the formal differences in Subsection 3.3.2. Furthermore, many of the statements in this section technically require some renaming to avoid internal name clashes and connect to the correct interfaces. These detract from the legibility of statements; we will therefore describe them less formally in this section, leaving the details to Subsection 3.3.6.

3.3.1 Knowledge Respecting Systems

The Algebraic Group Model [FKL18] popularised the idea of "algebraic" adversaries, which must adhere to outputting group elements through a representation describing how they may be constructed from input group elements. Security proofs in the AGM assume that all adversaries are algebraic and therefore the representation of group elements can be directly accessed in the reduction – by assumption it is provided by the adversary itself.

While this is equivalent to an extractor-based approach, for composition we will follow a similar "algebraic" approach. The premise is that for any random system R outputting (among other things) knowledge-implying objects in $\mathfrak{K}$, it is possible to construct an equivalent random system $\mathfrak{K}(R)$, which outputs the corresponding witnesses as well, provided each step of the random system is governed by a $\mathfrak{K}$-respecting algorithm.

Recall that a random system is an infinite sequence of probability distributions. As this is not in itself useful for applying Definition 3.1, we instead interpret them as an equivalence class over stateful, interactive, and probabilistic algorithms [LM20], with associated input and output types. For any such typed algorithm A and knowledge assumption $\mathfrak{K}_{\mathsf{pp}}$, A can be separated into A_1 and A_2, where A_1 outputs only a series of $[X_{\mathsf{pp}}]$ values and A_2 all the remaining information, such that A's output can be trivially reconstructed by inserting the $[X_{\mathsf{pp}}]$ values of A_1 into the gaps in A_2's outputs. Likewise, inputs can be split into the $\vec{I}$ and aux inputs used in Game 3.1. Given this, we can define when a random system is $\mathfrak{K}$-respecting. Each such system has a corresponding "$\mathfrak{K}$-lifted" system, which behaves "algebraically", in that it also output witnesses.

Definition 3.9 ($\mathfrak{K}$-Respecting Systems). A typed random system R is said to be $\mathfrak{K}$-respecting (or $R \in \mathsf{RespSys}_{\mathfrak{K}}$), if and only if its equivalence class of stateful probabilistic algorithms contains a stateful algorithm A that when split as described in Subsection 3.3.1 into A_1 and A_2, satisfies $A_1 \in \mathsf{Resp}_{\mathfrak{K}}$. For a set $\vec{\mathfrak{K}}$, $\mathsf{RespSys}_{\vec{\mathfrak{K}}} \coloneqq \bigcap_{\mathfrak{K} \in \vec{\mathfrak{K}}} \mathsf{RespSys}_{\mathfrak{K}}$.

Definition 3.10 ($\vec{\mathfrak{K}}$-Lifted Systems). A typed random system R induces a set of $\vec{\mathfrak{K}}$-lifted random systems. This is defined by replacing, for any $\mathfrak{K} = (\cdot, X, W, \mathcal{R}) \in \vec{\mathfrak{K}}$, any (part of) an output from R with type $[\mathfrak{K}_{\mathsf{pp}}]$ with (a part of) the output with type $\langle \mathfrak{K}_{\mathsf{pp}}, I_{\mathfrak{K}_{\mathsf{pp}}} \rangle$, where $I_{\mathfrak{K}_{\mathsf{pp}}}$ is constructed as the set of all prior inputs to R of type $[\mathfrak{K}_{\mathsf{pp}}]$. The output (part) $\langle x, w \rangle^{I_{\mathfrak{K}_{\mathsf{pp}}}}_{\mathfrak{K}_{\mathsf{pp}}}$ of the lifted system must be such that the equivalent output (part) on the unlifted system is $[x]_{\mathfrak{K}_{\mathsf{pp}}}$ and $(x, w) \in \mathcal{R}_{\mathfrak{K}_{\mathsf{pp}}}(I_{\mathfrak{K}_{\mathsf{pp}}})$ with overwhelming probability.

Theorem 3.2 ($\vec{\mathfrak{K}}$-Lifting is Possible). *For random systems $R \in \mathsf{RespSys}_{\vec{\mathfrak{K}}}$, at least one $\vec{\mathfrak{K}}$-lifting of R, denoted $\vec{\mathfrak{K}}(R)$, exists.*

Proof. Split R into algorithms $A_{\mathfrak{K}}$ for each $\mathfrak{K} \in \vec{\mathfrak{K}}$ and A_* for the remaining computation, such that each $A_{\mathfrak{K}}$ outputs only $[\mathfrak{K}]$ and A_* outputs no such values, as described above. Then, by Assumption 3.1, there exist corresponding extractors $\mathcal{X}_{\mathfrak{K}}$ for each $\mathfrak{K} \in \vec{\mathfrak{K}}$, such that given the same inputs $\mathcal{X}_{\mathfrak{K}}$ outputs witnesses to the knowledge-implying objects output by $A_{\mathfrak{K}}$.

Replace $A_{\mathfrak{K}}$ with $A'_{\mathfrak{K}}$, which runs both $A_{\mathfrak{K}}$ and $\mathcal{X}_{\mathfrak{K}}$, and outputs $\langle x, w \rangle_{\mathfrak{K}}$, where $[x]_{\mathfrak{K}}$ is the output of $A_{\mathfrak{K}}$ and w is the output of $\mathcal{X}_{\mathfrak{K}}$. When reassembled into a random system, this modification satisfies Definition 3.10. □

3.3.2 Basic Type Dependencies

Up to this point we have glossed over an inconsistency in the framework we presented: It relies on types depending on sets of input knowledge and public parameters, however types are statically defined, and set from initialisation. To circumvent this, we extend the definition of cryptographic networks with a limited support for type dependencies, just sufficient for the purposes of this chapter. To reason about the origin of public parameters, we first classify which interfaces can be used as a basis for other types. This definition and that of the network public parameters arise from are mutually recursive – they are nevertheless presented separately for clarity.

Definition 3.11 (Public-read Interface). A countably infinite set $A = \{(i_1, o_1), (i_2, o_2), \ldots\}$ is a public-read interface in a cryptographic network N if each of the following conditions hold:

1. All interfaces names are valid: $\forall j \in \mathbb{N} \colon \exists n_1, n_2 \in N \colon i_j \in I_{n_1} \wedge o_j \in O_{n_2}$.
2. The types of all input interfaces are $\mathbb{1}$.
3. The types of all output interfaces are equal.
4. Passing $\top$ to each input interface will:
   (a) Cause the corresponding output interface to output a value matching all others output by this public-read interface in the past, independent of which interface is queried.
   (b) Not impact any subsequent execution (that is, not change the system state).

An example of a common public-read interface is a common reference string, provided it has an explicit setup step, that is, the CRS is not selected on the first query. We make use of public-read interfaces by allowing them to parameterise types of other interfaces in the system, for instance to be used as public parameters in knowledge assumption types. We are primarily interested in infinite sets to ensure that arbitrarily many additional interfaces can be created, a fact exploited to ensure the uniqueness of interface names in $\mathfrak{R}$-lifting.

These changes cumulate in a fairly minor change in the definition of cryptographic networks and their execution, which does not affect the subsequent proofs and definitions presented in Section 3.2.

Definition 3.12 (Simply Dependently Typed Cryptographic Networks). A simply dependently typed cryptographic network N (over a set of knowledge assumptions $\vec{\mathfrak{K}}$) is defined as in Definition 3.2, with the following modifications:

1. Let R be a set of public read interfaces associated with N, and I be all interfaces in N. Then there exists a partial order $\prec \subset R \times (R \uplus I)$ indicating type dependencies.

2. If $A, B \in R$ and $A \prec B$, then for all $(\cdot, o) \in B$, $A \prec o$.

3. Nodes depending on a public-read interface must have access to it: $\forall n \in N, a \in I_n \cup O_n, A\colon A \prec a \implies \exists (i, o) \in A\colon o \in I_n \wedge i \in O_n$.

4. For each node $n \in N$ and knowledge assumption $\mathfrak{K} \in \vec{\mathfrak{K}}$, during execution, each message received on an interface in I_n is statefully recorded in $I^n_{\mathfrak{K}}$.

5. For all $n \in N, a \in I_n \cup O_n$, $\tau_n(a)$ returns a function taking the following inputs:

   - For each $\mathfrak{K} \in \vec{\mathfrak{K}}$, $I^n_{\mathfrak{K}}$.
   - For all $A \in R\colon A \prec a$, the output value of the public-read interface.

   The output of this function is the type of this interface *given a specific execution state* ($I_{\mathfrak{K}}$ and public-read interface values).

6. Interface types match if their concrete type matches at all possible system states, where type matching is relaxed to allow the output type to be a subset[6] of the input type. In particular, note that $I_1 \subseteq I_2 \implies \langle \mathfrak{K}, I_1 \rangle \subseteq \langle \mathfrak{K}, I_2 \rangle$.

Typically the easiest way to ensure that interfaces are matching in all possible states is to ensure they depend on the same public-read interface. The rest of Section 3.2 can be established analogously for simply dependently typed cryptographic networks, with the modification of using the above definition of interface matching.

Definition 3.13 (Public parameters). A public-read interface A supplies the public parameters for a knowledge assumption $\mathfrak{K} = (\mathsf{init}, \ldots)$ if and only if its

[6] Where subsets of types are constructed by $\langle \mathfrak{K}_{\mathsf{pp}}, I_1 \rangle \subseteq \langle \mathfrak{K}_{\mathsf{pp}}, I_2 \rangle \iff \mathcal{R}_{\mathfrak{K}_{\mathsf{pp}}}(I_1) \subseteq \mathcal{R}_{\mathfrak{K}_{\mathsf{pp}}}(I_2)$, and for all other defined naturally over all recursive types, for instance $\tau^* \subseteq \tau'^* \iff \tau \subseteq \tau'$.

output type is $\mathbb{1} + O$, where O is the type of the codomain of init, with $\mathsf{inj}_1(\top)$ indicating the public parameters are uninitialised, and the output value v satisfies the following criteria:

1. Initially, $v = \mathsf{inj}_1(\top)$.
2. The value v changes at most once during any execution and, if it does, it is distributed according to $\mathsf{inj}_2(\mathsf{init})$.

We will largely use this implicitly – for pp being the value supplied by such a public-read interface, we will write X_{pp}, W_{pp}, $\mathcal{R}_{\mathsf{pp}}$ and $\mathfrak{K}_{\mathsf{pp}}$ as usual – for $\mathsf{pp} = \mathsf{inj}_2(\mathsf{pp}')$, these are synonyms for $X_{\mathsf{pp}'}$, etc., while for $\mathsf{pp} = \mathsf{inj}_1(\top)$, the uninitialised state is captured by defining $X_{\mathsf{inj}_1(\top)} \coloneqq W_{\mathsf{inj}_1(\top)} \coloneqq \mathcal{R}_{\mathsf{inj}_1(\top)} \coloneqq \varnothing$.

3.3.3 Lifting Networks for Knowledge Extraction

The set of $\vec{\mathfrak{K}}$-respecting random systems $\mathsf{RespSys}_{\vec{\mathfrak{K}}}$, along with the transformation $\vec{\mathfrak{K}}(R)$ for any $R \in \mathsf{RespSys}_{\vec{\mathfrak{K}}}$, provides a means of lifting individual random systems. Applied to networks, it is clear something more is necessary – the lifting does not preserve the types of output interfaces, and to permit these to match again some additional changes need to be made to the networks. Looking forward, the lifted systems will interact with a separate, universal node REPO, which stores witnesses for the simulator to access.

We extend the notion of $\vec{\mathfrak{K}}$-respecting to apply to networks, a network is $\vec{\mathfrak{K}}$-respecting if and only if all vertices in it are also $\vec{\mathfrak{K}}$-respecting (we will use $\mathsf{RespNet}_{\vec{\mathfrak{K}}}$ as the corresponding set of $\vec{\mathfrak{K}}$-respecting networks[7]). In lifting networks in this set, not only is each individual node lifted, but all outgoing connections are connected to a new node, which we name CHARON, which acts as a relay; re-erasing witnesses, while also informing a central repository of knowledge (outside of this network) of any witnesses it processes. We take the name from the ferryman of the dead in ancient Greek mythology, who in our case demands his toll in knowledge rather than coins. For any $\vec{\mathfrak{K}}$-respecting network N, we define the lifting $\vec{\mathfrak{K}}(N)$ as follows:

Definition 3.14 (Network Lifting). The network lifting $\vec{\mathfrak{K}}(N)$ for any cryptographic network $N \in \mathsf{RespNet}_{\vec{\mathfrak{K}}}$ is defined to compose as expected. In particular,

[7] This set also forbids interface name clashes with REPO, ensuring this can be safely inserted and is a subset of $*$.

if there exists $\vec{\mathfrak{K}}', N' : N = \vec{\mathfrak{K}}'(N')$, then $\vec{\mathfrak{K}}(N)$ is defined as $(\vec{\mathfrak{K}} \cup \vec{\mathfrak{K}}')(N')$. Otherwise[8], $\vec{\mathfrak{K}}(N)$ is defined as consisting of nodes n' for each node $n \in N$, where $R_{n'} = \vec{\mathfrak{K}}(R_n)$ and each output interface is renamed to a unique[9] new interface name. For each output interface now named x and previously named y in N, $\vec{\mathfrak{K}}(N)$ contains a new node CHARON($\vec{\mathfrak{K}}$, adv), where adv denotes if the interface is adversarial, connected to free interfaces on the knowledge repository REPO and the public parameters for each knowledge assumption. Note that REPO is *not* part of the lifted network itself, which allows disjoint networks to remain disjoint when lifted.

Where CHARON is defined as follows:

Node CHARON($\vec{\mathfrak{K}}$, adv)

This node intercepts an outgoing message of $\vec{\mathfrak{K}}(R)$ and maps it to the corresponding message R would have output, as well as sending the additional witnesses to REPO($\mathfrak{K}$). adv indicates if this node should be adversarial or not. τ and τ' represent the arbitrary types of the interface in $\mathfrak{K}(R)$ and R respectively, differing only in that τ has instances of $[\mathfrak{K}]$ replaced with $\langle \mathfrak{K}, I \rangle$.

Interfaces and their types:

	Type	Description
a/b	τ/τ'	The input and output messages
$\mathrm{pp}_{o,\mathfrak{K}}/\mathrm{pp}_{i,\mathfrak{K}}$	$\tau_{\mathrm{pp},\mathfrak{K}}/\mathbb{1}$	Public parameters read interface (for all $\mathfrak{K} \in \vec{\mathfrak{K}}$)
$k_{i,\mathfrak{K}}/k_{o,\mathfrak{K}}$	$X_{\mathrm{pp}} \times W_{\mathrm{pp}}/\mathbb{1}$	The knowledge output interface (for all $\mathfrak{K} = (\cdot, X, W, \cdot) \in \vec{\mathfrak{K}}$, pp as received on $\mathrm{pp}_{o,\mathfrak{K}}$)

$I = \{a\} \cup \{\, k_{i,\mathfrak{K}} \mid \mathfrak{K} \in \vec{\mathfrak{K}} \,\}, O = \{b\} \cup \{\, k_{o,\mathfrak{K}} \mid \mathfrak{K} \in \vec{\mathfrak{K}} \,\}, A = \{a, b\}$ if adv, $\varnothing$ otherwise

---

When receiving x on interface a:

Recursively replace all $\langle x, w \rangle_{\mathfrak{K}_{\mathrm{pp}}}$ values in x with $[x]_{\mathfrak{K}_{\mathrm{pp}}}$ where the corresponding part of τ' does not have type $\langle \mathfrak{K}_{\mathrm{pp}}, \cdot \rangle$. Record the list of such values in $K_{\mathfrak{K}}$ for each $\mathfrak{K} \in \vec{\mathfrak{K}}$.

for $\mathfrak{K} \in \vec{\mathfrak{K}}$ **do**

  for $\langle x, w \rangle_{\mathfrak{K}_{\mathrm{pp}}} \in K_{\mathfrak{K}}$ **do**

[8] Note that this is well-founded recursion, due to the base-case of $\vec{\mathfrak{K}} = \varnothing$ and as the order in which knowledge assumptions are added does not affect CHARON or REPO.

[9] Where we assume uniqueness, this is assumed globally: In $\vec{\mathfrak{K}}(A)\vec{\mathfrak{K}}(B)$, the uniquely selected interface names should not clash, therefore being the same as $\vec{\mathfrak{K}}(AB)$.

output (x, w) on $k_{i,\mathfrak{K}}$

require response $\top$ on $k_{o,\mathfrak{K}}$

output x on b

The node REPO($\mathfrak{K}$) collects witnesses from CHARON and provides adversarial access to them. REPO allows for some variation. For instance, it could:

1. Return the set of all witnesses.
2. Return at most one witness.
3. Abort when no witness is available.
4. For recursive witnesses (such as those used in the AGM and KEA assumptions), consolidate the witness into a maximal one, by recursively resolving (INPUT, i) terms.

We focus on 1., as it is the simplest, although we also specify the case of 4., as it matches reality more closely. First, 1. is formally defined:

Node REPO($\mathfrak{K}$)

This node stores all transmitted witnesses in the network and permits adversarial querying of statements, returning all appropriate witnesses. Where pp is used, it is as received on the public parameter read interface pp_o.

State variables and initialisation values:

Variable	Description
$K\colon (X_{\mathrm{pp}} \times W_{\mathrm{pp}})^* \coloneqq \varepsilon$	List of acquired knowledge

Interfaces and their types:

	Type	Description
$k^j_{i,\mathfrak{K}}/k^j_{o,\mathfrak{K}}$	$X_{\mathrm{pp}} \times W_{\mathrm{pp}}/\mathbb{1}$	Knowledge inputs
$\mathrm{pp}_{o,\mathfrak{K}}/\mathrm{pp}_{i,\mathfrak{K}}$	$\tau_{\mathfrak{K}_{\mathrm{pp}}}/\mathbb{1}$	Public parameters
$x^j_{\mathfrak{K}}/w^j_{\mathfrak{K}}$	$X_{\mathrm{pp}}/W^*_{\mathrm{pp}}$	Witness request

$I = \{\, k^j_{i,\mathfrak{K}}, x^j_{\mathfrak{K}} \mid j \in \mathbb{N} \,\} \cup \{\mathrm{pp}_{o,\mathfrak{K}}\}$
$O = \{\, k^j_{o,\mathfrak{K}}, w^j_{\mathfrak{K}} \mid j \in \mathbb{N} \,\} \cup \{\mathrm{pp}_{i,\mathfrak{K}}\}$
$A = \{\, x^j_{\mathfrak{K}}, w^j_{\mathfrak{K}} \mid j \in \mathbb{N} \,\}$

When receiving (x, w) on interface k_i^j:

let $K \leftarrow (x, w), K$

output $\top$ on k_o^j

When receiving x on interface x^j:

let $w \leftarrow \varepsilon$

for (x', w') **in** K **do**

  if $x = x'$ **then let** $w \leftarrow w', w$

output w on w^j

4 can be defined as:

Node REPO($\mathfrak{K}$)

This node stores all transmitted witnesses in the network and permits adversarial querying of statements, returning all appropriate witnesses. Witnesses w are assumed to be representable as $(a, w_1, \ldots, w_n)$, where $a \notin W$ and $w_i \in W$. Any (INPUT, i), at any level of the tree, is recursively substituted with the first alternatively available witness, if possible. It is assumed that the result is still in W. Where pp is used, it is as received on the public parameter read interface pp_o.

State variables and initialisation values:

Variable	Description
$K\colon (X_{\text{pp}} \times W_{\text{pp}})^* \coloneqq \varepsilon$	List of acquired knowledge

---

Interfaces and their types:

	Type	Description
$k_{i,\mathfrak{K}}^j / k_{o,\mathfrak{K}}^j$	$X_{\text{pp}} \times W_{\text{pp}} / \mathbb{1}$	Knowledge inputs
$\text{pp}_{o,\mathfrak{K}} / \text{pp}_{i,\mathfrak{K}}$	$\tau_{\mathfrak{K}_{\text{pp}}} / \mathbb{1}$	Public parameters
$x_{\mathfrak{K}}^j / w_{\mathfrak{K}}^j$	$X_{\text{pp}} / W_{\text{pp}}^*$	Witness request

$I = \{ k_{i,\mathfrak{K}}^j, x_{\mathfrak{K}}^j \mid j \in \mathbb{N} \} \cup \{\text{pp}_{o,\mathfrak{K}}\}$
$O = \{ k_{o,\mathfrak{K}}^j, w_{\mathfrak{K}}^j \mid j \in \mathbb{N} \} \cup \{\text{pp}_{i,\mathfrak{K}}\}$
$A = \{ x_{\mathfrak{K}}^j, w_{\mathfrak{K}}^j \mid j \in \mathbb{N} \}$

When receiving (x, w) on interface k_i^j:

if $w \neq (\text{INPUT}, x)$ **then**

  let $K \leftarrow (x, w), K$

output $\top$ on k_o^j

When receiving x on interface x^j:

let $w \leftarrow \varepsilon$; $K' \leftarrow K$

while $K' = (x', w'), K''$ **do**

if $x = x'$ **then let** $w \leftarrow \mathsf{unwind}(w'), w$

let $K' \leftarrow K''$

output w on w^j

Helper procedures:

procedure $\mathsf{unwind}((a, w_1, \ldots, w_n))$

for i **in** $\mathbb{Z}_n$ **where** $\exists x\colon w_{i+1} = (\textsc{input}, x)$ **do**

for (x', w') **in** K **do**

if $x = x'$ **then**

let $w_{i+1} \leftarrow \mathsf{unwind}(w')$

break

return $(a, w_1, \ldots, w_n)$

The set of valid $\vec{\mathfrak{K}}$-distinguishers $\mathfrak{D}_{\vec{\mathfrak{K}}}$ is defined with respect to REPO, where we assume the choice of variation is made separately for each knowledge assumption. Informally, it ensures that all parts of the distinguisher are $\vec{\mathfrak{K}}$-lifted, and the distinguisher collects all witnesses in a central knowledge repository REPO, but does not retrieve witnesses from this, effectively only providing access to the simulator.

Definition 3.15 ($\vec{\mathfrak{K}}$-Distinguishers). The set of valid $\vec{\mathfrak{K}}$-distinguishers $\mathfrak{D}_{\vec{\mathfrak{K}}}$, for any set of knowledge assumptions $\vec{\mathfrak{K}}$, is defined as the closure under internal renaming of

$$\left\{ \vec{\mathfrak{K}}(N) \cup \bigcup_{\mathfrak{K} \in \vec{\mathfrak{K}}} \textsc{repo}(\mathfrak{K}) \;\middle|\; N \in \mathsf{RespNet}_{\vec{\mathfrak{K}}} \right\}.$$

Note that as $N \in \mathsf{RespNet}_{\vec{\mathfrak{K}}}$, it cannot directly connect to any of the REPO nodes.

As the number of REPO and public parameter interfaces may differ between the real and ideal world, we must normalise them before establishing indistinguishability. To do so, we wrap both worlds to contain an additional node $\perp$ which consumes all remaining interfaces, depending on the number already used. This node depends on which interfaces *are* consumed in the world, and we therefore formally also define a wrapper which roughly says

"consume all remaining interfaces", specifically for the REPO interfaces and public parameters. In the remainder of this section, we leave this wrapper as implicit, with it detailed in Subsection 3.3.6. Formally, the $\perp$ node is specified as:

Node $\perp(\vec{\mathfrak{K}}, \mathbf{n})$

This node does nothing, except connect to dangling $k_{i,\mathfrak{K}}/k_{o,\mathfrak{K}}$, $x_{\mathfrak{K}}/w_{\mathfrak{K}}$, and $\mathrm{pp}_{o,\mathfrak{K}}/\mathrm{pp}_{i,\mathfrak{K}}$ interfaces. $\mathbf{n}\colon \vec{\mathfrak{K}} \to \mathbb{N}^3$ represents the number of each interface *not* to connect to and we will denote $(a_{\mathfrak{K}}, b_{\mathfrak{K}}, c_{\mathfrak{K}}) \coloneqq \mathbf{n}(\mathfrak{K})$.

Interfaces and their types:

	Type	Description
$k^j_{o,\mathfrak{K}}/k^j_{i,\mathfrak{K}}$	$\mathbb{1}/X_{\mathrm{pp}} \times W_{\mathrm{pp}}$	Knowledge repository inputs
$\mathrm{pp}_{o,\mathfrak{K}}/\mathrm{pp}_{i,\mathfrak{K}}$	$\tau_{\mathfrak{K}_{\mathrm{pp}}}/\mathbb{1}$	Public parameters
$w^j_{\mathfrak{K}}/x^j_{\mathfrak{K}}$	X_{pp}	Witness requests

$I = \{\, k^{j+a_{\mathfrak{K}}}_{o,\mathfrak{K}}, w^{j+b_{\mathfrak{K}}}_{\mathfrak{K}}, \mathrm{pp}^{j+c_{\mathfrak{K}}}_{o,\mathfrak{K}} \mid j \in \mathbb{N} \,\}$
$O = \{\, k^{j+a_{\mathfrak{K}}}_{i,\mathfrak{K}}, x^{j+b_{\mathfrak{K}}}_{\mathfrak{K}}, \mathrm{pp}^{j+c_{\mathfrak{K}}}_{i,\mathfrak{K}} \mid j \in \mathbb{N} \,\}$
$A = \{\, x^{j+b_{\mathfrak{K}}}_{\mathfrak{K}}, w^{j+b_{\mathfrak{K}}}_{\mathfrak{K}} j \mid j \in \mathbb{N} \,\}$

When receiving any input:

abort

A minor result of interest can be obtained in the case that a cryptographic network does not make use of a knowledge assumption. Formally, we define this as $\vec{\mathfrak{K}}$-agnosticism.

Definition 3.16 ($\vec{\mathfrak{K}}$-(Semi-)Agnostic). A cryptographic network A is $\vec{\mathfrak{K}}$-agnostic if and only if no outputs with a component $y\colon [\mathfrak{K}_{\mathrm{pp}}]$ for any $\mathfrak{K}_{\mathrm{pp}} \in \vec{\mathfrak{K}}$ are ever made. A cryptographic network R is semi-$\mathfrak{K}$-agnostic if and only if any output with a component $y\colon [\mathfrak{K}_{\mathrm{pp}}]$ was previously received as an input.

Given these definitions, existing indistinguishability and constructions results between $\mathfrak{K}$-respecting networks can be lifted to equivalent results between the lifted networks with respect to $\mathfrak{K}$-distinguishers:

Lemma 3.4 (Indistinguishability Lifting). *If* $A_1 A_2 \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}_1}}{\sim} B_1 B_2$, *where for* $i \in \{1,2\}$, $A_i, B_i \in \mathrm{RespNet}_{\vec{\mathfrak{K}}_2}$, $\vec{\mathfrak{K}}_1 \cap \vec{\mathfrak{K}}_2 = \varnothing$, *and* $\vec{\mathfrak{K}} \coloneqq \vec{\mathfrak{K}}_1 \cup \vec{\mathfrak{K}}_2$, *then:*

$$A_1 A_2 \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}_1}}{\sim} B_1 B_2 \implies A_1 \vec{\mathfrak{K}}_2(A_2) \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\sim} B_1 \vec{\mathfrak{K}}_2(B_2).$$

Proof. Recall from Definition 3.7 that three conditions need to be satisfied for indistinguishability: a) Unbound interfaces must match, b) $\delta^{\mathfrak{D}}(A, B) \leq \varepsilon$, and c) the set of distinguishers must be closed under internal renaming.

For any $\vec{\mathfrak{K}}$, $\mathfrak{D}_{\vec{\mathfrak{K}}}$ is closed under internal renaming by definition. Furthermore, as the interfaces of $A_1 A_2$ and $B_1 B_2$ match by precondition and interfaces not related directly to the knowledge assumption are preserved (with only their types being modified equally). For this point we only need to consider the knowledge-supplying, public parameter and knowledge-querying interfaces, which will be equal due to the definition of $\perp$-lifting and $\vec{\mathfrak{K}}_2(\cdot)$ using the same types.
It remains to show

$$\sup_{D \in \mathfrak{D}_{\vec{\mathfrak{K}}}} \Delta^D(A_1 \vec{\mathfrak{K}}_2(A_2), B_1 \vec{\mathfrak{K}}_2(B_2)) =$$
$$\sup_{D \in \mathfrak{D}_{\vec{\mathfrak{K}}}} |\Pr(DA_1 \vec{\mathfrak{K}}_2(A_2) = 1) - \Pr(DB_1 \vec{\mathfrak{K}}_2(B_2) = 1)| \leq \varepsilon.$$

Consider how the behaviour of $DA_1 \vec{\mathfrak{K}}(A_1)$ and $DB_1 \vec{\mathfrak{K}}(B_2)$ differs from $D' A_1 A_2$ and $D' B_1 B_2$ respectively, where D' is D without REPO($\mathfrak{K}$) (with any internal connections being replaced with a dummy REPO with no other purpose) for $\mathfrak{K} \in \vec{\mathfrak{K}}_2$, not exporting public-parameter interfaces. A_2 lacks CHARON nodes which impart knowledge to D's REPO($\mathfrak{K}$) nodes (for $\mathfrak{K} \in \vec{\mathfrak{K}}_2$). However, as in $DA_1 \vec{\mathfrak{K}}_2(A_2)$ these nodes do not reveal any information to the distinguisher (as the distinguisher cannot connect to the knowledge exporting interfaces, and as A_1 is $\vec{\mathfrak{K}}_2$-respecting) and neither reveals any information to the network $A_1 \vec{\mathfrak{K}}_2(A_2)$ (due to the $\perp$-lifting forcing these to be a no-op), this additional mechanism has no impact on the behaviour. As a result, the output of $D' A_1 A_2$ equals that of $DA_1 \vec{\mathfrak{K}}_2(A_2)$. As $D' \in \mathfrak{D}_{\vec{\mathfrak{K}}_1}$, we have

$$|\Pr(D' A_1 A_2 = 1) - \Pr(D' B_1 B_2 = 1)| \leq \varepsilon \implies$$
$$|\Pr(DA_1 \vec{\mathfrak{K}}_2(A_2) = 1) - \Pr(DB_1 \vec{\mathfrak{K}}_2(B_2) = 1)| \leq \varepsilon. \qquad \square$$

Lemma 3.5 (Construction Lifting). *For $A_{1,2}, B_{1,2}, \alpha_{1,2} \in \mathsf{RespNet}_{\vec{\mathfrak{K}}_2}$ and $\vec{\mathfrak{K}}_1, \vec{\mathfrak{K}}_2$ where $\vec{\mathfrak{K}}_1 \cap \vec{\mathfrak{K}}_2 = \varnothing$ and $\vec{\mathfrak{K}} \coloneqq \vec{\mathfrak{K}}_1 \cup \vec{\mathfrak{K}}_2$:*

$$A_1 A_2 \xrightarrow{\varepsilon, \alpha_1 \alpha_2, \mathfrak{D}_{\vec{\mathfrak{K}}_1}} B_1 B_2 \implies A_1 \vec{\mathfrak{K}}_2(A_2) \xrightarrow{\varepsilon, \alpha_1 \vec{\mathfrak{K}}_2(\alpha_2), \mathfrak{D}_{\vec{\mathfrak{K}}}} B_1 \vec{\mathfrak{K}}_2(B_2).$$

Proof. Recall from Definition 3.8 that network construction $A \xrightarrow{\varepsilon, \alpha, \mathfrak{D}} B$ has two separate conditions: α and B must have disjoint honest unbound interfaces, and

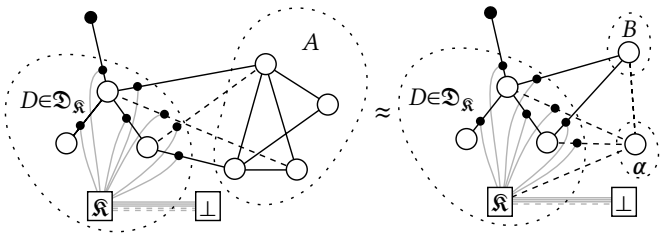


Figure 3.3: A visual representation of the $A \overset{\alpha,\mathfrak{D}_{\mathfrak{K}}}{\rightarrowtail} B$ experiment. The small points denote CHARON($\mathfrak{K}$) nodes, while $\mathfrak{K}$ denotes the REPO($\mathfrak{K}$) node. Public parameters have been omitted. Note that outside of D CHARON nodes are permitted, but not required.

$A \overset{\varepsilon,\mathfrak{D}}{\sim} \alpha B$. From the precondition, we know that α' and B' (defined as above) have disjoint honest interfaces. Interface names do not get changed through the $\vec{\mathfrak{K}}_2$ lifting, however new interfaces do get added. There is no clash in these interfaces, due to the global uniqueness requirement in the lifting. Furthermore, the $\bot$-lifting will be the same for all $\mathfrak{K} \in \vec{\mathfrak{K}}_1$ and normalise the interfaces available for $\mathfrak{K} \in \vec{\mathfrak{K}}_2$.
It remains to show:

$$A_1\vec{\mathfrak{K}}_2(A_2) \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\sim} \alpha_1\vec{\mathfrak{K}}_2(\alpha_2)B_1\vec{\mathfrak{K}}_2(B_2),$$

which by Lemma 3.4 follows from $A_1A_2 \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}_1}}{\sim} \alpha_1B_1\alpha_2B_2$, which in turn holds as part of the precondition. □

We visualise the construction experiment against a knowledge-respecting distinguisher set $\mathfrak{D}_{\mathfrak{K}}$ in Figure 3.3. This may be contrasted with Figure 3.2, which does not have REPO($\mathfrak{K}$) and does not allow the simulator to extract.

Lemma 3.6 ($\mathfrak{D}_{\vec{\mathfrak{K}}}$ Closure). *$\mathfrak{D}_{\vec{\mathfrak{K}}}$ is closed under sequential composition with lifted (with respect to $\vec{\mathfrak{K}}$) networks in* $\mathsf{RespNet}_{\vec{\mathfrak{K}}}$*:* $\forall R \in \mathsf{RespNet}_{\vec{\mathfrak{K}}} \colon \mathfrak{D}_{\vec{\mathfrak{K}}}\vec{\mathfrak{K}}(R) \subseteq \mathfrak{D}_{\vec{\mathfrak{K}}}$

Proof. Follows immediately from $\mathsf{RespNet}_{\vec{\mathfrak{K}}}$ being closed under set union, and Definition 3.15 stating that any $\vec{\mathfrak{K}}$-lifted network has a corresponding distinguisher in $\mathfrak{D}_{\vec{\mathfrak{K}}}$. □

As a stricter set of knowledge assumptions corresponds to a smaller set of permissible distinguishers, indistinguishability and construction results can be transferred to larger sets of knowledge assumptions. A proof without knowledge assumptions is clearly ideal – it still holds, regardless which knowledge assumptions are added.

Lemma 3.7 (Knowledge Weakening). *In addition to weakening with respect to a subset of distinguishers being possible, weakening is also possible for distinguishers with a greater set of knowledge assumptions. For all $A, B, C, \alpha \in *$, $\vec{\mathfrak{K}}_1, \vec{\mathfrak{K}}_2$, where $\vec{\mathfrak{K}}_1 \subseteq \vec{\mathfrak{K}}_2$:*

$$A \overset{\varepsilon, \mathfrak{D}_{\vec{\mathfrak{K}}_1} C}{\sim} B \implies A \overset{\varepsilon, \mathfrak{D}_{\vec{\mathfrak{K}}_2} C}{\sim} B \tag{3.13}$$

$$A \overset{\varepsilon, \alpha, \mathfrak{D}_{\vec{\mathfrak{K}}_1} C}{\rightarrowtail} B \implies A \overset{\varepsilon, \alpha, \mathfrak{D}_{\vec{\mathfrak{K}}_2} C}{\rightarrowtail} B \tag{3.14}$$

Proof. For (3.13), it is sufficient to show that

$$\sup_{D \in \mathfrak{D}_{\vec{\mathfrak{K}}_1} C} \Delta^D(A, B) \geq \sup_{D \in \mathfrak{D}_{\vec{\mathfrak{K}}_2} C} \Delta^D(A, B)$$

For this it is sufficient to show that every $D \in \mathfrak{D}_{\vec{\mathfrak{K}}_2} C$ has an equivalent $D' \in \mathfrak{D}_{\vec{\mathfrak{K}}_1} C$:

$$\forall D \in \mathfrak{D}_{\vec{\mathfrak{K}}_2} C \colon \exists D' \in \mathfrak{D}_{\vec{\mathfrak{K}}_1} C \colon DA = D'A \wedge DB = D'B.$$

For each distinguisher D in $\mathfrak{D}_{\vec{\mathfrak{K}}_2} C$, it consists of: a) C itself, b) $\textsc{repo}(\mathfrak{K})$ nodes for every $\mathfrak{K} \in \vec{\mathfrak{K}}_2$, and c) $\vec{\mathfrak{K}}_2(A)$ nodes for some $A \in \mathsf{RespNet}_{\vec{\mathfrak{K}}_2}$. For D' in $\mathfrak{D}_{\vec{\mathfrak{K}}_1}$ the same applies, however with fewer $\textsc{repo}(\mathfrak{K})$ nodes and a $\vec{\mathfrak{K}}_1(A)$ wrapping for $A \in \mathsf{RespNet}_{\vec{\mathfrak{K}}_1}$ instead. As $\vec{\mathfrak{K}}_1 \subseteq \vec{\mathfrak{K}}_2$, $\mathsf{RespNet}_{\vec{\mathfrak{K}}_1} \supseteq \mathsf{RespNet}_{\vec{\mathfrak{K}}_2}$, and it is therefore sufficient to show that the different wrapper and lack of additional $\textsc{repo}$ nodes does not change the behaviour. This follows directly as the effect of the additional wrapper in $\vec{\mathfrak{K}}_2 \setminus \vec{\mathfrak{K}}_1$ can be emulated without changing whether a node is $\vec{\mathfrak{K}}_1$-respecting, and likewise $\textsc{repo}$ nodes are trivially $\vec{\mathfrak{K}}_1$-respecting, as they do not produce knowledge-implying objects themselves. It follows that for every distinguisher $D \in \mathfrak{D}_{\vec{\mathfrak{K}}_2}$ we can construct a semantically identical distinguisher $D' \in \mathfrak{D}_{\vec{\mathfrak{K}}_1}$.

(3.14) follows directly from (3.13) and Definition 3.8. □

Lemma 3.8 (Agnostic Indistinguishability). *For $\vec{\mathfrak{K}}_1 \subseteq \vec{\mathfrak{K}}_2$, any $\vec{\mathfrak{K}}_1$-agnostic (resp. semi-$\vec{\mathfrak{K}}_1$-agnostic) network A and $C \in *$, where $\mathfrak{K} \in \vec{\mathfrak{K}}_1$ uses a basic (resp. recursive unwinding) $\textsc{repo}(\mathfrak{K})$ node:*

$$A \overset{0, \mathfrak{D}_{\vec{\mathfrak{K}}_2} C}{\sim} \vec{\mathfrak{K}}_1(A)$$

Proof. We consider the semantic behaviour of DCA and $DC\vec{\mathfrak{K}}_1(A)$ for all $D \in \mathfrak{D}_{\vec{\mathfrak{K}}_1}$. First, consider the case of A being $\vec{\mathfrak{K}}_1$-agnostic. In this case, the $\vec{\mathfrak{K}}_1$ wrapper around A has no impact on the semantics – the additional $\textsc{Charon}$ nodes give no information to $\textsc{repo}$, as there is no information to relay. Their behaviour is

therefore simply to relay – the same as when they are absent. As a result DCA behaves exactly like $DC\vec{\mathfrak{K}}_1(A)$, implying a statistical distance of zero.

If A is only semi-$\vec{\mathfrak{K}}_1$-agnostic, CHARON *does* include some witnesses, however they are always (INPUT, $\cdot$). Combined with the fact that these are actively ignored by the recursive unwinding REPO, this again has no impact on semantics. Through the same reasoning, it is possible to mix fully-$\mathfrak{K}$-agnostic and semi-$\mathfrak{K}$ agnostic for different knowledge assumptions $\mathfrak{K} \in \vec{\mathfrak{K}}_1$, provided the corresponding REPO($\mathfrak{K}$) are basic and recursive unwinding respectively. □

3.3.4 A Restricted Composition Theorem

The rules established in Theorem 3.1 still hold, and it is clear why a simplification as in Corollary 3.2 is not possible – it assumes that the distinguisher set $\mathfrak{D}$ is closed under sequential composition with simulators and networks, which is not the case for $\mathfrak{D}_{\vec{\mathfrak{K}}}$.

Theorem 3.1 already provides a sufficient condition for what needs to be proven to enable this composition, however we can go a step further: While $\mathfrak{D}_{\vec{\mathfrak{K}}}$ is not closed under sequential composition with arbitrary networks, it *is* closed under sequential composition with knowledge-lifted networks. We can use this fact to establish a simplified composition theorem when composing with a $\vec{\mathfrak{K}}$-lifted proof or network component. We observe that this implies composition with proofs which do not utilise knowledge assumptions, as they are isomorphic to $\vec{\mathfrak{K}} = \varnothing$. In particular, Constructive Cryptography proofs directly imply construction in the context of this paper as well, and can therefore be composed with protocols utilising our framework freely.

Theorem 3.3 (Knowledge Composition). *When composing proofs against $\vec{\mathfrak{K}}_1$ or $\vec{\mathfrak{K}}_2$ distinguishers, where $\vec{\mathfrak{K}}_1 \cap \vec{\mathfrak{K}}_2 = \varnothing$ and $\vec{\mathfrak{K}} \coloneqq \vec{\mathfrak{K}}_1 \cup \vec{\mathfrak{K}}_2$, the following simplified composition rules of* ***transitivity*** *(3.15) and* ***subgraph substitution*** *(3.16) apply. For all $A, B, \alpha \in \mathsf{RespNet}_{\vec{\mathfrak{K}}_2}$, $F \in \mathsf{RespNet}_{\vec{\mathfrak{K}}}$, $C, D, E, \beta, \gamma \in *, \varepsilon, \varepsilon_1, \varepsilon_2$.*

$$\left[\begin{array}{ccc} A & \overset{\varepsilon_1,\alpha,\mathfrak{D}_{\vec{\mathfrak{K}}_1}}{\rightarrowtail} & B \\ & \wedge & \\ B & \overset{\varepsilon_2,\beta,\mathfrak{D}_{\vec{\mathfrak{K}}_2}}{\rightarrowtail} & C \end{array}\right] \wedge \alpha\beta C \in * \implies A \overset{\varepsilon_1+\varepsilon_2,\vec{\mathfrak{K}}_2(\alpha)\beta,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\rightarrowtail} C \tag{3.15}$$

$$D \overset{\varepsilon,\gamma,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\rightarrowtail} E \wedge IO_{\mathcal{H}}(F) \cap IO_{\mathcal{H}}(\gamma E) = \varnothing \implies \vec{\mathfrak{K}}(F)D \overset{\varepsilon,\gamma,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\rightarrowtail} \vec{\mathfrak{K}}(F)E \tag{3.16}$$

Proof. The proof is done in parts.

Transitivity. By Lemma 3.6 and (3.5):

$$B \overset{\varepsilon_2,\beta,\mathfrak{D}_{\vec{\mathfrak{K}}_2}}{\rightarrowtail} C \implies B \overset{\varepsilon_2,\beta,\mathfrak{D}_{\vec{\mathfrak{K}}_2}\vec{\mathfrak{K}}_2(\alpha)}{\rightarrowtail} C.$$

By Lemma 3.5 and preconditions, $\vec{\mathfrak{K}}_2(A) \overset{\varepsilon_1,\vec{\mathfrak{K}}_2(\alpha),\mathfrak{D}_{\vec{\mathfrak{K}}}}{\rightarrowtail} \vec{\mathfrak{K}}_2(B)$ and, by Lemma 3.7 (3.14), $B \overset{\varepsilon_2,\beta,\mathfrak{D}_{\vec{\mathfrak{K}}}\vec{\mathfrak{K}}_2(\alpha)}{\rightarrowtail} C$. Therefore:

$$A \overset{\varepsilon_1,\vec{\mathfrak{K}}_2(\alpha),\mathfrak{D}_{\vec{\mathfrak{K}}}}{\rightarrowtail} B \overset{\varepsilon_2,\beta,\mathfrak{D}_{\vec{\mathfrak{K}}}\vec{\mathfrak{K}}_2(\alpha)}{\rightarrowtail} C,$$

and by Theorem 3.1 (3.7) (and $\alpha\beta C \in *$ implying $\vec{\mathfrak{K}}_2(\alpha)\beta C \in *$)

$$A \overset{\varepsilon_1+\varepsilon_2,\vec{\mathfrak{K}}_2(\alpha)\beta,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\rightarrowtail} C.$$ □

Subgraph substitution. By Lemma 3.6, $\mathfrak{D}_{\vec{\mathfrak{K}}}\vec{\mathfrak{K}}(C) \subseteq \mathfrak{D}_{\vec{\mathfrak{K}}}$. Therefore, the precondition can be weakened (3.5) to $A \overset{\varepsilon,\alpha,\mathfrak{D}_{\vec{\mathfrak{K}}}\vec{\mathfrak{K}}(C)}{\rightarrowtail} B$. The rest follows by Theorem 3.1 Equation (3.8) and the honest networks intersection not being affected by the $\vec{\mathfrak{K}}$-lifting. □

3.3.5 Reusing Knowledge Assumptions

Theorem 3.3 and its supporting lemmas prominently require disjoint sets of knowledge assumptions. The primary reason for this lies in the definition of $\vec{\mathfrak{K}}$ using the union of the knowledge assumptions $\vec{\mathfrak{K}}_1$ and $\vec{\mathfrak{K}}_2$ – all statements could also be made using a disjoint union here instead. If knowledge assumptions were not disjoint, this would place an unreasonable constraint on the distinguisher however: It would prevent it from copying information from one instance of a knowledge assumption to another instance of the same knowledge assumption, something any adversary is clearly capable of doing.

Equality for knowledge assumptions is not really well defined, and indeed knowledge assumptions may be related. The disjointness requirement is therefore more a statement of intent than an actual constraint and we stress the importance of it for reasonably constraining the distinguisher set here: If the dis-

tinguisher is constrained with respect to two instances of knowledge assumptions which are related, it may not be permitted to copy from one two to another for instance, an artificial and unreasonable constraint.

Care must be taken that knowledge stemming from one knowledge assumption does not give an advantage in another. In many – but not all – cases this is easy to establish, for instance, we conjecture that multiple instances with the AGM with independently sampled groups are sufficiently independent. If this care is not taken, the union of two knowledge assumptions may be greater than the sum of its parts, as using both together prevents the distinguisher from exploiting structural relationships between the two, something a real adversary may do.

3.3.6 Formal Renamings and Liftings in Section 3.3

In Definition 3.14, $\textsc{Charon}(\vec{\mathfrak{K}}, \mathrm{adv})$ nodes have the renaming $[a/x, b/y][k_{o,\mathfrak{K}}/k^j_{i,\mathfrak{K}}, k_{i,\mathfrak{K}}/k^j_{o,\mathfrak{K}}, \mathrm{pp}_{i,\mathfrak{K}}/i^l_{\mathfrak{K}}, \mathrm{pp}_{i,\mathfrak{K}}/o^l_{\mathfrak{K}} \mid \mathfrak{K} \in \vec{\mathfrak{K}}]$ applied to them, where j and l are uniquely assigned, and $(i^l_{\mathfrak{K}}, o^l_{\mathfrak{K}})$ are members of the public-read interface providing $\mathfrak{K}$'s public parameters, as defined in Subsection 3.3.2, Definition 3.13. If this public-parameter interface is already fully used, by the nature of countable infinity, space can be made by partial renaming.

In Definition 3.15, $\textsc{repo}(\mathfrak{K})$ has the renaming $[\mathrm{pp}_{i,\mathfrak{K}}/\mathrm{pp}^n_{i,\mathfrak{K}}, \mathrm{pp}_{o,\mathfrak{K}}/\mathrm{pp}^n_{o,\mathfrak{K}}]$ applied to it, for any $n \in \mathbb{N}$, selected separately for each knowledge assumption $\mathfrak{K}$.

If the distinguisher consumes the first n knowledge-supplying REPO interfaces and m public-parameter interfaces, and the network A consumes the next a of the former, b of the latter, and c of the knowledge-query interfaces, then $\vec{\mathfrak{K}}^{\perp}(A, \mathbf{n})$, where $\mathbf{n}$ encodes the connections used by the distinguisher, will use all countably infinite interfaces through a new $\perp$ instance. This normalises the connections between multiple different resources.

Definition 3.17 ($\perp$-Lifting). The lifting $\vec{\mathfrak{K}}^{\perp}(A, \mathbf{n})$ for any network A using $a_{\mathfrak{K}}$ knowledge-supplying interfaces (for $\mathfrak{K}$), $b_{\mathfrak{K}}$ public-parameter interfaces (for $\mathfrak{K}$), and $c_{\mathfrak{K}}$ knowledge-query interfaces (for $\mathfrak{K}$), and any $\mathbf{n}\colon \vec{\mathfrak{K}} \to \mathbb{N}^2$ is defined as $A' \cup \{\perp(\vec{\mathfrak{K}}, \mathbf{n}')\}$, where: A' is A with all the above $a_{\mathfrak{K}}$ interfaces renamed to fall between $a_{D,\mathfrak{K}} + 1$ and $a_{D,\mathfrak{K}} + a_{\mathfrak{K}}$, and likewise with $b_{\mathfrak{K}}$ and $c_{\mathfrak{K}}$ to be between 1 and $c_{\mathfrak{K}}$. $\mathbf{n}'$ is defined by $\mathbf{n}'(\mathfrak{K}) = (a_{D,\mathfrak{K}} + a_{\mathfrak{K}}, b_{D,\mathfrak{K}} + b_{\mathfrak{K}}, c_{\mathfrak{K}})$, with $(a_{D,\mathfrak{K}}, b_{D,\mathfrak{K}}) = \mathbf{n}(\mathfrak{K})$.

In Lemma 3.4, a more formal statement of

$$A_1 A_2 \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}_1}}{\sim} B_1 B_2 \implies A_1 \vec{\mathfrak{K}}_2(A_2) \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\sim} B_1 \vec{\mathfrak{K}}_2(B_2)$$

using $\perp$-lifting is that $\forall(\mathbf{n}\colon \vec{\mathfrak{K}}_2 \to \mathbb{N}^2)$:

$$\text{let } A' = \vec{\mathfrak{K}}_2^{\perp}(\vec{\mathfrak{K}}_2(A_2), \mathbf{n}), B' = \vec{\mathfrak{K}}_2^{\perp}(\vec{\mathfrak{K}}_2(B_2), \mathbf{n}) \text{ in } A_1 A' \overset{\varepsilon,\mathfrak{D}_{\vec{\mathfrak{K}}}}{\sim} B_1 B'$$

The simplified notation for construction without $\perp$-lifting is stated with renaming below:

$$\begin{aligned}
&\forall(\mathbf{n}\colon \vec{\mathfrak{K}} \to \mathbb{N}^2)\colon \exists B', \alpha', \vec{n}_1, \vec{n}_2, \mathbf{n}'\colon \\
&\alpha' B' = \vec{\mathfrak{K}}^{\perp}(\alpha B, \mathbf{n}) \wedge \\
&\alpha' = \alpha[\vec{n}_1] \wedge B' = B[\vec{n}_2] \cup \{\perp(\vec{\mathfrak{K}}, \mathbf{n}')\} \wedge \\
&\vec{\mathfrak{K}}^{\perp}(A, \mathbf{n}) \xrightarrow{\varepsilon,\alpha',\mathfrak{D}_{\vec{\mathfrak{K}}}} B'.
\end{aligned}$$

These modifications are necessary to ensure the renaming to use the first $\mathbf{n}$ instances can be distributed across α and B.

3.4 A Composable SNARK

To demonstrate the usefulness of this framework, we will showcase an example of how it can lift existing results to composability. For brevity, we sketch the approach instead of providing it in full detail. Specifically, we sketch how, using the bilinear AGM knowledge assumption $\mathfrak{K}_{\text{bAGM}}$ defined in Subsection 3.1.1 and an updateable reference string (which will be the focus of Chapter 4, but which we will briefly use here), we can implement a *succinct* NIZK functionality.

We rely on the results of Baghery et al. [BKSV21], which demonstrate that Groth's zk-SNARK [Gro16] has simulation extractability, with CøCø [KZM+15] demonstrating how – once extractability is given – the property-based definition can be lifted to a simulation-secure one. We conjecture that simulation extractability holds in the AGM for most zk-SNARKs, however it has not been proven in most cases.

Once our proof sketch is complete, we also give a clear example of why universal composition is not possible with knowledge assumptions: Specifically, we construct a complementary ideal network and simulator which clearly violates the zero-knowledge properties of the NIZK, and allows distinguishing the real

and ideal worlds. We stress that this is only possible due to it extracting from the same knowledge assumption.

3.4.1 Construction

Our construction consists of a real and of an ideal world. Throughout the construction, we assume a set of n parties, identified by an element in $\mathbb{Z}_n$. We assume static corruption with at least one honest party – specifically we assume a set of adversaries $\mathcal{A} \subset \mathbb{Z}_n$ and a corresponding set of honest parties $\mathcal{H} \coloneqq \mathbb{Z}_n \setminus \mathcal{A}$. These sets cannot be used in the protocols themselves, but are known to the distinguisher and non-protocol nodes (that is, can be used to define ideal behaviour). In all node specifications except $\mathbb{G}$, public-parameter interfaces are omitted and should be assumed.

$\mathfrak{K}_{\text{bAGM}}$ Parameters. In both worlds, the group initialisation is available as a common reference string and is specified as the node $\mathbb{G}$:

Node $\mathbb{G}$

This node provides the initialisation of $\mathfrak{K}_{\text{bAGM}}$. We assume the domain of init is $\tau_{\text{pp},\mathfrak{K}_{\text{bAGM}}}$.

State variables and initialisation values:

Variable	Description
$\text{pp}\colon \mathbb{1} + \tau_{\text{pp},\mathfrak{K}_{\text{bAGM}}} \coloneqq \text{inj}_1(\top)$	Public parameters

Interfaces and their types:

	Type	Description
$\text{init}_i/\text{init}_o$	$\mathbb{1}/\mathbb{1}$	Initialisation
$\text{pp}_i^j/\text{pp}_o^j$	$\mathbb{1}/\mathbb{1} + \tau_{\text{pp},\mathfrak{K}_{\text{bAGM}}}$	Public parameter queries

$I = \{\text{init}_i\} \cup \{\ \text{pp}_i^j \mid j \in \mathbb{N}\ \}$

$O = \{\text{init}_o\} \cup \{\ \text{pp}_o^j \mid j \in \mathbb{N}\ \}$

$A = \varnothing$

When receiving $\top$ *on interface* init_i*:*

let $\text{pp} \xleftarrow{*} \text{inj}_2(\text{init})$

output $\top$ on init_o

When receiving $\top$ *on interface* pp_i^j*:*

output pp on pp_o^j

SNARKs. The ideal world consists of a proof-malleable NIZK node (NIZK), found below.

Node NIZK

A proof-malleable NIZK for a relation $\mathcal{R}$. Assumed are the following types: a) X for statements, W for witnesses, Π for proofs.

State variables and initialisation values:

Variable	Description
$\pi\colon (X \times \Pi)^* \coloneqq \varepsilon$	Accepted proofs
$\overline{\pi}\colon (X \times \Pi)^* \coloneqq \varepsilon$	Rejected proofs

---

Interfaces and their types:

	Type	Description
$\mathsf{prove}_i^j/\mathsf{prove}_o^j$	$X \times W/\mathbb{1} + \Pi$	Proving
$\mathsf{verify}_i^j/\mathsf{verify}_o^j$	$X \times \Pi/2$	Verifying
$\mathsf{maul}_i/\mathsf{maul}_o$	$X \times \Pi/\mathbb{1}$	Proof malleability
$\mathsf{wit}_o/\mathsf{wit}_i$	$\mathbb{1} + W/X \times \Pi$	Adversarial witness query
$\mathsf{prf}_o/\mathsf{prf}_i$	Π/X	Proof object selection

$I = \{\mathsf{maul}_i, \mathsf{wit}_o, \mathsf{prf}_o\} \cup \{\ \mathsf{prove}_i^j, \mathsf{verify}_i^j \mid j \in \mathcal{H}\ \}$

$O = \{\mathsf{maul}_o, \mathsf{wit}_i, \mathsf{prf}_i\} \cup \{\ \mathsf{prove}_o^j, \mathsf{verify}_i^j \mid j \in \mathcal{H}\ \}$

$A = \{\mathsf{maul}_i, \mathsf{maul}_o, \mathsf{wit}_i, \mathsf{wit}_o, \mathsf{prf}_o, \mathsf{prf}_i\}$

When receiving (x, w) *on interface* prove_i^j*:*

if $(x, w) \notin \mathcal{R}$ **then**

  output $\mathsf{inj}_1(\top)$ on prove_o^j

else

  output x on prf_i

  require response π on prf_o

  assert $(x, \pi) \notin \overline{\pi}$

  let $\pi \leftarrow (x, \pi) :: \pi$

  output $\mathsf{inj}_2(\pi)$ on prove_o^j

When receiving (x, π) *on interface* verify_i^j*:*

if $(x, \pi) \notin (\pi \cup \overline{\pi})$ **then**

output (x, π) on wit_i

require response r on wit_o

if $\exists w\colon r = \mathsf{inj}_2(w) \wedge (x, w) \in \mathcal{R}$ **then**

let $\pi \leftarrow (x, \pi)::\pi$

if $(x, \pi) \in \pi$ **then**

output 1 on verify_i^j

else

let $\overline{\pi} \leftarrow (x, \pi)::\overline{\pi}$

output 0 on verify_i^j

When receiving (x, π) *on interface* maul_i*:*

if $\exists \pi'\colon (x, \pi') \in \pi \wedge (x, \pi) \notin \overline{\pi}$ **then**

let $\pi \leftarrow (x, \pi)::\pi$

In the corresponding real-world, we use a zk-SNARK scheme $\mathcal{S} = (S, T, P, \mathsf{Prove}, \mathsf{Verify}, \mathsf{SimProve}, \mathcal{X}_w)$ satisfying the standard properties of correctness, soundness, and zero-knowledge in the random oracle model with SRS. Here S, T, and P, are the structure function, trapdoor domain, and permissible permutations of the structured reference strings, which will be discussed in more detail in Chapter 4. SimProve should take as inputs only the witness x and trapdoor $t \in T$. In addition, $\mathcal{S}$ should be simulation extractable with respect to the AGM – after any arbitrary interaction, $\mathcal{X}_w$ should be able to produce the witness for any valid statement/proof pair, with the sole exception that the proof was generated with SimProve. Such white-box simulation extractability has been under-studied for zk-SNARKs, although it has been established for Groth's zk-SNARK [BKSV21] and is plausible to hold in the AGM for most SNARKs. For this reason, we rely on Groth's zk-SNARK to concretely instantiate this example, although we conjecture it applies to other SNARKs, notably Sonic. In the real-world, an adversarially biased updateable structured reference string (SRS), parameterised for the SNARK's reference string, is available, specified as:

Node SRS

The SRS node constructs a (adversarially biased) structured reference string. The rationale behind this design is formally introduced in Chapter 4 as $\mathcal{F}_{\mathsf{uSRS}}$. Assumed are the following types: a) T for trapdoors, b) S for reference strings, c) P for permis-

sible permutations over T.

State variables and initialisation values:

Variable	Description
$\mathsf{ok}^j : \mathbb{2} \coloneqq \mathsf{inj}_1(\top)$	Initialisation status ($j \in \mathbb{Z}_n$)
$t_{\mathcal{H}} : \mathbb{1} + T \coloneqq \mathsf{inj}_1(\top)$	Honest trapdoor
$t : \mathbb{1} + T \coloneqq \mathsf{inj}_1(\top)$	Full trapdoor

---

Interfaces and their types:

	Type	Description
$\mathsf{init}_i^j / \mathsf{init}_o^j$	$\mathbb{1}/\mathbb{1}$	SRS initialisation
$\mathsf{ninit}_o^j / \mathsf{ninit}_i^j$	$\mathbb{1}/\mathbb{1}$	SRS initialisation notification
$\mathsf{srs}_i^j / \mathsf{srs}_o^j$	$\mathbb{1}/S$	SRS query
$\mathsf{hsrs}_i / \mathsf{hsrs}_o$	$\mathbb{1}/S$	Honest SRS component query
$\mathsf{perm}_o / \mathsf{perm}_i$	$P/\mathbb{1}$	Permutation query

$I = \{\mathsf{hsrs}_i, \mathsf{perm}_o\} \cup \{\, \mathsf{init}_i^j, \mathsf{ninit}_o^j, \mathsf{srs}_i^j \mid j \in \mathcal{H} \,\}$

$O = \{\mathsf{hsrs}_o, \mathsf{perm}_i\} \cup \{\, \mathsf{init}_o^j, \mathsf{ninit}_i^j, \mathsf{srs}_o^j \mid j \in \mathcal{H} \,\}$

$A = \{\mathsf{hsrs}_i, \mathsf{hsrs}_o, \mathsf{perm}_i, \mathsf{perm}_o\} \cup \{\, \mathsf{ninit}_o^j, \mathsf{ninit}_i^j \mid j \in \mathcal{H} \,\}$

When receiving $\top$ *on interface* init_i^j*:*

if $\mathsf{ok}^j = \mathsf{inj}_1(\top)$ **then**
  let $\mathsf{ok}^j \leftarrow \mathsf{inj}_2(\top)$
  output $\top$ on ninit_i^j
  require response $\top$ on ninit_o^j
output $\top$ on init_o^j

When receiving $\top$ *on interface* hsrs_i*:*

if $t_{\mathcal{H}} = \mathsf{inj}_1(\top)$ **then**
  let $t_{\mathcal{H}} \xleftarrow{*} \mathsf{inj}_2(T)$
assert $\exists t' : t_{\mathcal{H}} = \mathsf{inj}_2(t')$
output $S(t')$ on hsrs_o

When receiving $\top$ *on interface* srs_i^j*:*

assert $\forall k \in \mathcal{H} : \mathsf{ok}^k = \mathsf{inj}_2(\top)$
if $t_{\mathcal{H}} = \mathsf{inj}_1(\top)$ **then**
  let $t_{\mathcal{H}} \xleftarrow{*} \mathsf{inj}_2(T)$

assert $\exists t' \colon t_{\mathcal{H}} = \mathsf{inj}_2(t')$
if $t = \mathsf{inj}_1(\top)$ **then**
  output $\top$ on perm_i
  require response p on $k_{o,\mathfrak{K}}$
  let $t \leftarrow \mathsf{inj}_2(p(t'))$
assert $\exists t' \colon t = \mathsf{inj}_2(t)$
output $S(t')$ on srs_o^j

Furthermore, for each honest party $j \in \mathcal{H}$, an instance of the SNARK protocol node (SNARK-NODE(j)) is available, which connects to the corresponding party's SRS interface and runs the SNARK Prove and Verify algorithms when queried. In both worlds, the $\mathfrak{K}_{\mathsf{bAGM}}$ public parameters are provided by the node G. SNARK-NODE(j) is specified as:

Node SNARK-NODE(j)

The SNARK protocol relies on the scheme's Prove and Verify algorithms, and access to the (adversarially biased) SRS. Each SNARK-NODE depends on the party ID j. As SNARKs often rely on a random oracle, an interface to query RO is available.

Interfaces and their types:

	Type	Description
$\mathsf{prove}_i^j/\mathsf{prove}_o^j$	$X \times W/\mathbb{1} + \Pi$	Proving
$\mathsf{verify}_i^j/\mathsf{verify}_o^j$	$X \times \Pi/\mathbb{2}$	Verifying
$\mathsf{srs}_o^j/\mathsf{srs}_i^j$	$S/\mathbb{1}$	SRS query
$\mathsf{ro}_o^j/\mathsf{ro}_i^j$	$\mathbb{2}^\kappa/\mathbb{2}^*$	Random oracle query

$I = \{ \mathsf{prove}_i^j, \mathsf{verify}_i^j, \mathsf{srs}_o^j \}$
$O = \{ \mathsf{prove}_o^j, \mathsf{verify}_i^j, \mathsf{srs}_i^j \}$
$A = \varnothing$

When receiving (x, w) *on interface* prove_i^j*:*
if $(x, w) \notin \mathcal{R}$ **then**
  output $\mathsf{inj}_1(\top)$ on prove_o^j
else
  output $\top$ on srs_i^j
  require response srs on srs_o^j
  output $\mathsf{Prove}(\mathsf{srs}, x, w)$ on prove_o^j

When receiving (x, π) *on interface* verify_i^j*:*

output $\top$ on srs_i^j

require response srs on srs_o^j

output $\mathsf{Verify}(\mathsf{srs}, x, \pi)$ on verify_o^j

Finally, the SNARK's Prove and Verify algorithms make use of a random oracle, which is available in the real world, providing query interfaces to all parties (we do not treat the random oracle as a knowledge assumption in this example). This is specified as:

Node RO

The random oracle node records queries of arbitrary bit strings and responds either with an already recorded response, or a value sampled uniformly at random from 2^κ.

State variables and initialisation values:

Variable	Description
$H\colon (2^* \times 2^\kappa)^* \coloneqq \varepsilon$	Recorded queries

Interfaces and their types:

	Type	Description
$\mathrm{ro}_i^j/\mathrm{ro}_o^j$	$2^*/2^\kappa$	Random oracle query

$I = \{\ \mathrm{ro}_i^j \mid j \in \mathbb{Z}_n\ \}$

$O = \{\ \mathrm{ro}_o^j \mid j \in \mathbb{Z}_n\ \}$

$A = \varnothing$

When receiving x *on interface* ro_i^j*:*

let $b \leftarrow 0$

for (x', h) **in** H **do**

  if $x = x'$ **then**

    let $b \leftarrow 1$

    output h on ro_o^j

if $b = 0$ **then**

  let $h \xleftarrow{*} 2^\kappa$

  let $H \leftarrow (x, h), H$

  output h on ro_o^j

The ideal-world therefore consists of $\{\text{NIZK}, \mathbb{G}\}$ (and the simulator, which will be introduced in the security analysis) and the real-world consists of $\text{SNARK} \uplus \{\text{SRS}, \text{RO}, \mathbb{G}\}$, where $\text{SNARK} \coloneqq \{\ \text{SNARK-NODE}(j) \mid j \in \mathcal{H}\ \}$. The topology of both worlds is sketched in Figure 3.4.

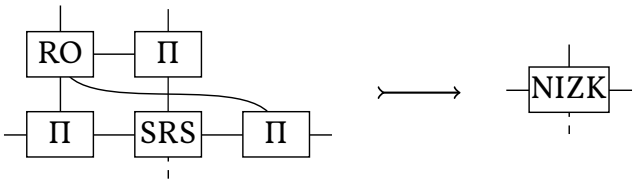


Figure 3.4: The Sonic to NIZK topologies. SNARK-NODE is represented by Π and the public parameter node $\mathbb{G}$ is omitted for clarity.

3.4.2 Security Analysis

The SNARK simulator α both faithfully simulates the SRS node, creates simulated proofs for honest proving queries and extracts witnesses using $\mathcal{X}_w$ (which is given access to $\text{REPO}(\mathfrak{K}_{\text{bAGM}})$) from adversarial proofs when requested by the NIZK node. Finally, if the simulator fails to extract a witness when asked for one for a valid proof, it requests a maul.

Node α

The SNARK to NIZK simulator α makes use of a simulated prover from [MBKM19], SimProve, as well as an assumed extractor $\mathcal{X}_w$, which makes use of the $\mathfrak{K}_{\text{bAGM}}$ knowledge extraction. Otherwise, the simulator mimics the SRS node, albeit retaining access to the full trapdoor. When SimProve and Verify require access to the random oracle, it simulates the behaviour of RO using H. This simulated behaviour is also exported on the corrupted parties ro interfaces.

State variables and initialisation values:

Variable	Description
$\text{ok}^j \colon 2 \coloneqq \text{inj}_1(\top)$	Initialisation status ($j \in \mathcal{H}$)
$t_{\mathcal{H}} \colon \mathbb{1} + T \coloneqq \text{inj}_1(\top)$	Honest trapdoor
$t \colon \mathbb{1} + T \coloneqq \text{inj}_1(\top)$	Full trapdoor
$H \colon (2^* \times 2^\kappa)^* \coloneqq \varepsilon$	Recorded queries

Interfaces and their types:

	Type	Description
$\mathsf{maul}_o/\mathsf{maul}_i$	$\mathbb{1}/X \times \Pi$	Proof malleability
$\mathsf{wit}_i/\mathsf{wit}_o$	$X \times \Pi/\mathbb{1} + W$	Adversarial witness query
$\mathsf{prf}_i/\mathsf{prf}_o$	X/Π	Proof object selection
$\mathsf{init}_i^j/\mathsf{init}_o^j$	$\mathbb{1}/S$	SRS initialisation
$\mathsf{ninit}_o^j/\mathsf{ninit}_i^j$	$\mathbb{1}/\mathbb{1}$	SRS initialisation notification
$\mathsf{hsrs}_i/\mathsf{hsrs}_o$	$\mathbb{1}/S$	Honest SRS component query
$\mathsf{perm}_o/\mathsf{perm}_i$	$P/\mathbb{1}$	Permutation query
$w_{\mathfrak{K}_{\mathsf{bAGM}}}/x_{\mathfrak{K}_{\mathsf{bAGM}}}$	$W_{\mathsf{pp}}^*/X_{\mathsf{pp}}$	Knowledge extraction
$\mathsf{ro}_i^j/\mathsf{ro}_o^j$	$2^*/2^\kappa$	Random oracle query

$I = \{\mathsf{maul}_o, \mathsf{wit}_i, \mathsf{prf}_i, \mathsf{hsrs}_i, \mathsf{perm}_o, w_{\mathfrak{K}_{\mathsf{bAGM}}}\} \cup \{\, \mathsf{init}_i^j, \mathsf{ninit}_o^j \mid j \in \mathcal{H} \,\} \cup \{\, \mathsf{ro}_i^j \mid j \in \mathcal{A} \,\}$
$O = \{\mathsf{maul}_i, \mathsf{wit}_o, \mathsf{prf}_o, \mathsf{hsrs}_o, \mathsf{perm}_i, x_{\mathfrak{K}_{\mathsf{bAGM}}}\} \cup \{\, \mathsf{init}_o^j, \mathsf{ninit}_i^j \mid j \in \mathcal{H} \,\} \cup \{\, \mathsf{ro}_o^j \mid j \in \mathcal{A} \,\}$
$A = \{\mathsf{maul}_o, \mathsf{maul}_i, \mathsf{wit}_i, \mathsf{wit}_o, \mathsf{prf}_i, \mathsf{prf}_o, \mathsf{hsrs}_i, \mathsf{hsrs}_o, \mathsf{perm}_o, \mathsf{perm}_i, w_{\mathfrak{K}_{\mathsf{bAGM}}}, x_{\mathfrak{K}_{\mathsf{bAGM}}}\} \cup$
$\{\, \mathsf{ninit}_o^j, \mathsf{nini}_i^j \mid j \in \mathcal{H} \,\}$

When receiving (x, π) on interface wit_i:

let $t' \leftarrow$ ensureSrs
if $\mathsf{Verify}(S(t'), x, \pi)$ **then**
  let $w \leftarrow \mathcal{X}_w(x, \pi)$
  if $(x, w) \in \mathcal{R}_{\mathsf{pp}}$ **then**
    output $\mathsf{inj}_2(w)$ on wit_o
  else
    output (x, π) on maul_i
    require response $\top$ on maul_o
    output $\mathsf{inj}_1(\top)$ on wit_o
else
  output $\mathsf{inj}_1(\top)$ on wit_o

When receiving x on interface prf_i:

let $t' \leftarrow$ ensureSrs
assert $\exists t' : t = \mathsf{inj}_2(t')$
return $\mathsf{SimProve}(S(t'), t', x)$

When receiving $\top$ on interface init_i^j:

let $\mathsf{ok}^j \leftarrow \mathsf{inj}_2(\top)$
output $\top$ on ninit_i^j

require response $\top$ on ninit_o^j

output $\top$ on init_o^j

When receiving $\top$ *on interface* hsrs_i*:*

if $t_{\mathcal{H}} = \mathsf{inj}_1(\top)$ **then**

let $t_{\mathcal{H}} \xleftarrow{*} \mathsf{inj}_2(T)$

assert $\exists t' \colon t_{\mathcal{H}} = \mathsf{inj}_2(t')$

output t' on hsrs_o

When receiving x *on interface* ro_i^j*:*

let $b \leftarrow 0$

for (x', h) **in** H **do**

if $x = x'$ **then**

let $b \leftarrow 1$

output h on ro_o^j

if $b = 0$ **then**

let $h \xleftarrow{*} 2^\kappa$

let $H \leftarrow (x, h), H$

output h on ro_o^j

Helper procedures:

procedure ensureSrs

assert $\forall k \in \mathcal{H} \colon \mathsf{ok}^k = \mathsf{inj}_2(\top)$

if $t_{\mathcal{H}} = \mathsf{inj}_1(\top)$ **then**

let $t_{\mathcal{H}} \xleftarrow{*} \mathsf{inj}_2(T)$

assert $\exists t' \colon t_{\mathcal{H}} = \mathsf{inj}_2(t')$

if $t = \mathsf{inj}_1(\top)$ **then**

output $\top$ on perm_i

require response p on $k_{o,\mathfrak{K}}$

let $t \leftarrow \mathsf{inj}_2(p(t'))$

assert $\exists t' \colon t = \mathsf{inj}_2(t)$

return t'

Theorem 3.4 (SNARKs Constructs NIZKs). *For any secure SNARK scheme* $\mathcal{S}$*:*

$$\mathfrak{K}(\textsc{snark}) \uplus \{\textsc{srs}, \mathfrak{K}(\textsc{ro}), \mathbb{G}\} \xrightarrow{\alpha, \mathfrak{D}_{\mathfrak{K}}} \{\textsc{nizk}, \mathbb{G}\}. \tag{3.17}$$

Proof (sketch). All honestly generated proofs will verify in both worlds, by definition in the ideal world and by the correctness of the SNARK in the real

world. Furthermore, the proofs themselves are indistinguishable, by the zero-knowledge property of the SNARK.

Adversarial proofs which fail to verify will also be rejected in the ideal world, as the simulator will refuse to provide a witness, causing them to be rejected. As per the above, the extractor $\mathcal{X}_w$ is able to (using $\text{REPO}(\mathfrak{K}_{\text{bAGM}})$) extract the witnesses for any adversarial proof which *does* verify, except for cases of malleability. As $\mathcal{S}$ is only (at most) proof-malleable, the simulator can, and does, account for this by attempting to create a mauled proof when extraction fails.

The simulator provides the ideal-world simulation of the SRS node, which is emulated faithfully except that the simulator has access to the trapdoor. As a result, this part of the system cannot be used to distinguish. □

3.4.3 The Impossibility of General Composition

The two parts of Theorem 3.3 are limited when compared to Corollary 3.2 in two separate, but related ways: The closure under subgraph substitution requires the added node to be a $\vec{\mathfrak{K}}$-wrapped node, and transitivity requires the two composing proofs to use separate knowledge assumptions.

We will demonstrate that the nicer results from Corollary 3.2 are not achievable with respect to knowledge-respecting distinguishers, by means of a small counter-example for both situations.

Theorem 3.5 (Subgraph Substitution is Limited). *Subgraph substitution with knowledge assumptions does not universally preserve secure construction.* $\exists A, B, C, \alpha \in *, \varepsilon \in \mathbb{R}, \vec{\mathfrak{K}}$*:*

$$A \xrightarrow{\varepsilon,\alpha,\mathfrak{D}_{\vec{\mathfrak{K}}}} B \implies CA \xrightarrow{\varepsilon,\alpha,\mathfrak{D}_{\vec{\mathfrak{K}}}} CB$$

Proof (sketch). Let A be the SNARK real world and B be the NIZK ideal world respectively, with α being their simulator and $\vec{\mathfrak{K}}$ being $\{\mathfrak{K}_{\text{bAGM}}\}$. Let C be a node which receives elements in X_{pp}, queries $\text{REPO}(\mathfrak{K}_{\text{bAGM}})$ and returns the witness to the distinguisher.

Then the following distinguisher can trivially distinguish the two worlds: a) Make any honest proving query. b) Request extraction. c) Output whether or not extraction succeeded. □

Theorem 3.6 (Transitivity is Limited). *Construction with knowledge assumptions*

is not universally transitive. $\exists A, B, C, \alpha, \beta \in *, \varepsilon_1, \varepsilon_2 \in \mathbb{R}, \mathfrak{D}_{\vec{\mathfrak{K}}}$:

$$A \xrightarrow{\varepsilon_1,\alpha,\mathfrak{D}_{\vec{\mathfrak{K}}}} B \wedge B \xrightarrow{\varepsilon_2,\beta,\mathfrak{D}_{\vec{\mathfrak{K}}}} C \not\Longrightarrow A \xrightarrow{\varepsilon_1+\varepsilon_2,\alpha\beta,\mathfrak{D}_{\vec{\mathfrak{K}}}} C$$

Proof (sketch). Let B be the SNARK real world and C be the NIZK ideal world, respectively, with β being their simulator and $\vec{\mathfrak{K}}$ being $\{\mathfrak{K}_{\text{bAGM}}\}$. Let A be the SNARK with additional interfaces for each party to reveal any witnesses of broadcast proofs, which are shared through an additional broadcast channel. Let α reproduce this functionality by extracting witnesses from the provided proofs.

Then a distinguisher which makes an honest proof and extracts it will receive the witness in the real and hybrid world, but not in the ideal world, where the knowledge extraction of the proof will fail, as it is simulated by β. It is therefore possible to distinguish and transitivity does not hold. □

3.5 Relation to Simple UC

As the remainder of this thesis is stated using the UC model (see Subsection 2.3.1), it is important to discuss how the result of composable SNARKs from this chapter can be used together with the rest of the results in this thesis. While the relation with UC is less clear than that with Constructive Cryptography, it is still the case that a proof of construction $A \xrightarrow{\varepsilon,\alpha,*} B$ is directly equivalent to a statement of UC-emulation and vice versa. This is due to both settings having the same token-passing execution mechanism, and the computational abstractions of random systems and Turing machines are equivalent. There are two main inconsistencies: UC has both an environment and adversary, and only two "interfaces" exist for each ITI: the adversarial "backdoor" tape and the honest input tape, with the difference that any ITI can write to the input tape.

The adversary/environment difference is resolved by UC itself [Can01], which notes that a dummy adversary can be used without loss of generality. Using this, the environment and distinguisher are in effect the same entity. Interfaces are also less of an issue than they may appear. If multiple interfaces exist between two nodes, this is equivalent to a single interface with appropriate multiplexing. As UC specifies external writes to provide information about who a message is from, this multiplexing is already build-in further – it can be separated out into

a pair of interfaces for each ITI sending a message to another, and message types can be multiplexed if desired.

In effect, it is possible to naturally transform a UC functionality or protocol into a cryptographic network and vice versa, for which the UC-emulation statement is equivalent to a statement of construction (with negligible ε and the distinguisher set $*$). This allows results from subsequent chapters to be "imported" into this framework and, as they are with respect to $*$ and not using any knowledge assumptions, the SNARK construction from Section 3.4 can be directly composed with them according to Theorem 3.3.

4 Secure Reference Strings from Consensus

This chapter is based on "Mining for Privacy: How to Bootstrap a Snarky Blockchain" [KKK21c], first published at the 2021 Conference on Financial Cryptography and Data Security, primarily authored by Thomas Kerber, and co-authored by Aggelos Kiayias and Markulf Kohlweiss.

Polynomial commitments used in zk-SNARKs, commonly build on a structured reference string, or SRS, consisting of different powers of a group generator, for instance g^{x^i} for some $x \in \mathbb{F}_p$ and for all $i \in \mathbb{Z}_n$. While impossibility results [GO94] for non-interactive zero-knowledge show that *some* common reference string (or other setup, such as random oracles) are required, these strings are worse, as not only do they need to be randomly distributed, but their underlying *trapdoor* value (in the toy example of g^{x^i}, this would be x) must remain secret.

The obvious way of sampling such a reference string from public randomness reveals the exponents used – and knowledge of these values breaks the soundness of the proof system itself. To make matters worse, the security of SNARKs typically relies on knowledge assumptions, which state that to create group elements related in such a way *requires* knowing the underlying exponents and hence any SRS sampler will have to know the exponents used and be trusted to erase them, becoming effectively a single point of failure for the underlying system. While secure multi-party computation can be, and has been, used to reduce the trust placed on such a setup process [Zca18], the selection of the participants for the secure computation and the verification of the generation of the SRS by the MPC protocol retain an element of centralisation.

For *updateable* reference strings, such as [GKM+18, MBKM19], it is possible to produce an updated reference string from a prior one, such that knowing the trapdoor of the new string requires both knowing the trapdoor of the old string, *and* knowing the randomness used in the update. Groth et al. [GKM+18] conjectured that a blockchain protocol may be used to securely generate such a refer-

ence string. This conjecture is confirmed by the results of this chapter, which presents a protocol to do so, primarily relying on the *chain quality* property of "Nakamoto-style" ledgers [GKL15]. The reference string updates are integrated into the block creation process, with special care being taken to accommodate the high computation and communication cost of processing these updates. A further important consideration is that an adversary may gain a significant advantage by "cheating" on the reference string update process, by choosing low-entropy updates which are easy to compute. This chapter considers both the adversarial and rational impact of this and demonstrates how the latter can be controlled.

Related Work. Beyond the obvious relation to the works introducing updateable reference strings in [GKM+18, MBKM19] (most notably Sonic [MBKM19], which we follow closely in our instantiation), there have been attempts of practically answering the question of how to securely generate reference strings. These have been in a setting where the string is *not* updateable.

Notably Bowe et al. [BGG19] describe the mechanism used by Sprout, the first version of Zcash, during the initial setup of the cryptocurrency's SRS. It uses multi-party computation to generate a reference string with a root of trust on the initial group of people participating. Due to performance constraints on the MPC protocol, the set of parties participating is relatively small, although only the honesty of a single participating party is required.

For the Sapling version of Zcash, a different approach was used when their reference string was replaced (due to an upgrade of the zero-knowledge statement and proof system used). Their second CRS generation mechanism, described in [BGM17] uses a multiple-phase round-robin mechanism to generate a reference string for Groth's zk-SNARK [Gro16]. They utilise a random beacon to ensure the uniform distribution of the result and a coordinator to perform deterministic auxiliary computations.

Finally, Abdolmaleki et al. [ABL+19] demonstrate the UC security of an MPC-based approach to SRS generation.

4.1 Updateable Structured Reference Strings

While updateable structured reference strings (uSRSs) are modelled in the works we are building on [MBKM19, Section 3.2], we model their security in the setting of universal composability (UC) [Can01]. Here, a uSRS is a reference string with an underlying trapdoor τ, which has had a structure function S imposed on it. $S(\tau)$ is the reference string itself, while τ is not revealed to the adversary. In Subsection 4.1.3, we prove that Sonic [MBKM19] (with small modifications for extraction, as described in Subsection 4.1.2), satisfies all the properties we require in this section. Our main proof is independent of the Sonic protocol however, and applies to any updateable reference string scheme satisfying the properties laid out in the rest of this section.

4.1.1 Standard Requirements

A uSRS scheme $\mathcal{S}$ consists of a trapdoor domain T, an initial trapdoor τ_0, a set P of permissible (and invertible) permutations over T (that is, bijective functions whose domain and codomain is T), and a structure function S with the domain T. We require P to include the identity function id and to be closed under function composition: $\forall p_1, p_2 \in P\colon p_1 \circ p_2 \in P$. An efficient permutation lifting $\dagger$ must exist (and we will demonstrate its existence for Sonic), such that for any permutation $p \in P$ and $\tau \in T$, $p^\dagger(S(\tau)) = S(p(\tau))$. Finally, there must exist algorithms $\rho \leftarrow \mathsf{ProveUpd}(S(\tau), p)$ and $b \leftarrow \mathsf{VerifyUpd}(S(\tau), \rho, S(p(\tau)))$ for creating and verifying update proofs respectively. The format of these update proofs is not specified, however the following constraints must be met:

1. **Correctness.** Applying an honestly generated update proof will verify:
   $\forall p \in P, \tau \in T\colon \mathsf{VerifyUpd}(S(\tau), \mathsf{ProveUpd}(S(\tau), p), S(p(\tau)))$.

2. **Structure preservation.** Applying *any* valid update is equivalent to applying *some* permutation $p \in P$ on the trapdoor:
   $\forall \rho, \tau, \mathsf{srs}'\colon \mathsf{VerifyUpd}(S(\tau), \rho, \mathsf{srs}') \implies \exists p \in P\colon \mathsf{srs}' = S(p(\tau))$.

3. **Update uniformity.** Applying a random permutation is equivalent to selecting a new random trapdoor: Let D be the uniform distribution over T and, for all $\tau \in T$, let D_τ be the uniform distribution over the multiset $\{\, p(\tau) \mid p \in P \,\}$. Then $\forall \tau \in T\colon D = D_\tau$.

We define a corresponding UC functionality $\mathcal{F}_{\mathsf{uSRS}}$, which provides a reference string $S(p(\tau_{\mathcal{H}}))$, which the adversary can influence by providing the permutation $p \in P$, given only $S(\tau_{\mathcal{H}})$ as input, for a randomly sampled $\tau_{\mathcal{H}} \in T$.

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

The updateable structured reference string functionality $\mathcal{F}_{\mathsf{uSRS}}$ allows the adversary to update a reference string by applying a permutation from a set of permissible permutations P.

The functionality is parameterised by a trapdoor domain T, a structure function S, and a set of permissible permutations P over T.

State variables and initialisation values:

Variable	Description
$\tau_{\mathcal{H}} \coloneqq \bot$	The honest part of the trapdoor
$\tau \coloneqq \bot$	The trapdoor

When receiving a message HONEST-SRS *from $\mathcal{A}$:*

if $\tau_{\mathcal{H}} = \bot$ **then let** $\tau_{\mathcal{H}} \xleftarrow{*} T$

return $S(\tau_{\mathcal{H}})$

When receiving a message SRS *from a party ψ:*

query $\mathcal{A}$ **with** (PERMUTE, ψ) and **receive the reply** p

if $\tau = \bot$ **then**

  assert $p \in P \wedge \tau_{\mathcal{H}} \neq \bot$

  let $\tau \leftarrow p(\tau_{\mathcal{H}})$

return $S(\tau)$

We believe this functionality to be of independent interest and it is not explicitly tied to our implementation. Notably, while we use a distributed ledger as a weak form of a broadcast channel, other broadcasts can be considered without modification to this functionality. While, as presented, the functionality does not dictate any specific usage, we conjecture that when parameterised with an appropriate structure function and permutation set it can be used to securely instantiate updateable SRS-based SNARKs, such as Sonic [MBKM19], Marlin [CHM+20], or Plonk [GWC19]. Due to the UC setting, this would require additional lifting to enable UC knowledge extraction, such as that of C∅C∅ [KZM+15].

4.1.2 Simulation Requirements

In addition to the basic properties of correctness, structure preservation, and update uniformity, any simulator wishing to help realise $\mathcal{F}_{\mathsf{uSRS}}$ via updates will need to have access to two additional properties:

1. **Update proof simulation.** From an initial SRS $S(\tau)$ for which the simulator knows the trapdoor, it can produce a valid update to any (correctly structured) SRS. Formally, with $\mathcal{S}_\rho$ a PPT algorithm:
   $\exists \mathcal{S}_\rho \forall \tau_1, \tau_2 \in T\colon \mathsf{VerifyUpd}(S(\tau_1), \mathcal{S}_\rho(\tau_1, S(\tau_2)), S(\tau_2))$
2. **Permutation extraction.** The simulator must be capable of extracting the permutation p underlying any valid adversarial update proof.

The most natural method to achieve permutation extraction would be using white-box extractors, as the updates themselves typically rely on some form of knowledge assumption, such as knowledge-of-exponent. However, white-box extractors cannot be used in UC proofs. Instead, we will assume that the update proof is proven to correspond to a specific trapdoor through a lower-level NIZK. Crucially, this lower-level NIZK should not require a *structured* reference string and can rely only on a common random string, or a random oracle. Fortunately, it is not subject to stringent efficiency requirements as Section 4.4 demonstrates.

Specifically, we assume that the basic update proof ρ is a statement in a NIZK relation $\mathcal{R}$ where the witness is an encoding of the corresponding permutation p. We require each update proof to have one and only one corresponding permutation, formally expressed by requiring $\mathcal{R}$ to be a bijection. This results in a straightforward modification to the $\mathsf{ProveUpd}$ and $\mathsf{VerifyUpd}$ algorithms that permits the extraction of the underlying permutations even in the UC setting: $\mathsf{ProveUpd}$ also creates a NIZK proof π of (ρ, p) and returns (ρ, π), While $\mathsf{VerifyUpd}$ returns true only if this newly embedded NIZK proof also verifies.

The addition of this NIZK trivially preserves all security properties including correctness, due to the definition of $\mathcal{R}$:

Definition 4.1. A uSRS scheme is permutation extractable if the relation

$$\mathcal{R} \coloneqq \{ (\mathsf{ProveUpd}(S(\tau), p), p) \mid \tau \in T, p \in P \}$$

is a bijection and in NP.

4.1.3 The Sonic uSRS

Sonic's uSRS [MBKM19, Section 4.3] consists of a series of exponentiations of group elements in pairing groups $\mathbb{G}_1$ and $\mathbb{G}_2$ of prime order q, where a bilinear pairing $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ exists. Specifically, given generators $g \in \mathbb{G}_1, h \in \mathbb{G}_2$ and a depth parameter $d \in \mathbb{Z}_q$, the SRS has a trapdoor of $(\alpha, x) \in \mathbb{F}_q^{*2}$, with $\tau_0 = (1, 1)$.

The corresponding structure function is defined as:

$$S((\alpha, x)) \coloneqq \left(\left\{ g^{x^i}, h^{x^i}, h^{\alpha x^i} \right\}_{i=-d}^{d}, \left\{ g^{\alpha x^i} \right\}_{i=-d, i\neq 0}^{d} \right)$$

Specification of Sonic Updates. We omit the $e(g, h^{\alpha})$ term presented in Sonic, as this can be computed from the rest of the SRS and is therefore immaterial to the update procedure. The permitted trapdoor permutations are field multiplications:

$$P \coloneqq \left\{ (\alpha, x) \mapsto (\alpha\beta, xy) \mid (\beta, y) \in \mathbb{F}_q^{*2} \right\}.$$

Correspondingly, † exponentiates group elements:

$$\begin{aligned} p = (\alpha, x) \mapsto (\alpha\beta, xy) \implies \\ p^{\dagger} = \left(\left\{ G_i, H_i, H_i' \right\}_{i=-d}^{d}, \left\{ G_i' \right\}_{i=-d, i\neq 0}^{d} \right) \\ \mapsto \left(\left\{ G_i^{y^i}, H_i^{y^i}, H_i'^{\beta y^i} \right\}_{i=-d}^{d}, \left\{ G_i'^{\beta y^i} \right\}_{i=-d, i\neq 0}^{d} \right) \end{aligned}$$

Observe that field *multiplications* over α or x can efficiently be applied to the corresponding structure through exponentiation: $g^{(\alpha x^i)\beta y^i} = \left(g^{\alpha x^i} \right)^{\beta y^i}$. The full update proof procedure is as follows:

procedure ProveUpd(srs, p)
  let $(\beta, y) \leftarrow p((1, 1))$
  return $(g^y, g^{\beta y}, \pi)$

The verification procedure ensures correct computation by checking the consistency of various pairing computations:

procedure VerifyUpd(srs, ρ, srs′)
  let $(\{G_i, H_i, H_i'\}_{i=-d}^{d}, \{G_i'\}_{i=-d, i\neq 0}^{d}) \leftarrow$ srs
  let $(\{I_i, J_i, J_i'\}_{i=-d}^{d}, \{I_i'\}_{i=-d, i\neq 0}) \leftarrow$ srs′
  let $(A, B, \pi) \leftarrow \rho$
  if $e(I_1', h) \neq e(B, H_1') \vee e(g, J_1') \neq e(B, H_1') \vee e(I_1, h) \neq e(A, H_1) \vee e(g, J_1) \neq e(A, H_1) \vee$
    $I_0 \neq g \vee J_0 \neq h$ **then**

return 0

for $i = -d$ **to** d **do**

if $\neg(i = d \vee e(I_i, J_1) = e(I_1, J_i) = e(I_{i+1}, h) = e(g, J_{i+1})) \vee \neg(e(I_i, J_0') = e(g, J_i')) \vee$
$(i \neq 0 \wedge \neg e(I_i, J_0') = e(I_i', h))$ **then**

return 0

return 1

Satisfaction of Security Properties.

Theorem 4.1. *Sonic, as described in this section, is an updatable reference string scheme, satisfying correctness, structure preservation, update uniformity, update extraction, permutation extraction, and permutation lifting.*

Proof. We prove each property individually.

Correctness. Follows from all pairing checks being satisfied. □

Structure preservation. Suppose a structured input $S(\tau)$, an update proof ρ, and a new SRS srs′, where:

$$S(\tau) = \left(\left\{g^{x^i}, h^{x^i}, h^{\alpha x^i}\right\}_{i=-d}^{d}, \left\{g^{\alpha x^i}\right\}_{i=-d, i\neq 0}^{d}\right)$$

$$\mathsf{srs}' = \left(\left\{g^{k_i}, h^{m_i}, h^{n_i}\right\}_{i=-d}^{d}, \left\{g^{l_i}\right\}_{i=-d, i\neq 0}^{d}\right)$$

$$\rho = (g^y, g^{\beta y})$$

If VerifySRS returns 1, we know all of the following hold, due to the conditions checked:

- $e(g^{l_1}, h) = e(g, h^{n_1}) = e(g^{\beta y}, h^{\alpha x})$
- $e(g^{k_1}, h) = e(g, h^{m_1}) = e(g^y, h^x)$
- $\forall i \in [-d, d)$: $e(g^{k_i}, h^{m_1}) = e(g^{k_1}, h^{m_i}) = e(g^{k_{i+1}}, h) = e(g, h^{m_{i+1}})$
- $\forall i \in [-d, d]$: $e(g^{k_i}, h^{n_0}) = e(g, h^{n_i})$
- $\forall i \in [-d, d] \setminus \{0\}$: $e(g^{k_i}, h^{n_0}) = e(g^{l_i}, h)$

As $e(g, h)$ is a generator over $\mathbb{G}_T$ and each of the above can be expressed as an equality of exponentiations of the form $e(g, h)^a = e(g, h)^b$, we simplify these to equalities within $\mathbb{F}_q^*$ of their exponents:

- $l_1 = n_1 = \alpha\beta xy$
- $k_1 = m_1 = xy$
- $\forall i \in [-d, d)\colon k_i m_1 = k_1 m_i = k_{i+1} = m_{i+1}$
- $\forall i \in [-d, d]\colon k_i n_0 = n_i$
- $\forall i \in [-d, d] \setminus \{0\}\colon k_i n_0 = l_i$

It follows directly that $n_0 = \alpha\beta$, $k_i = m_i = (xy)^i$, and $l_i = n_i = \alpha\beta(xy)^i$. As a result, srs′ matches exactly the structured reference string $S((\alpha\beta, xy)) = p^\dagger(S(\tau))$. □

Update uniformity. Let $\tau = (\alpha, x)$. $p \xleftarrow{*} P$ is defined by a multiplication with two uniformly sampled field elements in $\beta, y \xleftarrow{*} \mathbb{F}_q^*$, such that the trapdoor $p(\tau) = (\alpha\beta, xy)$. Due to multiplication in prime fields with a fixed element (here α and x) being a bijective functions, the result $(\alpha\beta, xy)$ is also distributed uniformly at random in $\mathbb{F}_q^{*2}$, therefore being indistinguishable from a new, randomly sampled trapdoor. □

Update proof simulation. We present the following simulation algorithm:

procedure $\mathcal{S}_\rho((\alpha, x), \text{srs})$
  $(\{G_i, H_i, H_i'\}_{i=-d}^{d}, \{G_i'\}_{i=-d, i\neq 0}^{d}) \leftarrow \text{srs}$
  return $\left(G_1^{(x^{-1})}, G_1'^{(x^{-1})(\alpha^{-1})}\right)$

This utilises only a small number of efficient group operations and is therefore PPT. As the VerifyUpd pairing checks all succeed, the returned update proof will verify. □

Permutation Extraction. Observe that

$$\mathcal{R}((A, B), p) \iff \text{let } (a, b) = p((1, 1)) \text{ in } A = g^a \wedge B = g^b.$$

A straightforward encoding of p is the pair of field elements (a, b). This relation is clearly in NP, and is also a bijection due to the relation of $\mathbb{G}_1$ and $\mathbb{F}_q^*$. □

Instantiating $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$. We can employ Fischlin's transform [Fis05] in combination with a simple sigma protocol to prove knowledge of pairs of exponents. Specifically, we propose the parallel composition of two Schnorr proofs of

knowledge of exponent [Sch90]. It is important to treat these as a single proof and not two separate proofs, as the latter would enable the adversary to create proofs which are only partially extractable. We posit that these would still allow for simulation, however the simulator would be tasked with a more difficult and implementation specific book-keeping.

4.2 Building uSRS from Chain Quality

This section shows how to securely initialise a uSRS using a distributed ledger by requiring block creators to perform updates on an evolving uSRS during an initial setup period. After waiting for agreement on the final uSRS, it can be safely used. To formally model this approach, we discuss the ideal and real (or more accurately, hybrid) worlds used in our simulation proof. Both worlds have access to a ledger, however the ideal world's ledger is independent of the reference string (which is instead provided by the independent $\mathcal{F}_{\text{uSRS}}$ functionality), while the real world's ledger is programmed to generate it using updates.

4.2.1 High-Level Overview

This basic premise of this chapter relies on Nakamoto-style ledgers' basic means of operation: Different users can extend a chain of blocks if they can satisfy some condition, with this condition being associated with a type of hardness which ensures attackers are limited in the number of extensions they can perform. Given such a structure, we associate a uSRS update with each block prior to a time δ_1. This time is selected such that the security properties of the ledger ensure at least one of the blocks is honest in each competitive chain at this point.

In our modelling, we construct this from a ledger functionality with an additional *leadership state*, which is derived from information miners embed in their blocks. Specifically for our case, these encode uSRS updates. We leave this sufficiently general to allow other uses as well. The basic idea is to show that a ledger which performs uSRS updates in its leadership state is equivalent to one which does not, but is accompanied by the $\mathcal{F}_{\text{uSRS}}$ functionality. They make up our real and ideal worlds, respectively. After δ_1, users wait a further period δ_2 until the common prefix ensures that all parties agree on the reference string.

While ledger functionalities are often treated as global, our approach effec-

tively constructs one ledger from another – the ledger is not a dependency of our protocol, but a component. In this context, globality is irrelevant, as the environment already has direct access to the functionality. We expect protocols building on the ledger to use it in a global fashion, however. The same is not true for the uSRS – most usages will likely rely on the simulator being able to extract its trapdoor.

4.2.2 Our Ledger Abstraction

Our construction of the updateable structured reference string functionality relies heavily on the properties of *common prefix*, *chain quality*, and *chain growth* defined in the "Bitcoin backbone" analysis by Garay et al. [GKL15] for Nakamoto-style consensus algorithms. Despite our use in the section title, we make use of all three properties, not just that of chain quality. We emphasise chain quality, as it is the property central to ensuring an honest update has occurred. We briefly and informally restate the three properties:

- **Common prefix.** Given the current chains Π_1 and Π_2 of two parties, and removing k blocks from the first, it is a prefix of the second: $\Pi_1^{\lceil k} \prec \Pi_2$.
- **Chain quality.** For any party's current chain Π, any consecutive l blocks in this chain will include μ blocks created by an honest party.
- **Chain growth.** If a party's chain is of length c, then s time slots later, it will be at least of length $c + \gamma$.

These parameters determine the length of the two phases of our protocol. In the first phase, we construct the reference string itself from the liveness parameter (assuming $\mu \geq 1$), and, in the second phase, we wait until this reference string has propagated to all users. The length of the first phase is at least $\delta_1 \geq \lceil l\gamma^{-1} \rceil s$ and that of the second at least $\delta_2 \geq \lceil k\gamma^{-1} \rceil s$. Combined, they make up the total uSRS generation delay $\delta \geq (\lceil l\gamma^{-1} \rceil + \lceil k\gamma^{-1} \rceil)s$.

We assume a ledger which guarantees the backbone properties, formally described in Subsection 2.4.3.3. Our functionality further depends on the *global clock* $\mathcal{G}_{\text{clock}}$, as described in Subsection 2.3.5. For the purposes of this section, it is sufficient that this is a beacon providing monotonically increasing values representing the current time to any party requesting them.

In addition to this, we assume that each block created can contain additional information, provided by its creator (the "miner"), which can be aggregated to construct a "leader state". Each created block is associated with an *update* a, and the ledger is parameterised by two procedures, Gen and Apply, which describe the honest selection of updates and the semantics of updates, respectively. Looking forward, these utilise ProveUpd and VerifyUpd internally, although the formalism is sufficiently general to allow usage of the leader state for other, parallel purposes. The exact parameters differ in our ideal and real world, with the ideal world "hiding" the uSRS updates. Additionally, the real world adds time-sensitivity: It does nothing to the SRS after the setup period. Gen is randomised, takes a leader state σ and the current time t as inputs, and produces an update a. Apply takes a leader state σ, an update a, and an update time t, and returns a successor state σ': $\sigma' = \mathsf{Apply}(\sigma, (a, t))$. For a chain, the leader state may be computed by sequentially applying all updates in the chain, starting from an initial state $\varnothing$.

The adversary controls when and which party creates a new block, as well as the transactions each new block contains (provided it does not violate the backbone properties). For transactions created by a corrupted party, the adversary can further control the block's timestamp (within the reasonable limits of not being in the future and being after the previous block) and the desired update a itself. For honest parties updates, Gen is used instead.

The UC interfaces our ledger provides are:

- SUBMIT. Submitting new transactions for the ledger.
- READ. Reading the confirmed sequence of transactions.
- PROJECTION. Reading the current chain's sequence of (potentially unconfirmed) transactions.
- LEADER-STATE. Reading the confirmed leader state.
- ADVANCE. The adversary switches a party to a longer chain.
- EXTEND. The adversary instructs a party to create a block.

4.2.3 The Ideal World

Our ideal world consists of two functionalities, composed in parallel (by which we mean: the environment may address either and they do not interact). The first is a variant of $\mathcal{F}_{\text{uSRS}}$, with the modification that it cannot be addressed by honest parties before δ time slots have passed. Formally, this modification is made with a wrapper functionality $\mathcal{W}_{\text{Delay}}(\mathcal{F}, \delta)$, formally described as follows:

Functionality $\mathcal{W}_{\text{Delay}}(\delta, \mathcal{F})$

The wrapper functionality $\mathcal{W}_{\text{Delay}}(\delta, \mathcal{F})$ of $\mathcal{F}$ accepts *honest* inputs only after δ time slots.

When receiving a message M from a party ψ:

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t
if $t < \delta \wedge \psi \in \mathcal{H}$ **then return** $\perp$
else
  send M **to** $\mathcal{F}$ and **receive the reply** y
  return y

The second is the Nakamoto-style ledger functionality, parameterised with arbitrary leader-state generation and application procedures which are also partially used in the hybrid world: Gen = GenIdeal and Apply = ApplyIdeal, and the following ledger parameters:

1. A common prefix parameter k.
2. Chain quality parameters μ and l.
3. Chain growth parameters γ and s.

Formally then, our ideal world consists of the pair $(\mathcal{W}_{\text{Delay}}(\delta, \mathcal{F}_{\text{uSRS}}), \mathcal{F}^{\text{ideal}}_{\text{NakLedger}})$, as well as the global functionality $\mathcal{G}_{\text{clock}}$.

4.2.4 The Hybrid World

In our hybrid world, we use a uSRS scheme $\mathcal{S}$, with algorithms ProveUpd, VerifyUpd, the structure function S, permissible permutations P, permutation lifting $\dagger$, and initial trapdoor τ_0. The hybrid world consists of a separate Nakamoto-style ledger $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$, a NIZK functionality $\mathcal{F}^{\mathcal{R}}_{\text{NIZK}}$, and the global

clock $\mathcal{G}_{\text{clock}}$. The ledger is then parameterised by the same chain parameters as those in the ideal world and the following leader-state procedures:

procedure Apply$((\text{srs}, \sigma^{\text{ideal}}), ((\text{srs}', \rho, \pi, a^{\text{ideal}}), t))$
  if srs $= \varnothing$ **then let** srs $\leftarrow S(\tau_0)$
  if $t \leq \delta_1 \wedge$ VerifyUpd$(\text{srs}, \rho, \text{srs}')$ **then**
    send $(\textsc{verify}, \rho, \pi)$ **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and **receive the reply** b
    if b **then**
      let srs $\leftarrow$ srs$'$
  return (srs, ApplyIdeal$(\sigma^{\text{ideal}}, a^{\text{ideal}}, t)$)

procedure Gen$((\text{srs}, \sigma^{\text{ideal}}), t)$
  if $t > \delta_1$ **then**
    return $(\varepsilon, \varepsilon, \varepsilon,$ GenIdeal$(\sigma^{\text{ideal}}, t))$
  else
    let $p \xleftarrow{*} P; \rho \leftarrow$ ProveUpd(srs, p)
    send $(\textsc{prove}, \rho, p)$ **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and **receive the reply** π
    return $(p^{\dagger}(\text{srs}), \rho, \pi,$ GenIdeal$(\sigma^{\text{ideal}}, t))$

Note that these parametrising algorithms use $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and are therefore the reason the ledger depends on this hybrid functionality.

Key here is that once a block is received after the initial chain quality period, any reference string update it may declare is no longer carried out – at this point the uSRS is not necessarily stable, as the chain may still be reorganised, but should not change for this particular chain. Furthermore, these procedures always mimic the ideal-world behaviour, extending it rather than replacing it. This demonstrates the composability of allowing block leaders to produce updates: One system using updates for security does not impact other parallel uses of the leadership state.

There is little additional work to be done to UC-emulate the ideal-world behaviour, besides ensuring that queries are routed appropriately, especially how the reference string is queried in the hybrid world. We describe this with a small "adaptor" protocol LEDGER-ADAPTOR. This forwards most queries and treats uSRS queries by querying the appropriate part of the leader state after time δ, and by ignoring them before. Formally, it is described by:

Protocol LEDGER-ADAPTOR

The protocol adaptor fits the interface of $\mathcal{F}_{\text{NakLedger}}^{\text{real}}$ to match those of $\mathcal{F}_{\text{uSRS}}$ and

$\mathcal{F}^{\text{ideal}}_{\text{NakLedger}}$. It operates in the $(\mathcal{F}^{\text{real}}_{\text{NakLedger}}, \mathcal{G}_{\text{clock}})$-hybrid world.

When receiving a message (SUBMIT, tx) *from a party* ψ*:*

send (SUBMIT, tx) **to** $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$

When receiving a message READ *from a party* ψ*:*

send READ **to** $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$ and **receive the reply** txs

return txs

When receiving a message PROJECTION *from a party* ψ*:*

send PROJECTION **to** $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$ and **receive the reply** txs

return txs

When receiving a message LEADER-STATE *from a party* ψ*:*

send LEADER-STATE **to** $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$ and **receive the reply** $(\cdot, \sigma^{\text{ideal}})$

return σ^{ideal}

When receiving a message SRS *from a party* ψ*:*

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t

if $t < \delta$ **then return** $\perp$

else

  send LEADER-STATE **to** $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$ and

    receive the reply (srs, $\cdot$)

  return srs

Forward SUBMIT, READ, *and* PROJECTION *queries to* $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$

Formally, our real world consists of LEDGER-ADAPTOR$(\delta, \mathcal{F}^{\text{real}}_{\text{NakLedger}}(\mathcal{F}^{\mathcal{R}}_{\text{NIZK}}))$, the functionalities $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$ and $\mathcal{F}^{\mathcal{R}}_{\text{NIZK}}$ it accesses, and the global $\mathcal{G}_{\text{clock}}$.

4.2.5 Alternative Usage of $\mathcal{G}_{\text{clock}}$

In both worlds, $\mathcal{G}_{\text{clock}}$ is used to determine the cutoff point after which the reference string is deemed secure. A simple alternative to this usage of the clock is to instead rely on the length of the chain for this purpose. We did not make this choice as it complicates the ideal world: The delay wrapper would have to communicate with the ideal world ledger and query it for the length of parties' chains. We do not regard a clock as a significant additional assumption, however little of the remainder of this chapter differs if chain lengths are used instead. Even in this case, a clock is present to guarantee liveness, although it used only to constrain the adversary.

4.3 Security Analysis

Our security is derived through UC-emulation, stated in the following theorem:

Theorem 4.2. *For any updateable reference string scheme $\mathcal{S}$, satisfying correctness, structure preservation, update uniformity, update simulation with $\mathcal{S}_\rho$, and permutation extraction,* LEDGER-ADAPTOR *(in the $(\mathcal{F}_{\text{NakLedger}}^{\text{real}}, \mathcal{F}_{\text{NIZK}}^{\mathcal{R}})$-hybrid world, parameterised as in Subsection 4.2.4) UC-emulates the pair of functionalities $(\mathcal{F}_{\text{NakLedger}}^{\text{ideal}}, \mathcal{W}_{\text{Delay}}(\delta, \mathcal{F}_{\text{uSRS}}))$, parameterised as in Subsection 4.2.3, in the presence of the global clock functionality $\mathcal{G}_{\text{clock}}$, with the simulator $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$.*

As for any UC proof, we require a simulator which ensures the ideal world behaves indistinguishably from the real world. Intuitively, this simulator ensures that the real and ideal world's ledgers are equivalent and that the real world uSRS is equal to the uSRS produced in the ideal world.

In order to achieve this, the simulator ensures that the initial honest reference string provided by $\mathcal{F}_{\text{uSRS}}$ is the basis of the uSRS of a simulated execution of the real-world protocol. Doing so relies primarily on three things: First, the simulator's ability to extract the permutation from any adversarial reference string update. Second, the simulator's ability to, given the adversarial trapdoors, then produce a valid "honest" update which ensures the reference string is a random permutation of the ideal-world honest string $S(\tau_{\mathcal{H}})$. And finally, the simulator's knowledge that the final reference string in its simulation will have at least one honest update.

The simulator observes each of the competing chains and, when the first honest update occurs in each, coerces the simulated update into a permutation of the ideal honest reference string. For each subsequent honest update, the simulator performs the update normally, remembering the randomness used. Combined with extracting from adversarial updates, the simulator either knows the *entire* trapdoor of the reference string (if there was no honest update), or all except for the first honest update. By the backbone properties enforced by $\mathcal{F}_{\text{NakLedger}}$, the simulator knows that the first case will not apply, and that only one prefix of valid updates will exist, after δ time has passed. As a result, the simulator knows exactly which permutation to apply to the honest ideal reference string to match the real world's result.

As $\mathcal{F}_{\text{uSRS}}$ only provides a single honest SRS, the simulator applies a random

permutation to this for each initial honest update, ensuring that the updates of different chains remain unlinkable. The full specification of the simulator is as follows:

Simulator $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$

The simulator between the protocol adaptor over $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$, and $\mathcal{F}^{\text{ideal}}_{\text{NakLedger}}$ and $\mathcal{F}_{\text{uSRS}}$. It operates in the $\mathcal{G}_{\text{clock}}$-hybrid world.

State variables and initialisation values:

Variable	Description
$\mathcal{F}^{\text{simul}}_{\text{NakLedger}}$	A simulation of the hybrid-world ledger
$\mathcal{F}^{\mathcal{R}}_{\text{NIZK}}$	A simulation of the low-level NIZK functionality
$A \coloneqq \varnothing$	Map from honest updates to the applied permutation

When receiving a message $(\textsc{transaction}, \mathsf{tx}, t)$ *from* $\mathcal{F}^{\text{ideal}}_{\text{NakLedger}}$*:*

simulate sending $(\textsc{submit}, \mathsf{tx})$ **to** $\mathcal{F}^{\text{simul}}_{\text{NakLedger}}$

When receiving a message $(\textsc{submit}, \mathsf{tx})$ *from* $\mathcal{A}$ *for* $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$*:*

send $(\textsc{submit}, \mathsf{tx})$ **to** $\mathcal{F}^{\text{ideal}}_{\text{NakLedger}}$

When receiving a message $(\textsc{permute}, \psi)$ *from* $\mathcal{F}_{\text{uSRS}}$*:*

simulate sending LEADER-STATE **to** $\mathcal{F}^{\text{simul}}_{\text{NakLedger}}$

through ψ and

receive the reply $(\mathsf{srs}, \cdot)$

let $\vec{a} \leftarrow \mathsf{map}(\mathsf{proj}_2, \mathcal{F}^{\text{simul}}_{\text{NakLedger}}.\Pi(\psi))$

return $\mathcal{X}_p(\vec{a})$

When receiving a message $(\textsc{extend}, \psi, B, t, a)$ *from* $\mathcal{A}$ *for* $\mathcal{F}^{\text{real}}_{\text{NakLedger}}$*:*

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t'

if $\psi \in \mathcal{H} \wedge t' \leq \lceil l\gamma^{-1} \rceil s$ **then**

let $\vec{a} \leftarrow \mathsf{map}(\mathsf{proj}_2, \mathcal{F}^{\text{simul}}_{\text{NakLedger}}.\Pi(\psi))$

let $(\mathsf{srs}, \cdot) \leftarrow \mathsf{foldl}(\mathsf{Apply}, \varnothing, \vec{a})$

let $p \leftarrow \mathcal{X}_p(\vec{a})$

if $p^{-1\dagger}(\mathsf{srs}) \neq S(\tau_0)$ **then**

```
// We cannot extract a trapdoor;
// the SRS is already secure
```

let $p' \xleftarrow{*} P; \rho \leftarrow \mathsf{ProveUpd}(\mathsf{srs}, p')$

let $\mathsf{srs}' \leftarrow p'^{\dagger}(\mathsf{srs})$

simulate sending $(\textsc{prove}, \rho, p')$ **to** $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ and

receive the reply π

else

// We produce an update to match a

// random "initial" SRS

let $\tau \leftarrow p(\tau_0)$

let $p' \xleftarrow{*} P$

send HONEST-SRS **to** $\mathcal{F}_{\mathsf{uSRS}}$ and

receive the reply $\mathsf{srs}_{\mathcal{H}}$

let $\mathsf{srs}' \leftarrow p'^{\dagger}(\mathsf{srs}_{\mathcal{H}})$

let $\rho \leftarrow \mathcal{S}_{\rho}(p(\tau), \mathsf{srs}')$

query $\mathcal{A}$ **with** $(\textsc{prove}, \rho)$ and **receive the reply** π,

satisfying $\pi \neq \bot \wedge (\rho, \pi) \notin \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\overline{\Pi} \wedge (\cdot, \pi) \notin \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi$, else **sampling from** $\{0,1\}^{\kappa}$

let $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi \leftarrow \mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi \cup \{(\rho, \pi)\}$

let $A(\rho) \leftarrow p$

let $a^{\mathsf{ideal}} \leftarrow \bot$

else if $\psi \in \mathcal{H}$ **then**

let $\mathsf{srs}', \rho, \pi \leftarrow \varepsilon$

let $a^{\mathsf{ideal}} \leftarrow \bot$

else let $(\mathsf{srs}', \rho, \pi, a^{\mathsf{ideal}}) \leftarrow a$

send $(\textsc{extend}, \psi, B, t, a^{\mathsf{ideal}})$ **to** $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{NakLedger}}$ and

receive the reply $(B, a^{\mathsf{ideal}}, \mathsf{id}, t)$

if $\psi \in \mathcal{H}$ **then**

let $\mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}.\mathsf{hon}(\mathsf{id}) \leftarrow 1$

else

let $\mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}.\mathsf{hon}(\mathsf{id}) \leftarrow 0$

let $\mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}.\Pi(\psi) \leftarrow \mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}.\Pi(\psi) \parallel (B, (\mathsf{srs}', \rho, \pi, a^{\mathsf{ideal}}), \mathsf{id}, t)$

assert $\forall \psi' \in \mathcal{P}\colon \mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}.\Pi(\psi)^{\lceil k} \prec \mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}.\Pi(\psi')$

return $(B, (\mathsf{srs}', \rho, \pi, a^{\mathsf{ideal}}), \mathsf{id}, t)$

When receiving a message $(\textsc{advance}, \psi, \Sigma')$ *from* $\mathcal{A}$ *for* $\mathcal{F}^{\mathsf{real}}_{\mathsf{NakLedger}}$*:*

simulate sending $(\textsc{advance}, \psi, \Sigma')$ **to** $\mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}$

// Remove SRS updates from Σ'

let $\Sigma' \leftarrow \mathsf{map}(\lambda(B, (\cdot, \cdot, \cdot, a^{\mathsf{ideal}}), t)\colon (B, a^{\mathsf{ideal}}, t), \Sigma)$

send $(\textsc{advance}, \psi, \Sigma')$ **to** $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{NakLedger}}$

Forward requests to $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and all other adversarial messages for $\mathcal{F}_{\text{NakLedger}}^{\text{real}}$ to $\mathcal{F}_{\text{NakLedger}}^{\text{simul}}$.

Helper procedures:

procedure $\mathcal{X}_p(\vec{a})$
  let $p \leftarrow \text{id}$
  let srs = $S(\tau_0)$
  for $(\text{srs}', \rho, \pi, \cdot)$ **in** $\vec{a}$ **do**
    // Skip invalid updates
    if $\neg\text{VerifyUpd}(\text{srs}, \rho, \text{srs}') \vee (\rho, \pi) \notin \mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.\Pi$ **then continue**
    let srs $\leftarrow$ srs′
    if $(\rho, \pi) \in \mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.W$ **then**
      let $p \leftarrow \mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.W((\rho, \pi)) \circ p$
    else if $\rho \in A$ **then**
      // The update is honest.
      // Start with its permutation.
      let $p \leftarrow A(\rho)$
    else
      // A witness-less adversarial update
      // was encountered.
      abort
  return p

We will prove UC-emulation, and will therefore refer to the ideal and real worlds frequently throughout the proof. Beyond this, the simulator locally simulates the NIZK functionality and the ledger functionality. To be clear which functionality we are talking about at any point, we will use $\mathcal{F}_{\text{NakLedger}}^{\text{ideal}}$, $\mathcal{F}_{\text{NakLedger}}^{\text{simul}}$, and $\mathcal{F}_{\text{NakLedger}}^{\text{real}}$ to refer to the ideal, simulated, and real ledgers, respectively. We refer to the real-world NIZK functionality as $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and the simulted NIZK as $\mathcal{S}_{\text{LEDGER-ADAPTOR}}.\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$. The notation $\mathcal{F}.x$ is used to mean "the variable x within the functionality $\mathcal{F}$" – it is also used to refer to the ideal trapdoor $\mathcal{F}_{\text{uSRS}}.\tau_{\mathcal{H}}$.

Our simulator, which we assume is provided with the update simulation algorithm $\mathcal{S}_\rho$ and which can extract permutations from adversarial updates via a simulated NIZK, is equipped with a helper function $\mathcal{X}_p$. Given a series of updates, $\mathcal{X}_p$ computes the permutation applied to the reference string's trapdoor as far back as possible. It receives as inputs the sequence of updates $\vec{a}$, and has access to a mapping W from NIZK statements and proofs to corresponding witnesses

(as far as the simulator knows them) and a mapping A from honest updates to the permutation applied to the honest SRS. It returns a permutation in P, which can be applied either to the initial trapdoor τ_0, or to the initial honest trapdoor $\tau_{\mathcal{H}}$, to create the same SRS as the sequence of updates. We prove this in the following auxiliary lemma that will be used in the proof of our main theorem.

Lemma 4.1. *In the ideal-world execution of $\mathcal{S}_{\text{LEDGER-ADAPTOR}}$, $\mathcal{X}_p(\vec{a})$ outputs a permutation $p \in P$, such that its inverse, applied to the underlying trapdoor of the SRS generated from the given sequence of updates $\vec{a}$, is either the initial trapdoor τ_0, or the honest trapdoor $\tau_{\mathcal{H}}$.*

Proof. The output of $\mathcal{X}_p$ is either id, a permutation in the mapping A, a permutation recorded by the simulated NIZK, or a series of function compositions of the above. As only permutations in P are stored in A, $\mathsf{id} \in P$ and, as P is closed under composition, the returned permutation is in P. The permutation applied corresponds directly to how the underlying trapdoor of the uSRS is updated by longest suffix of updates in $\vec{a}$ for which the trapdoor is known – that is, the trapdoor permutation is recorded in $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W$, or a permutation of the honest trapdoor is recorded in A. When this is not the case, the update is skipped and the trapdoor reset, ensuring that any trapdoors preceeding a non-extractable value are ignored. The case that the trapdoors are known for *all* of the updates is trivial; as by definition inverting this permutation will result in the initial trapdoor τ_0.

If, however, at any point the trapdoor is not recorded in $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W$ (despite $\mathsf{VerifyUpd}$ succeeding), at this point the trapdoor must be honestly generated: As this update was not skipped, the NIZK proofs associated with it must verify. The only way for the proofs to verify and the NIZK functionality *not* to have recorded the corresponding witnesses, however, is that the simulator added the proof manually to the NIZK's set of valid proofs. This only happens at one point – when creating simulated NIZK proofs to accompany simulated update proofs, which is used *only* for random permutations applied to the honest reference string. While (if the adversary is capable of inverting the structure function) multiple honest updates may exist in the same chain, if at least one of them is a replayed update, the last such effectively "resets" the reference string to a known permutation of the honest reference string.

Finally, we note that for this witness-less update, the remaining trapdoor defines a permutation of $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$. Algorithm $\mathcal{X}_p$ extracts the trapdoors from all

subsequent updates to compute the permutation applied to this honest trapdoor – ensuring precisely that inverting this permutation results in $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$. □

Proof (of Theorem 4.2). If the environment can distinguish between these worlds, there must exist a minimal series of interactions the environment, combined with its adversary, can make to cause the other UC ITMs to behave sufficiently differently to allow distinguishing. We will show that for any interaction the environment makes, it will not learn enough information to distinguish the two worlds and therefore that across *all* (polynomially many) interactions it also cannot distinguish. First, we consider what actions the adversary/environment pair can take. The interactions fall into the following categories:

1. Honest or adversarial SUBMIT, READ, LEADER-STATE, or PROJECTION queries
2. Interactions with $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$, or $\mathcal{G}_{\mathsf{clock}}$
3. ADVANCE queries
4. EXTEND queries
5. SRS queries

We will establish the following invariants throughout the execution of the UC security game:

- $\mathcal{G}_{\mathsf{clock}}$ has the same internal state in both the real and ideal worlds.
- $\mathcal{S}_{\textsc{ledger-adaptor}}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$ has the same internal state as the real-world $\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$, except that it does not know the witnesses for honestly generated proofs or their mauled variants.
- $\mathcal{S}_{\textsc{ledger-adaptor}}.\mathcal{F}^{\mathsf{simul}}_{\mathsf{NakLedger}}$ has the same internal state as the real-world ledger $\mathcal{F}^{\mathsf{real}}_{\mathsf{NakLedger}}$ and differs from the ideal-world ledger $\mathcal{F}^{\mathsf{ideal}}_{\mathsf{NakLedger}}$ only in that all state updates contain an addition SRS update term.

Ledger reads and submissions. Given these invariants, it is clear that the environment cannot distinguish given the results of READ and PROJECTION queries – they must return the same value! Furthermore, as the adaptor protocol strips the SRS component from the leader state and the ideal world's leader state is precisely defined as being without this component, it is clear that also LEADER-STATE queries will be indistinguishable (even if made directly by the

adversary, since these are answered by $\mathcal{F}^{\text{simul}}_{\text{NakLedger}}$). For SUBMIT queries by either the environment or the adversary, both worlds will add the transaction, with the current timestamp, to their ledger's submitted transactions, and will notify the adversary *once* and return the transaction together with the timestamp. This does not reveal any information to the environment which could be used to distinguish.

Queries to other functionalities. Likewise, $\mathcal{F}^{\mathcal{R}}_{\text{NIZK}}$ queries clearly will not permit the environment to distinguish, or invalidate the above mentioned invariants – they do not go beyond the NIZK functionality, and this does not read (only update) the witness map. Similarly for $\mathcal{G}_{\text{clock}}$, as this exists in both worlds and is not manipulated by the simulator (or any other entity), beyond read-only operations, it will behave identically.

ADVANCE queries. The simulator first simulates advancing a specified party's ledger state on $\mathcal{F}^{\text{simul}}_{\text{NakLedger}}$. If this succeeds, the simulator knows that the advancement will succeed in the ideal world as well, where the ledger state is less constrained. It removes the SRS updates from the ledger state being switched to and issues a corresponding advance query to $\mathcal{F}^{\text{ideal}}_{\text{NakLedger}}$. If the simulated ADVANCE does not succeed, it will also have failed in the real world execution, both of which will abort. If the update succeeds, the invariant between the various ledger states is preserved – up to the lack of SRS updates in the ideal world, they are the same. If the update fails, both worlds terminate execution.

(EXTEND, ψ, B, t, a) queries. Let us first detail the function of EXTEND queries. Called by the adversary, if the party parameter ψ represents an honest party, the query runs Gen to generate a new update a to apply to this party's view of the leadership state. If the party is adversarial on the other hand, an adversary-supplied update parameter a is used instead. With the timestamp t (or the accurate time for honestly created blocks), block content B, state update a, and a randomly sampled ID, a new block is created and appended to ψ's projected chain. Finally, it is asserted that the common prefix property still holds.

Once the simulator intercepts such a query, it needs to ensure not only that the same EXTENDs are carried out in the simulated and ideal ledgers, but also that honest SRS updates are (when necessary) sourced from the $\mathcal{F}_{\text{uSRS}}$ function-

ality. In the case that the party extending the chain is adversarial, this is simple – split the adversarial real world update a into an SRS update and an ideal-world update (it is worth noting that these need not be valid), and forward only the ideal-world update in an EXTEND query to $\mathcal{F}^{\text{ideal}}_{\text{NakLedger}}$. This already results in the real and ideal ledgers satisfying the invariant, leaving the simulated ledger. For this, the simulator manually inserts the ID returned from the ideal-world ledger, inserts the new block, and asserts the same common prefix condition as the real world does, ensuring these two ledgers are in the same state and – crucially – abort under the same conditions. The returned value is identical to that returned in the real world.

For honest updates, things are more complex. If the current time is after when honest SRS updates are performed, the honest SRS update is set to ε, as in the real world. Otherwise, the SRS is reconstructed from the party's current projected ledger view and the simulator attempts to extract the trapdoor permutation from this SRS. If it succeeds in extracting the entire trapdoor, the simulator ensures it is updated such that it can no longer do so: It updates the uSRS to a permutation of the honest uSRS $\mathcal{F}_{\text{uSRS}}.\tau_{\mathcal{H}}$, by first applying a fresh permutation to it, recording this in the map A, and creating the corresponding update proof using $\mathcal{S}_\rho$.

By the update uniformity property, this is indistinguishable from the result of Gen, which the environment expects. In case the full trapdoor cannot be extracted, Gen is used to generate the "honest" SRS update, ensuring the simulator knows the trapdoor for this update as well (as it retains the NIZK witness used). Finally, the ideal ledger is sent an EXTEND query, with a^{ideal} set to $\perp$. Execution proceeds as in the adversarial case, with the SRS part of the update being distributed equally in the real and simulated ledgers, and the ideal-world component being generated directly by the ideal world functionality (and therefore also being distributed the same as in the real world, which samples from the same distribution).

SRS queries. Finally, a user may query the SRS. If this happens before time δ, both worlds return $\perp$ – the delay wrapper does so in the ideal world and the adaptor protocol does so in the real world. Otherwise, the real world reconstructs the leadership state and returns only the SRS component, while the ideal world queries the simulator for a trapdoor permutation and, if the SRS is not yet fi-

nalised, applies it to the honest SRS.

Recall that after every extension, $\mathcal{F}_{\mathsf{NakLedger}}$ ensures that the common prefix property holds. Further, once a party's projected ledger state has some common prefix, this is only ever extended – either by extending the whole projection (in EXTEND), or by switching to a different one with the same prefix (in ADVANCE). After time δ, if chain quality and liveness hold, we can split each party's projected chain into two parts: Blocks with a timestamp at or before the time δ_1, and those with a timestamp after it. As EXTEND enforces timestamps to be monotonically increasing, these concatenate to form the entire chain. By the chain growth property, and as it is at least time δ, we know that the first part contains at least l blocks and the second at least k blocks. Chain quality ensures that the first part contains at least μ honest blocks, while Apply ignores updates with a timestamps after δ_1. Combined, these facts imply that, for any party, the valid SRS updates, taken from their stable chain, are identical.

After the first SRS query, both the ideal and real worlds will not change what value they return, the former because it has then recorded the final trapdoor and the latter because the common prefix containing valid reference string updates cannot change. The first query is therefore the most interesting.

From Lemma 4.1, we know that the permutation p extracted by the simulator when it is queried for the SRS permutation will, inverted and applied to the SRS' underlying trapdoor, either result in τ_0, or $\mathcal{F}_{\mathsf{uSRS}}.\tau_{\mathcal{H}}$. From the above we know that the SRS the simulator is extracting from matches that honest parties generate – containing at least one honest update (by chain quality). As the first honest update in any chain is extracted from an $\mathcal{F}_{\mathsf{uSRS}}$-provided reference string, (and, by the correctness property, it *is* valid) it cannot be τ_0. Therefore, the simulator, by providing p to $\mathcal{F}_{\mathsf{uSRS}}$, satisfies its requirements of a permissible permutation in P and ensures that once the permutation is applied, the same SRS is returned: $S(p(\tau_{\mathcal{H}})) = S(p(p^{-1}(\tau))) = S(\tau)$.

In the above we have brushed aside the issue of aborts, however these are also simple to deal with. $\mathcal{F}_{\mathsf{uSRS}}$ aborts if given an invalid permutation, which the simulator does not do. In the real world, if liveness or chain quality are violated, $\mathcal{F}_{\mathsf{NakLedger}}$ aborts. In each query, the simulator ensures that the same query is run against the simulated ledger, ensuring that both will abort under the same conditions. This is the primary purpose for which $\mathcal{F}_{\mathsf{uSRS}}$ asks for a permutation on each invocation, despite only using it on the first, as well as why it supplies

the identity of the calling party to the ideal-world adversary. □

As it is possible to construct (non-succinct) non-interactive zero-knowledge schemes from a random oracle, we can remove the requirement on $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and instead rely on a random oracle $\mathcal{F}_{\text{RO}}$ (formally described in Subsection 2.3.5). As almost all constructions of Nakamoto-style ledgers are in the random oracle model, our usage of a low-level NIZK is not a major additional assumption.

Corollary 4.1. *For any updateable reference string scheme $\mathcal{S}$, it is possible to realise the pair of functionalities $(\mathcal{F}_{\text{NakLedger}}^{\text{ideal}}, \mathcal{W}_{\text{Delay}}(\delta, \mathcal{F}_{\text{uSRS}}))$ in the $(\mathcal{F}_{\text{NakLedger}}^{\text{real}}, \mathcal{F}_{\text{RO}})$-hybrid world and in the presence of $\mathcal{G}_{\text{clock}}$.*

4.4 Implementation and Parameter Selection

We have implemented [Ker20] Sonic's update mechanism (see Subsection 4.1.3) and, using this, provide performance estimates for SRS generation in a live blockchain network. Further, we simulate the optimal adversarial attack strategy and demonstrate how this may be used to select optimal parameters for the secure generation of reference strings. We demonstrate that for currently typical applications, these parameters are practical for real-world usage.

While we have not modified a full blockchain client to utilise this extended consensus, we discuss the impact it would have on each of the following points:

- block verification
- block generation
- chain reorganisation
- network usage
- local storage

While the Bitcoin backbone paper [GKL15] provides bounds on chain parameters in given situations, these have three main drawbacks in the context of this chapter:

1. The bounds are not tight.
2. The criteria for security are stricter than required: It asserts liveness and persistence are never violated, while this chapter only requires them in a few select cases.

3. The analysis is in the synchronous model – while the generation and verification of reference strings can take a significant amount of time.

To obtain sensible parameters to generate reference strings, we measure the time taken for computing and verifying updates, and factor this processing overhead into a simulation of the optimal adversarial strategy to subvert the SRS generation procedure.

The implementation and numbers provided for execution time and storage use the commonly used BLS12-381 curve pair. Circuits which have been practically deployed tend to require a depth of at most half a million, so we will often assume a Sonic uSRS depth of 500,000. All data shown is available at [Ker20] and may be reproduced with the provided source code.

4.4.1 Execution Time of uSRS Operations

We tested our implementation of the uSRS generation mechanism on an AMD Ryzen 7 2700X 8-core processor with hyper-threading enabled. This processor is a standard consumer-grade CPU – in proof-of-work mining it is likely that miners will have access to better hardware. All operations have been parallelised, and the verification operation has been additionally optimised to use fewer pairing operations. The workload, especially for uSRS generation, is also highly parallelisable (consisting of primarily a large number of group exponentiations), suggesting further improvements by utilising GPUs and clusters of machines are possible. If such improvements are applied, the total time *delay* required for the secure generation procedure, as well as the optimal intended block time could be reduced proportionally to the increase in parallelisation; assuming paralellisation across 10 machines could reduce both by an order of magnitude, for instance.

We measured the time taken to create and verify a uSRS update in relation to the uSRS depth in Figure 4.1. For our NIZK, we use a UC-secure Fischlin proof, described in Subsection 4.1.3. We measure the overhead of these proofs to be 23.956ms for proving and 1.567ms for verifying (a Fiat-Shamir proof of the same type was measured to 0.921ms and 0.870ms respectively), using SHA-3 in place of a random oracle. For larger dimensions of reference strings, neither have much impact on the total runtime.

Finally, we implemented *aggregate updates*: The bulk of Sonic's update verifica-

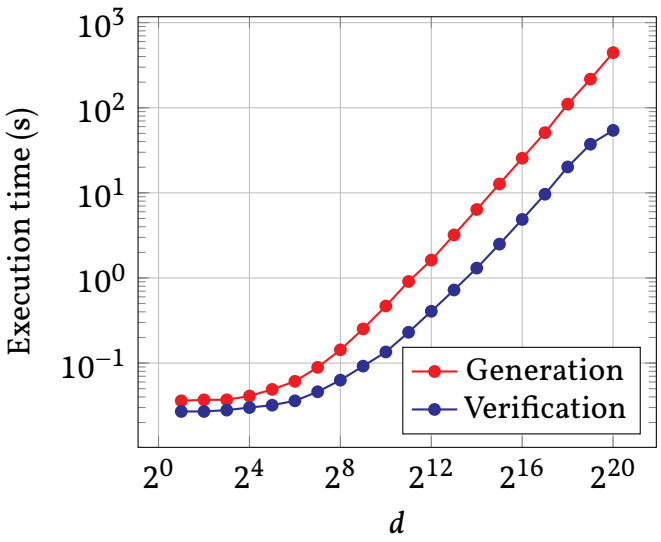


Figure 4.1: The time taken to produce and verify uSRS updates, as a function of the Sonic SRS depth d.

tion procedure is concerned with verifying the structure of the reference string, while a few parts of it verify that it is an exponentiation of the previous string. By retaining only the latter parts, a series of updates can be verified almost as quickly as a single update. The verification of aggregate proofs has an overhead of 1.634ms per update included in the aggregate. The bulk of this cost arises from the verification of the Fischlin proof. This allows for even large chain reorganisations to be quickly verified.

4.4.2 Simulating the Optimal Attack Strategy

The mechanism we have presented in this chapter operates in two phases. In the first phase, the adversary has the chance to *subvert* the reference string, while in the second phase it can carry out a denial of service attack, potentially convincing users that an incorrect (but not subverted) reference string is the canonical one.

For the first phase, the adversary's optimal strategy is to mine entirely independently from any honest activity: the adversary cannot adopt any honest block – doing so would break the subversion of its reference string. Further, the adversary has no reason to share any of its own blocks except if it reached the threshold of having a fully valid subverted reference string – it only gives the honest network a chance to catch up, in the case that the adversary is ahead. This

allows for a straightforward simulation of the consensus protocol: The probability of either honest parties, or the adversary creating an individual block is exponentially distributed. In addition to this, honest parties have a fixed processing overhead before they may start mining: This may include a networking delay, but more crucially it includes the time taken to verify a newly received block's uSRS update and to produce the subsequent update. We assume that the adversary can bypass large parts of this overhead, by virtue of network dominance, by skipping verification and by producing reference string updates with small (and therefore insecure) exponents.

The overhead manifests as shifting the honest party's exponential distribution for block generation by a fixed constant. More precisely, we parameterise each experiment by:

- The intended time between blocks b
- The combined networking and update overhead d
- The fraction of adversarial mining power α

Of these three, d can be seen as fixed, depending on the depth of the uSRS being generated and the corresponding speed of verification and update generation. For simplicity, we assume a uSRS depth of 500,000, which corresponds to d being approximately 250 seconds on our single-CPU setup.

We draw the time of the next adversarial block from the exponential distribution with $\lambda = \alpha/b$, and the next honest block from the exponential distribution with $\lambda = (1-\alpha)/b$, shifted to the right by d (that is, the probability density is 0 for $x < d$). The simulation is then advanced to the lesser of the two times, which is resampled from the same distribution. The number of times the adversary or the honest parties have extended their chain is counted and the honest parties win at any point if and only if the honest chain is longer than the adversarial chain.

We ran one million experiments in parallel, either up to a fixed end time, or until a large enough fraction of the experiments end in honest victory. We refer to the probability of an adversarial success as the probability of subversion ε. Figure 4.2 demonstrates that for a fixed d, a trade-off exists between the target time between blocks b and the time until any given subversion threshold ε is met.

A practical limit of this simulation approach is that it cannot by itself determine the length of time needed to wait until ε is negligible for most typical security parameters. We can however observe that for fixed parameters, ε decreases

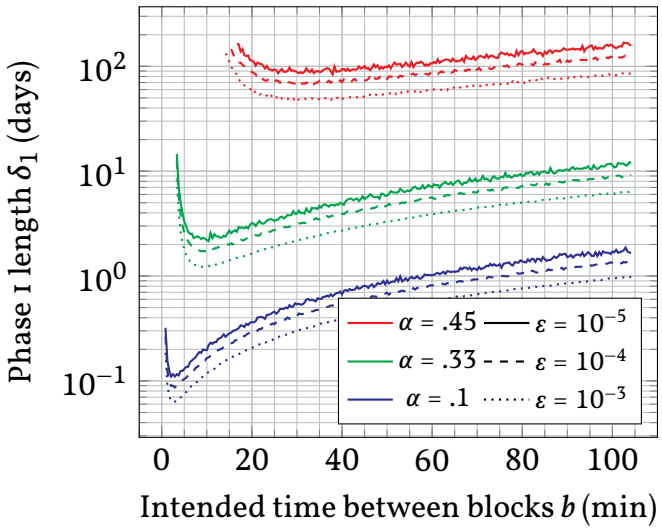


Figure 4.2: The time required to generate a secure uSRS, as a function of the intended time between blocks. This depends on the proportion of adversarial mining power α and the bound ε on the probablity of subversion. Each data point represents the time until at most a fraction of ε of one million parallel experiments ended in adversarial victory. Values are given assuming $d = 250$ s and both axes scale linearly to d.

approximately exponentially as time passes, as seen in Figure 4.3, outside of a brief initial window.

While the second phase – that where the adversary attempts to create disagreement as to which reference string is the canonical one – may initially seem different, its optimal strategy is identical, as it essentially wishes to create as long as possible a fork, starting one block prior to the end of the first phase (to select a different reference string). As creating the longest fork *forking at this point* does not allow the adversary to accept honest blocks after it, nor gives the adversary a reason to share its blocks, the adversarial strategy is the same and therefore the same analysis applies.

4.4.3 Storage and Network Usage

A Sonic reference string consists of $4d + 1$ elements in $\mathbb{G}_1$ and $4d + 2$ elements in $\mathbb{G}_2$. For the commonly used BLS12-381 curve pair, $\mathbb{G}_1$ elements have a storage requirement of 48 bytes each and $\mathbb{G}_2$ elements of 96 bytes each. An update proof includes an additional two $\mathbb{G}_1$ elements and a Fischlin proof, which itself con-

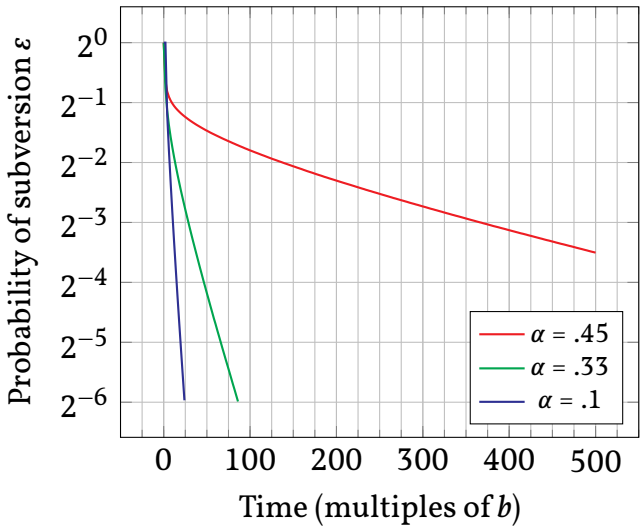


Figure 4.3: The probability of the reference string being subverted ε, as a function of the time passed, in multiples of the intended time between blocks b. This depends on the proportion of adversarial mining power α and the compound overhead d. b is selected to be approximately at the minimum seen in Figure 4.2, with $d = .15b$, $d = .4b$, and $d = 2b$ for the $\alpha = .45, .33$, and .1, respectively.

sists of twelve iterations, each with 2 elements in $\mathbb{F}_q^*$ (each of which requires 32 bytes to store), two elements of $\mathbb{G}_1$, and a 16-bit nonce. Each part of an aggregate update has an additional two $\mathbb{G}_2$ elements.

As it is not necessary to retain intermediate reference strings, and aggregate updates are sufficient, for a chain of length l, and with an uSRS depth of d, this is a storage requirement of $576d + 288$ bytes for the uSRS itself and $l \cdot (2 \cdot 48 + 2 \cdot 96 + 12 \cdot (2 \cdot 32 + 2 \cdot 48 + 2)) = 2,232l$ bytes for storing updates.

For 500,000 gates and chains of length 20,000, this corresponds to a total storage requirement of 318MiB, with the reference string itself being the largest part, at 275MiB. Although this is quite manageable as a storage requirement, it must be considered that the SRS itself (and a single update of around 2KiB) has to be re-transmitted with each block. While at the common home-internet upload speed of 10Mb/s, a block would take slightly under 4 minutes to transmit, it is reasonable to assume that miners would invest in high-grade connections to offset the chance of their block being replaced with a competitors. Speeds up to 10Gb/s are commercially available, which would reduce the transmission time to under a second.

One remaining issue is that of denial-of-service. The receipt and verification

of a reference string is costly, and should therefore be done only *after* a block's proof-of-work has been received, which should depend on a commitment to the subsequently sent reference string – such as the update proof itself. An attacker can still perform a limited denial of service attack with blocks they legitimately mined – however this uses no more resources in verification than a legitimate block would.

4.4.4 Conclusion

Figure 4.2 provides insight into the space of tradeoffs which can be made for the secure generation of reference strings. While the secure generation of a reference string is possible even for a small honest majority, the time required to do so is much higher than for a more relaxed setting, with δ_1 being approximately three *months* for $\alpha = .45$, in contrast to around two *days* for $\alpha = .33$. The full setup is double this: six months for $\alpha = .45$ and four days for $\alpha = .33$. Perhaps surprisingly, the desired probability of subversion ε has a more muted effect on the required setup time.

The minima observed for δ_1 suggest that simply deploying this system on existing blockchain systems as they are currently parameterised is unwise: Most blockchains emphasise small values of b to enable transactions to settle quickly, with even notoriously slow chains such as Bitcoin having values on the lower end of our scale. This is directly linked to the compound overhead of verification and update generation – when b is small, the adversary can better use its advantage of bypassing large parts of the verification and update procedure. As previously noted, there is a lot of room for speedup by assuming miners use greater computation power – if each miner used ten machines, even the $\alpha = .45$ case would be reduced to under a month in total.

4.5 Low-Entropy Update Mitigation

While our analysis indicates that in a Byzantine, honest majority setting, our protocol produces a trustworthy reference string, it also asks participants to dedicate computational resources to updates. It follows that in a rational setting, players need to be properly incentivised to follow the protocol. We emphasise that the rational setting is not our focus, and optimistically, in a setting where

the majority of miners are rational and a small fraction honest, the few honest blocks are sufficient to eliminate the issue described in this section.

For Sonic, a protocol deviation exists that breaks the security of the reference string: By choosing the exponent in a specific low-entropy fashion, (e.g., $y = 2^l$) the computation of the update, which primarily relies on repeated squaring, can be done significantly faster. More generally, some permutations in P may be more efficiently computable. In more detail, instead of using a random permutation p, a specific choice is made that eases the computation of srs' – in the most extreme case, for any uSRS scheme, the update for $p = \mathsf{id}$ is trivial.

4.5.1 Proposed Construction

In order to facilitate a mitigation for this class of attacks, we will need to assume an additional property of the underlying ledger, in particular it must provide a "resettable" randomness beacon: With each ADVANCE operation (where the adversary must be restricted in how often it may do such ADVANCE queries), a random beacon value is sampled in a variable bcn and is associated with the corresponding block. Prior work [DGKR18] demonstrates that such beacon values allow for the adversary to bias them only by "resetting" it at most a certain number of times, say t, before they are fixed by entering the ledger's confirmed state, with the exact value of t depending on the chain parameters.

We can then amend Gen to derive its random values from the random oracle, by sending the query (bcn, nonce) to $\mathcal{F}_{\mathrm{RO}}$, where nonce is a randomly selected nonce, and bcn is the previous block's beacon value. The response is used to index the set of trapdoor permutations P, choosing the result p, and the nonce is stored by miners locally, and kept private. We adapt the Phase 1 period δ_1 so that at least $l' \coloneqq l(1-\theta)^{-1} + c$ blocks will be produced, where θ and c are new security parameters (to be discussed below). Next, after Phase 2 ends, we can be sure that the beacon value associated with the end of Phase 1 has been reset at most t times.

We extract from bcn l' biased coins, each with probability θ. For each block, if the corresponding coin is 1, it is required to reveal its randomness within a period of time at least as long as the liveness parameter. Specifically, a party which created one of the selected blocks may reveal its nonce. If its update matches this nonce, the party receives an additional reward of value R times the standard block reward.

While this requires a stricter chain quality property, with the ledger functionality instead enforcing that one of these l non-opened updates are honest, we sketch why this property still holds in the next section.

4.5.2 Security Intuition

Consider now a rational miner with hashing power α. We know that, at best, using an underlying blockchain like Bitcoin, the relative rewards such a miner may expect are at most $\alpha/(1-\alpha)$ in expectation; this assumes a selfish mining strategy that wins all network races against the other rational participants. Now consider a miner who uses low entropy exponents to save on computational power on created blocks and, as a result, boosts their hashing power α to an increased relative hashing power of $\alpha' > \alpha$. The attacker can further try to influence the blockchain by forking and selectively disclosing blocks which has the effect of resetting the bcn value to a preferred one. To see that the impact of this is minimal, we prove the following lemma.

Lemma 4.2. *Consider a mapping $\rho \mapsto \{0,1\}^{l'}$ that generates l' independent biased coin flips, each with probability θ, when ρ is uniformly selected. Consider any fixed $n \leq l'$ positions and suppose an adversary gets to choose any one out of t independent draws of the mapping's random input with the intention to increase the number of successes in the n positions. The probability of obtaining more than $n(1+\varepsilon)\theta$ successes is* $\exp(-\Omega(\varepsilon^2\theta n) + \ln t)$.

Proof. In case $t = 1$, result follows from a Chernoff bound on the event E defined as obtaining more than $n(1+\varepsilon)\theta$ successes, and has probability $\exp(-\Omega(\varepsilon^2\theta n))$. Given that each reset is an independent draw of the same experiment, by applying a union bound we obtain the lemma's statement. □

The optimal strategy of a miner utilising low-entropy attacks is to minimise the number of blocks of other miners are chosen, to increase its relative reward. Lemma 4.2 demonstrates that at most a factor of $(1+\varepsilon)^{-1}$ damage can be done in this way. Regardless of whether a miner utilises low-entropy attacks or not, their optimal strategy beyond this is selfish mining, in the low-entropy attack mining in expectation $l'\alpha'/(1-\alpha')$ blocks [GKL15]. A rational miner utilising low-entropy attacks will not gain any additional rewards, while a miner not doing so will gain at least $l'\alpha/(1-\alpha)(1+\varepsilon)^{-1}\theta R$ rewards from revealing their randomness, by Lemma 4.2. It follows that for a rational miner, this strategy can be

advantageous to plain selfish mining only in case:

$$\frac{\alpha'}{1-\alpha'} > (1+\theta(1+\varepsilon)^{-1}R)\frac{\alpha}{1-\alpha}$$

If we assume a miner can increase their effective hash rate by a factor of c, using low-entropy exponents, then their advantage in the low entropy case is $\alpha' = \alpha c/(\alpha c + \beta)$, where $\beta = 1 - \alpha$ is the relative mining power of all other miners. If follows that the miner benefits if and only if:

$$\begin{aligned} \frac{\alpha c}{\alpha c+\beta} \cdot \frac{\alpha c+\beta}{\beta} &> (1+\theta(1+\varepsilon)^{-1}R)\frac{\alpha}{\beta} \\ \iff c &> 1+\theta(1+\varepsilon)^{-1}R \end{aligned}$$

If we adopt a sufficiently large intended time interval between blocks it is possible to bound the relative savings of a selfish miner using low-entropy exponents; following the parameterisation of Subsection 4.4.2, if a selfish miner using such exponents can improve their hashing power by at most a multiplicative factor c then we can mitigate such an attack by setting R to $(c-1)/(\theta(1+\varepsilon)^{-1})$.

4.6 Discussion

While the clean generation of a new reference string from a ledger protocol is itself useful, real-world situations are likely to be more complex. In this section we discuss practical adjustments that may be made.

4.6.1 Upgrading Reference Strings

As distributed ledgers are typically long-lived and may well outlive any reference string used within it – or have been running before a reference string was needed – a secure process to upgrade reference strings is important. Indeed, the Zcash protocol has seen upgrades in its reference string. A reference string being replaced with a new one is innocuous without further context, however it is important to consider how they are usually used in zero-knowledge proofs. If the proof they are used in is stateless, upgrading from an insecure to a secure reference string behaves as one may naively expect: It ensures that after the upgrade, security properties hold.

In the example of Zcash, which runs a variant of the Zerocash [BCG+14] protocol, the situation is more muddy. Zerocash makes *stateful* zero-knowledge

proofs. Suppose a user is sceptical of the security of the initial setup – and there is good reason to be [SWB19] – but is convinced the second reference string is secure. Is such a user able to use Zcash with confidence in its security?

Had Zcash not had safeguards in place, the answer would be no. While the protocol may operate as intended currently and the user can be convinced of that, due to the stateful nature of the proofs, the user cannot be convinced of the correctness of this state. The Zcash cryptocurrency did employ similar safeguards to those we outline below. We stress the importance of such safeguards here, as not every project may have the same foresight.

Specifically, for a Zerocash-based system, an original reference string's backdoor could have been used to create mismatched transactions and to effectively "mint" large coins illicitly. This process is undetectable at the time and the minted coins would persist across a reference string upgrade. Our fictitious user may therefore be rightfully suspicious as to the value of any coins he is sold – they may be a part of an almost infinite pool!

Such an attack, once carried out (especially against a currency) is hard to recover from – it is impossible to identify "legitimate" owners of the currency, even if the private transaction history were deanonymised and the culprit identified. The culprit may have traded whatever he created already. Simply invalidating the transaction would therefore harm those he traded with, not himself. In an extreme case, if he traded one-to-one with legitimate owners of the currency, he would succeed in effectively stealing the honest users funds. If such an attack is identified, the community has two unfortunate options: Annul the funds of potentially legitimate users, or accept a potentially large amount of inflation.

We may assume a less grim scenario however: Suppose we are *reasonably confident* in the security of our old reference string, but we are *more confident* of the new one. Is it possible to convince users that we have genuinely upgraded our security? We suggest the usage of a type of *firewalling* property. Such properties are common in the domain of cross-chain transfers [GKZ19] and are designed to prevent a catastrophic failure on one chain damaging another.

For monetary transfers, the firewall would guarantee an upper-bound of funds was not exceeded. Proving the firewall property is preserved is easy if a small loss of privacy is accepted – each private coin being re-minted before it can be used after the upgrade, during which time its value must be declared. Assuming everything operates fine and the firewall property is not violated,

users interacting with the post-firewall state can be confident as to the upper bound of funds available. Furthermore, attacks on the system can be identified: If an attacker mints too many coins, eventually the firewall property will be violated, indicating that too many coins were in circulation – bringing the question of how to handle this situation with it. We believe that a firewall property does give peace of mind to users of the system and is a practical means to assuage concerns about the security of a system which once had a questionable reference string.

In Zcash, a soft form of such firewalling is available, in that funds are split across several "pools", each of which uses a different proving mechanism. The total value of each pool can be observed, and values under zero would be considered a cause for alarm and rejected. Zcash uses the terminology "turnstiles" [Zca19] and no attacks have been observed through them.

A further consideration for live systems is that as Subsection 4.4.2 shows, the time required strongly depends on the frequency between blocks. This may conflict with other considerations for selecting the block time – a potential solution for this is to only perform updates on "superblocks": blocks which meet a higher proof-of-work (or other selection mechanism) target than usual.

4.6.2 The Root of Trust

An important question for all protocols in the distributed ledger setting is whether a user entering the system at some point during its runtime can be convinced to trust in its security. Early proof-of-stake protocols, such as [KRDO17], did poorly at this and were subject to "stake-bleeding" attacks [GKR18], for instance – effectively meaning new users could not safely join the network.

For reference strings, if a newly joining user is prepared to accept that the honest majority assumption holds, they may trust the security of the reference string, as per Theorem 4.2. There is a curious difference to the security of the consensus protocol however: to trust the consensus – at least for proof-of-work based protocols – it is most important to trust a *current* honest majority, as these protocols are assumed to be able to recover from dishonest majorities at some point in their past. The security of the reference string on the other hand only relies on assuming honest majority during the initial δ time units. This may become an issue if a large period of time passes – why should someone trust the

intentions of users during a different age?

In practice, it may make sense to "refresh" a reference string regularly to renew faith in it. It is tempting to instead continuously perform updates, however as noted in Subsection 4.6.1, this does not necessarily increase faith in a stateful system, although is can remove the "historical" part from the honest majority requirement when used with stateless proofs.

Most subversion attacks are detectable – they require lengthy forks which are unlikely to occur during a legitimate execution. In an optimistic case, where no attack is attempted, this may provide an additional level of confirmation: if there are no widespread claims of large forks during the initial setup, then the reference string is likely secure (barring large-scale out-of-band censorship). A flip side to this is that it may be a lot easier to sow doubt, however, as there is no way to *prove* this: A malicious actor could create a fork long after the initial setup and claim that it is evidence of an attack to undermine the credibility of the system.

4.6.3 Applications to Non-Updateable SNARKs

Updateable SNARK schemes have two distinct advantages which our protocol makes use of: First, they have an explicit update procedure which allows a party ψ to replace a reference string whose security depends on some assumption A, with one whose security depends on $A \vee (\psi \text{ is honest})$. Second, they can survive with a partially biased reference string, a fact which we do not use directly in this chapter, however the functionality $\mathcal{F}_{\mathsf{uSRS}}$ we provide permits rejection sampling, encoding it into the ideal world.

The lack of an update algorithm can be resolved for some zk-SNARKs, such as [Gro16], by the existence of a weaker property: In two phases, the reference string can be constructed with (potentially different) parties performing round-robin updates (also group exponentiations) in each phase. This approach is also detailed in [BGM17], and it implies a natural translation to our protocol, in which the first phase is replaced with two phases of the same length, performing the first and second phase updates, respectively.

The security of partially biased references strings has not been sufficiently analysed for non-updateable SNARKs, however this weakness can be mitigated. Following [BGM17], it is possible to use a pure random beacon (as opposed to

the resettable one used in Section 4.5) to create a "pure" reference string from the "impure" one presented so far. To sketch the design: The random beacon would be queried after time δ, and the randomness used to select a trapdoor permutation over the reference string. This would then be applied by each party independently, arriving at the same – randomly distributed – reference string.

As this is not required for updateable SRS schemes, we did not perform this analysis in depth. However the approach to the simulation would be to perform the SRS generation identically, and then program the random beacon to invert all permutations applied to the honest reference string. Since this includes the one honest permutation applied on every honest update, this is indistinguishable from a random value to the adversary. It is worth noting that the requirement of a random beacon is on the stronger side of requirements, especially as it should itself not allow adversarial influence to provide the desired advantage. Approaches using block hashes for randomness introduce exactly the limited influence which we are attempting to remove!

5 Privacy in Proof-of-Stake

This chapter is based on "Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake" [KKKZ19], first published at the 2019 IEEE Symposium on Security and Privacy, primarily authored by Thomas Kerber, and co-authored by Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas.

Construction of both privacy-preserving currency systems, such as Zerocash [BCG+14], and secure proof-of-stake, such as Ouroboros [KRDO17, DGKR18, BGK+18], has been well-studied. Combining both is a natural goal and goes beyond a simple parallel composition of two systems, as proof-of-stake is intrinsically interwoven with the currency, requiring knowledge of how much stake users own by definition. This poses significant modelling challenges, and questions of how the leakage from the proof-of-stake protocol can be minimised.

This chapter proposes a mechanism to combine the currency and ledger (while still allowing for flexible alternative usage), by allowing transactions to be addressed only to specific users. Furthermore, it presents Crypsinous, a proof-of-stake protocol which realises this ledger from a privacy-preserving currency. This is based on the Ouroboros Genesis protocol [BGK+18]. Crucially this analysis is not only universally composable and privacy-preserving, but also forward-secure, ensuring that privacy is preserved independently of any other protocols running concurrently, even considering adaptive corruption.

Proof-of-stake and transaction privacy is, seemingly, a contradiction in terms: issuing a block by proof-of-stake fundamentally leaks information about the issuer and the state of the ledger. We circumvent the contradiction by adapting the techniques of Zerocash's "transaction pouring" to our setting. A notable difference is that coins *evolve* when used in a proof-of-stake eligibility proof, allowing them to be both reused and spent without being linked to this proof.

The design has several subtleties since a critical consideration in the PoS setting is tolerating adaptive corruptions: this ensures that even if the adversary can corrupt parties in the course of the protocol execution in an adaptive manner, it does not gain any non-negligible advantage by, for instance, re-issuing past PoS blocks. In non-private PoS protocols such as Algorand [GHM+17] and Ouroboros Genesis [BGK+18] this is captured by employing forward secure signatures. In the context of our protocol however, a more sophisticated combination of key-private forward-secure encryption – a new encryption primitive which we formally define and realise – and an evolving coins mechanism is required to achieve the same level of security. Intuitively, the reason is that we need to ensure that past coins received provide no significant advantage to the adversary when it corrupts an active stakeholder. We note that the naive approach of simply paying oneself with a new coin does not work here, as the same coin should be able to be elected multiple times in a sequence of PoS invocations without leaving any evidence in the ledger.

The work presented in this chapter is concurrent and independent, of another paper on privacy-preserving proof-of-stake by Ganesh et al. [GOT18]. Their work focuses on constructing a generic, privacy-preserving leadership election, given a list of commitments to each party's stake. This chapter by contrast focuses on ensuring the proof of stake leadership election can run with a provably secure, privacy-preserving transaction scheme. Notably, Zerocash cannot immediately be used with the system of [GOT18], as it does not maintain a list of stake commitments – indeed, such a list would appear to reveal more about the shift in funds than Zerocash does, such as how long an account has seen no changes.

5.1 Protocol Intuition

To begin with, we give a high-level sketch of the Crypsinous protocol in this section, to aid in understanding the more formal break-down of the protocol in Section 5.4, and to introduce core concepts. We will first sketch the design of two protocols we are building on – Ouroboros Genesis [BGK+18] and Zerocash [BCG+14]. We will discuss how these can be combined, and the issues that arise through this combination. Finally, we will sketch how we have resolved these issues.

5.1.1 The Foundations of Genesis and Zerocash

Ouroboros Genesis [BGK+18], divides time into discrete *slots*. At protocol start, parties are assigned an initial *stake* in the system. Typically, only the relative amount of such a stake is considered, that is, the fraction of the total stake owned. By protocol-external means, the distribution of this stake may shift over time, for instance, by users trading it amongst each other. In each slot, users have a probability proportional[1] to their relative stake to be "elected" as a *leader* of the slot. In practice, this relies on a pseudo-random value being below a user-specific target. Such leaders may then create a new block and sign it with a proof of leadership eligibility. In order to prevent so-called "grinding attacks", in which parties attempt the leadership election arbitrarily often with different accounts, transferring themselves the funds, Genesis divides time further into *epochs*. In each of these, the distribution of stake considered for leadership is fixed, and the pseudo-random values used to determine it can only be predicted once the epoch starts.

Zerocash [BCG+14] achieves complete transactional privacy in a distributed ledger setting through the use of non-interactive zero-knowledge (NIZK) proofs. It represents monetary value through *coins*, which can be created, and spent once. Crucially, it prevents double-spends and ensures value is preserved, while at the same time preventing the creation and spending of a coin from being linked. A transfer allows spending two coins and creating two new coins of the same combined value. This closely mirrors the simplest form of Bitcoin transactions. Each party holds a secret key used to spend coins, which is simply a random string, and its corresponding public key is a hash of the secret key. When creating a new coin, it is created *for a public key*. Specifically, a nonce is randomly selected for the new coin, and the transaction creating it commits to the coin's public key, nonce, and value. All such created commitments are kept in a protocol-wide Merkle tree. To spend a coin, a party makes a zero-knowledge proof of two things: First, the protocol-wide Merkle tree contains a commitment to it, and second, the spender knows the preimage of the public key. This by itself would allow double spends, so Zerocash reveals a coin's *serial number*, which is defined as a PRF of the secret key and the coin's nonce. The transfer finally proves in zero-knowledge that the transaction is zero-sum.

[1] Technically it is not linear, however this is a close approximation.

5.1.2 The Core Protocol

The core principle of Crypsinous is combining the strengths of both the Ouroboros Genesis and Zerocash protocols. While Ouroboros Genesis assumes the distribution of stake to be public, this fact is only used in verifying that leaders of a slot met the appropriate target – to remove this intrinsic leakage, we have parties hold Zerocash-style coins, with each coin being separately considered for leadership. As in Ouroboros Genesis, each coin is eligible to be a leader if a pseudorandom value meets some target. Instead of revealing the coin's value, however, in Crypsinous parties produce a NIZK proof of this, as well as proving that the respective coin is unspent[2]. This also forces us to explicitly model the transaction system by which stake is allowed to shift – as the stake distribution is no longer simply supplied to every party by the environment, it is necessary to make explicit how it is derived. For this reason, the core Crypsinous protocol includes a Zerocash-like transaction system.

5.1.3 Freezing Stake in Zero Knowledge

The security argument of Ouroboros Genesis relies on parties not being able to manipulate whether or not they won a leadership election. Specifically, it assumes the distribution of stakeholders to be fixed *before* the randomness for the same epoch is decided. Likewise, the set of coins that are eligible for a slot in the leadership election is fixed in Crypsinous. The protocol maintains this frozen set of coins, $\mathfrak{C}^{\mathsf{lead}}$, separately to the set of coins usable for spending, $\mathfrak{C}^{\mathsf{spend}}$. In practice, as coins are anonymously represented as sets of both commitments and serial numbers, and as any reuse of a serial number would lead to some privacy leakage, we represent them through two sets of commitments, $\mathbb{C}^{\mathsf{lead}}$ and $\mathbb{C}^{\mathsf{spend}}$, and one set of serial numbers, $\mathbb{S}$. In creating the leadership proofs, a coin's serial number is revealed. As it may later be spent, this would lead to some privacy leakage. To mitigate this, we instead *evolve* the coin in the leadership transaction. This new, evolved coin can then be spent and used in further leadership proofs, the latter being possible as it is derived deterministically from the former coin, which does not allow influencing the probability of it being elected in the

[2] More precisely, we atomically spend the coin to ourselves to prove this. An alternative would be to produce a non-membership proof [BLL00], which has a higher circuit cost, but is arguably simpler.

remainder of the epoch. We note that as this design inherently destroys the old coin, it is important that even leadership transactions of different branches of the chain are imported and validated.

5.1.4 Adaptive Corruptions

As Ouroboros Genesis is secure in the adaptive corruption model, it seems natural that privacy results should be possible in the same model. The construction described so far is not directly secure against adaptive corruptions. An adversary could, after corrupting a party, attempt to create leadership proofs of past slots with the newly corrupted party. Furthermore – in the UC framework – a non-committing encryption [Nie02] would be needed for the ciphertexts in the Zerocash style transactions, as with a committing encryption, the simulator would be unable to produce ciphertexts that stand up to inspection after corruption.

We solve the former issue by adding a cheap key-erasure scheme into the NIZK for leadership proofs. Specifically, parties have a Merkle tree of secret keys, the root of which is hashed to create the corresponding public key. The Merkle tree roots act like a Zerocash coin secret key and can be used to spend coins. For leadership however, parties also must prove knowledge of a path in the Merkle tree to a leaf at the index of the slot they are claiming to lead. After a slot passes, honest parties erase their preimages of this part of that path in the tree. As the size of this tree is linear with the number of slots, we allow parties to keep it small, by restricting its size. Keys therefore are associated with their creation time, by committing to this in the corresponding public key. While this does mean keys can expire, parties can trivially refresh them, and we sketch in Section 5.6 that this is a rare occurrence for practical parameters. We emphasise that parties *are* able to spend and refresh keys, even when expired.

While we could easily present Crypsinous using non-committing encryption, known realisations of this primitive are not efficient enough for this purpose in practice. Instead, we take advantage of our protocols network assumptions, which include an upper bound on message delivery, Δ_{max}. This allows us to utilise forward secure encryption instead of non-committing encryption, under the assumption that corruption is "delayed" by Δ_{max}. This delay is modelled by restricting adversarial access to the forward secure encryption secret key at time

t to the key for time $t + \Delta_{\text{max}}$.

5.2 Components of Crypsinous

In this section we discuss the main components of the real-world execution, including the hybrid functionalities that the protocol uses. We discuss the ideal world, and in particular the private transaction ledger functionality in Section 5.3. We provide all the aspects of the execution model from [BMTZ17, BGK+18] that are needed for our protocol and proof, but omit some of the low-level details and refer the more interested reader to these works wherever appropriate.

As in the case of Bitcoin (see [GKL15, PSs17, BGK+18]), our protocol is implicitly aware of an overestimate Δ_{max} of the actual (unknown) network delay Δ. However, this Δ_{max} is not used in the message passing; instead the protocol proceeds in an optimistic manner once messages are received (after at most Δ rounds from sending) and Δ_{max} is only used in the staking procedure to determine the leader(s) of each slot.

Our protocol makes use of the following hybrid functionalities, similarly to [BGK+18].

- The global clock functionality $\mathcal{G}_{\text{clock}}$.
- Broadcast channels $\mathcal{F}_{\text{Net}}^{\text{bc}}$ and $\mathcal{F}_{\text{Net}}^{\text{tx}}$ with the delay Δ.
- The genesis block generation and distribution functionality $\mathcal{F}_{\text{Init}}$, which captures the assumption that all parties (old and new) agree on the first, so-called *genesis* block. In fact, this functionality is slightly different from the one in [BGK+18] as the blocks in our work have a different structure to ensure privacy. Concretely, in Ouroboros Genesis this block includes the keys, signatures, and original stake distribution of the parties that are around at the beginning of the protocol. Here, for each stakeholder registered at the beginning of the protocol, $\mathcal{F}_{\text{Init}}$ records his keys and initial coin commitments in the genesis block; this block is distributed to anyone who requests it in any future round. As in [BGK+18] we assume without loss of generality that the global time is $t = 0$ in the genesis round. The new genesis block functionality is specified below.

- A random oracle $\mathcal{F}_{\mathsf{RO}}$ for abstracting hash function queries.

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

The functionality $\mathcal{F}_{\mathsf{Init}}$ is parameterised by the set of initial stakeholders $\mathcal{P} = \{\psi_1, \ldots, \psi_n\}$ and their respective stakes $S\colon \mathcal{P} \to \mathbb{R}^+$. It allows each of these parties to register keys, and provides them with openings to their generated coins.

State variables and initialisation values:

Variable	Description
$\mathfrak{G} \coloneqq \bot$	The genesis block
$\mathbb{C} \coloneqq \varnothing$	Mapping from parties to their committed coin

When receiving a message CLAIM *from a party ψ:*

if $\psi \notin \mathcal{P} \vee \psi \in \mathbb{C}$ **then return** REJECT

sample sk^{COIN} as CRYPSINOUS on GENERATE

let $\rho_{\mathbf{c}} \xleftarrow{*} \{0,1\}^\kappa; pk^{\text{COIN}} \leftarrow \mathsf{prf}^{\mathsf{pk}}_{\mathsf{root}_{sk^{\text{COIN}}}}(0)$

let $(\mathbb{C}(\psi), r_{\mathbf{c}}) = \mathsf{comm}(pk^{\text{COIN}} \parallel S(\psi) \parallel \rho_{\mathbf{c}})$

return $((pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, S(\psi)), sk^{\text{COIN}})$

When receiving a message GENESIS *from a party ψ:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

if $t = 0 \vee \exists \psi \in \mathcal{P}\colon \psi \notin \mathbb{C}$ **then abort**

if $\mathfrak{G} = \bot$ **then**

  let $\eta_1 \xleftarrow{*} \{0,1\}^\kappa; \mathfrak{G} \leftarrow (\{\, \mathbb{C}(\psi) \mid \psi \in \mathcal{P} \,\}, \eta_1)$

return $\mathfrak{G}$

To ensure privacy of transactions, we need to equip our model with a couple of extra functionalities not present in previous works. For instance, the (non-private) Ouroboros protocol-line [DGKR18, BGK+18] relies on verifiable random functions and key-evolving signatures to ensure security of the lottery which defines slot leaders and prevents double spending in the presence of an adaptive adversary.

In this work we cannot use signatures to authenticate coins/transactions as we need to keep the spent amount and the identities of the receiver private. For this reason we introduce *key-private forward secure encryption* and non-interactive zero-knowledge proofs (NIZKs). Our protocol will be described as having access to hybrid-functionalities for these primitives. To our knowledge no definition of

key-private forward secure encryption or an implementation thereof has been suggested. In fact, for reasons discussed in Subsection 5.2.3 an implementation of this primitive against fully adaptive adversaries might be impossible without additional setup assumptions. Instead, here we make an assumption about the (in)ability of the adversary to quickly read keys of newly corrupted parties and prove the security of our protocols under this assumption. Proving impossibility of the primitive against a fully adaptive adversary (or providing a protocol for it) is an interesting future direction.

Finally, our construction will make use of non-interactive equivocal commitments and pseudo-random functions (PRFs). Construction of both these primitives exists assuming a CRS under standard hardness assumption, for instance, hardness of the DDH (Decision Diffie Hellman) problem. Notably we require a stronger-than-typical form of PRFs, which we capture in Subsection 5.2.4.

5.2.1 Protocol Assumptions Encoded as a Wrapper

The security statements about implementation of ledgers are typically conditional. For instance, the Bitcoin ledger is proved secure assuming the majority of the system's hashing power is honest, and the Ouroboros (Genesis) ledger is implemented assuming the majority of the stake is held by honest parties. These assumptions can be easily described by explicitly restricting the class of environments and adversaries, but this would sacrifice the universal composability of the statement. We follow the paradigm of [BMTZ17] to capture these assumptions without compromising composability: Instead of explicitly restricting the adversary and environment, we introduce a functionality wrapper that wraps the functionalities that the protocol accesses and forces the required assumptions on the adversary/environment. We refer to [BMTZ17] for a more detailed discussion. The full wrapper is defined below; as this wrapper only becomes relevant for interpreting our main theorems (Theorem 5.2 and Theorem 5.3) it might be easier for the first-time reader to postpone parsing it until then.

Functionality $\mathcal{W}^{\mathsf{PoS}}_{\mathsf{HonMaj}}\left(\mathcal{F}^{\mathcal{R}_{\textsc{lead}}}_{\mathsf{NIZK}}, \mathcal{F}^{\mathcal{R}_{\textsc{xfer}}}_{\mathsf{NIZK}}, \mathcal{F}^{\mathsf{tx}}_{\mathsf{Net}}, \mathcal{F}^{\mathsf{bc}}_{\mathsf{Net}}, \mathcal{G}_{\mathsf{clock}}\right)$

The wrapper functionality is parameterised by the bound β on participating stake ratio, as defined in Ouroboros Genesis [BGK+18], and $\varepsilon > 0$, the parameter that describes the gap between the honest and adversarial stake. The wrapper is assumed to be registered with the global clock $\mathcal{G}_{\mathsf{clock}}$ and is aware of sets of registered parties

and the set of corrupted parties.

The wrapper makes checks about the distribution of stake. While this is trivial in Ouroboros Genesis, it is not immediately obvious that the wrapper knows this information in CRYPSINOUS. The wrapper observes all network traffic and all NIZK witnesses however, allowing it to reconstruct any party's view of the ledger. We do not describe this extraction in full detail – it is possible as the wrapper is around all "real-world" functionalities. We can therefore make assertions about the stake distribution despite the addition of privacy.

State variables and initialisation values:

Variable	Description
$\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$	The simulated leadership NIZK
$\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{XFER}}}$	The simulated transfer NIZK
$\mathcal{F}_{\text{Net}}^{\text{tx}}$	The simulated transaction network
$\mathcal{F}_{\text{Net}}^{\text{tx}}$	The simulated chain network
$\mathcal{G}_{\text{clock}}$	The simulated clock

When receiving a message (PROVE, x, w) *from a corrupted party* ψ *for* $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$*:*

let $\alpha \leftarrow$ fraction of honest stake participating in this round
if α is sufficient, as per [BGK+18] **then**
  simulate sending (PROVE, x, w) **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$ and **receive the reply** π
  return π
else
  return $\perp$

When receiving a message (PROVE, x, w) *from an honest party* ψ *for* $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$*:*

simulate sending (PROVE, x, w) **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$ and **receive the reply** π
return π

Forward but evesdrop all other requests to their simulated functionalities.

5.2.2 Non-Interactive Zero Knowledge

We utilise the Non-Interactive Zero Knowledge functionality $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ constructed composably in Chapter 3. The reference string for this could be created in a public proof-of-stake protocol, as suggested in Chapter 4, although this is not described in detail in this chapter.

NIZKs can be used for signature-like behaviour by embedding the messages that are to be signed in the statements of simulation-extractable NIZKs, con-

structing in this way a *signature of knowledge* (SoK) [GM17]. In particular, we note that witnesses used to generate proofs in Crypsinous will contain the party's secret key and the proved statement commits to the party's public key. As a result, the NIZK used in Crypsinous has similar unforgeability properties as standard signatures.

5.2.3 Key-Private Forward-Secure Encryption

To guarantee the forward-privacy of transactions, a forward-secure encryption scheme [CHK03] is necessary to hide information sent encrypted to a party's long-term encryption secret key. Traditional forward-secure encryption is insufficient, as it would leak information about the recipient of a transaction. To preserve the recipient's anonymity in Crypsinous transactions, we therefore require key-privacy as well [BBDP01]. Furthermore, as the simulator must create simulated ciphertexts, which it may later need to reveal the message of, encryption in the UC setting needs to be non-committing to withstand adaptive corruptions. Interestingly, however, there are no existing encryption schemes that simultaneously achieve key-privacy, forward-security, and the non-commitment property.

We overcome the above limitation by slightly weakening the above security requirements and only requiring forward-security with a time-sensitive non-committing property: Informally, only messages addressed to a time window of size Δ_{max} into the future are protected. As it turns out, this weaker notion is sufficient for our purposes. Even for this notion, however, it is not evident how to efficiently realise such an encryption in the UC setting. To understand the issue, it is useful to recall how we can realise non-interactive non-committing encryption via erasures. The idea is to have parties update their keys once the message is received. More concretely, a message is encrypted at round t and sent over to the receiver so that it can be decrypted with key $\mathsf{sk}_t^{\mathsf{enc}}$. Upon receiving it, the receiver can decrypt it (using $\mathsf{sk}_t^{\mathsf{enc}}$) and immediately update the key to $\mathsf{sk}_{t'}^{\mathsf{enc}}$ for the next round (and erase $\mathsf{sk}_t^{\mathsf{enc}}$). This way the link between the ciphertext and the key is eliminated by the time the adversary corrupts the receiver.

The above approach clearly fails if the channel has any delay, as in our setting, as this gives the adversary a window of opportunity of size Δ, and bounded only by Δ_{max}, to attack during which the message is already being transmitted but

has not yet been received by the recipient. This makes erasures useless in this window (if correctness is to be maintained).

To bypass the above issue, we make an assumption on the adversary's adaptiveness which, roughly, implies that the adversary cannot immediately access the secret key of a newly corrupted party. Specifically, we assume that the adversary corrupting a party with key $\mathsf{sk}_t^{\mathsf{enc}}$ at time t does not receive $\mathsf{sk}_t^{\mathsf{enc}}$, but rather the key $\mathsf{sk}_{t+\Delta_{\mathsf{max}}}^{\mathsf{enc}}$, which this party would hold in time $t+\Delta_{\mathsf{max}}$, if it were allowed to properly update its key. We emphasise that this is a milder assumption than that of delayed party-corruption which underlines the security of [KRDO17, BPS16]. Indeed, in these works the adversary is forbidden from accessing the entire state of a corrupted party for a certain number of rounds after corruption; instead, here we only restrict his access to the present keys, and we even give the adversary an outlook, already upon corruption, of how the key will look in the near future.

To enforce the above restriction without affecting the universal composability of our statements, we use a technical trick inspired by [BMTZ17, DGHM13]: We introduce an ideal functionality which captures this restriction/assumption. This functionality, denoted by $\mathcal{F}_{\mathsf{KeyMem}}$, stores keys upon request from parties, and updates them every round using a one-way function Update; when an honest party requests a key it has submitted in the past, the functionality sends it the current key. However, when the adversary asks for a key (on behalf of a corrupted party) $\mathcal{F}_{\mathsf{KeyMem}}$ first applies Update Δ_{max} times and returns the updated key to the adversary.

As an added bonus from using the above functionality-based approach for restricting the adversary, our treatment ensures that the restriction is localised to the encryption functionality; thus, if someone comes up with an instantiation of the encryption functionality against a fully adaptive adversary, our protocol would immediately be secure against such an adversary. The $\mathcal{F}_{\mathsf{KeyMem}}$ functionality is specified below.

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

$\mathcal{F}_{\mathsf{KeyMem}}$ is parameterised by its corruption delay Δ_{max} and a memory update function update. We write $\mathsf{update}^{\Delta_{\mathsf{max}}}$ to mean "apply update Δ_{max} times."

State variables and initialisation values:

Variable	Description
$M_\psi \coloneqq \bot$	Memory for each party ψ

When receiving a message $(\textsc{init}, M_0)$ *from a party* ψ*:*

assert $M_\psi = \bot$

let $M_\psi \leftarrow M_0$

When receiving a message GET *from a party* ψ*:*

if $\psi \in \mathcal{H}$ **then return** M_ψ

else return $\mathsf{update}^{\Delta_{\mathsf{max}}}(M_\psi)$

When receiving a message UPDATE *from a party* ψ*:*

if $M_\psi \neq \bot$ **then let** $M_\psi \leftarrow \mathsf{update}(M_\psi)$

The UC functionality for key-private and forward-secure encryption, $\mathcal{F}_{\mathsf{FwEnc}}$ and the accompanying construction, are described in detail below.

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

$\mathcal{F}_{\mathsf{FwEnc}}$ is parameterised by a security parameter κ, a set of parties $\mathcal{P}$, and a maximum delay Δ_{max}.

State variables and initialisation values:

Variable	Description
$K \coloneqq \varnothing$	Mapping from parties to public keys
$T \coloneqq \varnothing$	Mapping from parties to earliest decryptable time slot
$M \coloneqq \varnothing$	Mapping from ciphertexts to their recipient, latest decryption time, and message

When receiving a message KEYGEN *from a party* ψ*:*

assert $\psi \notin K$

query $\mathcal{A}$ **with** $(\textsc{keygen}, \psi)$ and **receive the reply** pk,

satisfying $\nexists \psi' \colon K(\psi') = \mathsf{pk}$, else **sampling from** $\{0,1\}^\kappa$

let $K(\psi) \leftarrow \mathsf{pk}; T(\psi) \leftarrow 0$

return pk

When receiving a message $(\textsc{encrypt}, \mathsf{pk}, t, m)$ *from a party* ψ*:*

if $\exists \psi' \colon K(\psi') = \mathsf{pk} \wedge \psi' \in \mathcal{H} \wedge t < T(\psi') + \Delta_{\mathsf{max}}$ **then**

query $\mathcal{A}$ **with** $(\textsc{encrypt}, t, |m|)$ and **receive the reply** c,

satisfying $c \notin M$, else **sampling from** $\{0,1\}^\kappa$

else

query $\mathcal{A}$ **with** (LEAK-ENCRYPT, pk, t, m) and **receive the reply** c,

satisfying $c \notin M$, else **sampling from** $\{0,1\}^\kappa$

let $M(c) \leftarrow (\mathsf{pk}, m, t)$

When receiving a message (DECRYPT, t, c) *from a party* ψ*:*

if $\psi \in \mathcal{H}$ **then let** $\delta \leftarrow 0$

else let $\delta \leftarrow \Delta_{\max}$

if $t < T(\psi) + \delta \vee \psi \notin K$ **then return** FAIL

if $c \in M$ **then**

let $(\mathsf{pk}, m, t') \leftarrow M(c)$

if $\mathsf{pk} \neq K(\psi) \vee t \neq t'$ **then return** FAIL

else return (OK, m)

else

query $\mathcal{A}$ **with** (DECRYPT, $K(\psi)$, t, c) and **receive the reply** m

return m

When receiving a message UPDATE *from a party* ψ*:*

query $\mathcal{A}$ **with** (UPDATE, ψ)

let $T(\psi) \leftarrow T(\psi) + 1$

We extend the notion of forward-secure encryption (FSE) with a notion of *key privacy*, described in detail in Definition 5.1 below. While this definition itself is novel, it is possible to combine existing schemes to satisfy it. In particular, [CHK03] constructs FSE from hierarchical identity-based encryption (HIBE). Their scheme, paired with the anonymous HIBE construction of [BW06] satisfies our requirements of key-privacy as we will argue below.

For the argument of key privacy, the FSE from HIBE construction in [CHK03] is straightforward, with the ciphertexts simply being the underlying HIBE scheme's ciphertexts. The core argument of the anonymity of [BW06] is the indistinguishability of ciphertexts from random group elements – and therefore their independence of the encrypting identity [BW06, Lemmas 8 & 9]. We note that the ciphertexts' pseudo-randomness implies a stronger notion than just anonymity – the ciphertext also does not reveal any information about the HIBE public key. In particular, as ciphertexts are indistinguishable, our enhanced security game given in Definition 5.1 is satisfied.

This construction's time and space complexity is logarithmic in the number

of time slots. As the number of slots is by necessity less than 2^κ, the use of this forward-secure encryption has a linear increase in cost with respect to the security parameter compared to standard encryption.

5.2.3.1 Key-Private Forward-Security Against Chosen Ciphertext Attacks

Definition 5.1. A key-evolving public-key encryption scheme $\mathcal{S} = (\mathsf{Upd}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is called *key-privately forward-secure against chosen ciphertext attacks (kp-fs-CCA)* if any PPT adversary has only negligible advantage in the kp-fs-CCA game, defined in Game 5.1, for any set of parties $\mathcal{P}$ and update bound N.

Game 5.1. *The adversary wins the kp-fs-CCA game if it can distinguish two arbitrary ciphertexts. The adversary may choose which parties they are encrypted for, and is given access to a decryption and corruption oracle which permits corrupting any party, including parties challenges have been issued for, as long as the corruption is requested for an updated key.*

Setup: *For each party $\psi \in \mathcal{P}$, sample $(\mathsf{pk}_\psi, \mathsf{sk}_\psi^0) \xleftarrow{*} \mathsf{Gen}(1^\kappa, N)$. The adversary receives all public keys pk_ψ. Furthermore, a bit $b \xleftarrow{*} \{0,1\}$ is selected, but not revealed to the adversary.*

Attack: *The adversary issues multiple* $\mathsf{challenge}(j, (\psi_0, m_0), (\psi_1, m_1))$ *queries, multiple* $\mathsf{corrupt}(i, \psi)$ *queries and multiple* $\mathsf{decrypt}(k, c, \psi)$ *queries, where $\psi, \psi_0, \psi_1 \in \mathcal{P}$ and $0 \leq i \leq N; 0 \leq j \leq N; k \leq N$. Furthermore, if a* corrupt *query is made for some party for which a challenge query is also made for, then the corresponding i must be greater than the corresponding j.* corrupt *queries may be issued only once for each party.*

- $\mathsf{corrupt}(i, \psi)$ *is answered with* $\mathsf{sk}_p^i \coloneqq \mathsf{Upd}(\ldots \mathsf{Upd}(\mathsf{sk}_p^0, 1), \ldots, i)$.
- $\mathsf{challenge}(j, (\psi_0, m_0), (\psi_1, m_1))$ *is answered by responding with* $c = \mathsf{Enc}_{\mathsf{pk}_{\psi_b}}(j, m_b)$, *and (j, c, ψ_0) and (j, c, ψ_1) are recorded as challenges. m_0 and m_1 must be of the same length.*
- $\mathsf{decrypt}(k, c, \psi)$ *is answered with $\perp$ if (k, c, ψ) is recorded as a challenge. Otherwise, it is answered with* $\mathsf{Dec}_{\mathsf{sk}_p^k}(k, c^*)$.

Guess: *The adversary outputs a guess $b' \in \{0,1\}$, and wins the game iff $b' = b$.*

For completeness, correctness is defined as usual:

Definition 5.2. A key-evolving public-key encryption scheme $\mathcal{S} = (\mathsf{Upd}, \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is *perfectly correct*, if for any message m, time bound N and time $t \leq N$:

$$\Pr[(\mathsf{pk}, \mathsf{sk}^0) \xleftarrow{*} \mathsf{Gen}(1^\kappa, N); \mathsf{Dec}_{sk^t}(t, \mathsf{Enc}_{\mathsf{pk}}(t, m)) = m] = 1,$$

where $sk^t \coloneqq \mathsf{Upd}(\ldots \mathsf{Upd}(\mathsf{sk}^0, 1), \ldots, t)$.

5.2.3.2 Lifting to a UC-Protocol

A kp-fs-CCA-secure key-evolving encryption scheme induces the following protocol for realising $\mathcal{F}_{\mathsf{FwEnc}}$ in the $\mathcal{F}_{\mathsf{KeyMem}}$-hybrid model:

Protocol FwEnc

FwEnc is parameterised by the corruption delay Δ_{max}, a time bound N, the underlying FSE scheme $\mathcal{S}$. It operates in the $\mathcal{F}_{\mathsf{KeyMem}}$-hybrid world, where $\mathcal{F}_{\mathsf{KeyMem}}$ is parameterised by Δ_{max} and the following Update function:

function Update((sk, t))
  return (Upd(sk, $t+1$), $t+1$)

State variables and initialisation values:

Variable	Description
$t \coloneqq \bot$	Earliest time which is decryptable

When receiving a message KEYGEN *from a party ψ:*

assert $t = \bot$
let $(\mathsf{pk}, \mathsf{sk}^0) \xleftarrow{*} \mathsf{Gen}(1^\kappa, N)$
send (INIT, $(\mathsf{sk}^0, 0)$) **to** $\mathcal{F}_{\mathsf{KeyMem}}$
erase sk^0
let $t \leftarrow 0$
return pk

When receiving a message (ENCRYPT, pk, t', m) *from a party ψ:*

return $\mathsf{Enc}_{\mathsf{pk}}(t', m)$

When receiving a message (DECRYPT, t', c) *from a party ψ:*

if $t' < t$ **then return** $\bot$
send GET **to** $\mathcal{F}_{\mathsf{KeyMem}}$ and **receive the reply** (sk, $\cdot$)
let $t'' \leftarrow t$
while $t'' < t'$ **do**
  let $t'' \leftarrow t'' + 1$; $\mathsf{sk} \leftarrow \mathsf{Upd}(\mathsf{sk}, t'')$
let $m \leftarrow \mathsf{Dec}_{\mathsf{sk}}(t', c)$
erase sk
return m

When receiving a message UPDATE *from a party ψ:*

let $t \leftarrow t + 1$

send UPDATE **to** $\mathcal{F}_{\text{KeyMem}}$

return $\top$

5.2.3.3 The Simulator

We now present the simulator for which we will show UC emulation.

Simulator $\mathcal{S}_{\text{FwEnc}}$

In addition to responding to $\mathcal{F}_{\text{FwEnc}}$, the simulator $\mathcal{S}_{\text{FwEnc}}$ maintains a simulated $\mathcal{F}_{\text{KeyMem}}$, through which it provides the adversary with (delayed) access to secret keys.

State variables and initialisation values:

Variable	Description
$K \coloneqq \varnothing$	Mapping from parties to their key pairs
$\mathcal{F}_{\text{KeyMem}}$	Simulated key memory

When receiving a message (KEYGEN, ψ) *from* $\mathcal{F}_{\text{FwEnc}}$*:*

let $K(\psi) \xleftarrow{*} \text{Gen}(1^\kappa, N)$; $(\text{pk}, \text{sk}) \leftarrow K(\psi)$; $T(\psi) \leftarrow 0$

simulate sending (INIT, (sk, 0)) **to** $\mathcal{F}_{\text{KeyMem}}$ **on behalf of** ψ

return pk

When receiving a message (ENCRYPT, t, l) *from* $\mathcal{F}_{\text{FwEnc}}$*:*

let $m \leftarrow 0^l$; $(\text{pk}, \cdot) \xleftarrow{*} \text{Gen}(1^\kappa, N)$

let $c \xleftarrow{*} \text{Enc}_{\text{pk}}(t, m)$

return c

When receiving a message (LEAK-ENCRYPT, pk, t, m) *from* $\mathcal{F}_{\text{FwEnc}}$*:*

return $\text{Enc}_{\text{pk}}(t, m)$

When receiving a message (DECRYPT, pk, t, c) *from* $\mathcal{F}_{\text{FwEnc}}$*:*

if $\exists \psi, \text{sk}: K(\psi) = (\text{sk}, \text{pk})$ **then**

  let $t' \leftarrow 0$

  while $t' < t$ **do**

    let $t' \leftarrow t' + 1$; $\text{sk} \leftarrow \text{Upd}(\text{sk}, t')$

  return $\text{Dec}_{\text{sk}}(t, c)$

else

  return $\perp$

When receiving a message (UPDATE, ψ) *from a party* ψ*:*

simulate sending UPDATE **to** $\mathcal{F}_{\mathsf{KeyMem}}$ **on behalf of** ψ

On receiving messages to $\mathcal{F}_{\mathsf{KeyMem}}$ *from* $\mathcal{A}$*:* Forward these messages to the simulated $\mathcal{F}_{\mathsf{KeyMem}}$.

5.2.3.4 UC Emulation

Theorem 5.1. *If the underlying key-evolving PKE scheme is kp-fs-CCA secure then* FwEnc *UC-emulates* $\mathcal{F}_{\mathsf{FwEnc}}$ *in the* $\mathcal{F}_{\mathsf{KeyMem}}$*-hybrid world.*

Proof. The points in which the simulator $\mathcal{S}_{\mathsf{FwEnc}}$, combined with $\mathcal{F}_{\mathsf{FwEnc}}$, can behave differently from FwEnc are in how they respond to various queries and the internal state they maintain. We will use t_ψ as ψ's time in both worlds – t in ψ's state in the real world and $T(\psi)$ in the ideal world. FwEnc maintains a public/private key pair for each party, which the simulator selects from exactly the same distribution and both return the public key, while storing sk_p^0. Furthermore, both initialise t_ψ to zero. As a result, for KEYGEN-queries, the simulation is perfect. For UPDATE, while the simulator does not call Upd on the secret key, this is merely because the call is deferred to the point where it is used, in DECRYPT. In both worlds however, t_ψ is updating the same way and matches the ideal functionality's t_ψ value.

What remains is showing the correctness of encryption, decryption, and corruption queries. We will reduce this to kp-fs-CCA security, by showing that if the environment can distinguish, we can extract a kp-fs-CCA adversary with black-box access to the distinguishing environment, which wins the kp-fs-CCA game with a non-negligible advantage. In both the real and ideal worlds, the public and secret keys for $\psi_1, \ldots, \psi_n$ are sampled from $\mathsf{Gen}(1^\kappa, N)$ – with in the real-world parties holding their own keys and, in the ideal world, the simulator holding all. We note that while the dummy key $\mathsf{pk}_{\mathsf{dummy}}$ exists only in the ideal world and its corresponding secret key is never used, we can assume it also exists in the real world, however remains entirely unused. Therefore as all (not adversarially generated) key pairs are sampled the same in both worlds, we can extract this sampling from the UC security definition – if all key pairs $(\mathsf{pk}_1, \mathsf{sk}_1), \ldots, (\mathsf{pk}_n, \mathsf{sk}_n), (\mathsf{pk}_{\mathsf{dummy}}, \mathsf{sk}_{\mathsf{dummy}})$ are sampled from the same distribution and fixed in both the real and ideal executions, the real and ideal distributions are indistinguishable with overwhelming probability. Given an envi-

ronment $\mathcal{Z}$ which can distinguish between the real and ideal world with non-negligible advantage, we can therefore assume that it can distinguish between the real and ideal world, with fixed keys, with a non-negligible advantage. We use $\mathcal{Z}$ to construct an adversary $\mathcal{A}$ for the kp-fs-CCA game and prove that $\mathcal{A}$ has a non-negligible advantage. Specifically, $\mathcal{A}$ simulates running $\mathcal{Z}$ against the ideal world, with the following modifications:

- The public/secret key pairs used by the simulator are supplied by $\mathcal{A}$ by programming the random tape.
- $\mathcal{A}$ monitors all messages sent in the simulation, in particular messages to the ideal functionality from all parties.
- Since $\mathcal{A}$ does *not* hold any party's secret keys, on a DECRYPT query to the simulator, it posts a $\mathsf{decrypt}(t, c, \psi)$ query and returns the response.
- We note secret keys are only used for decryption, as well as being handed to the (UC) adversary upon corruption. When the simulator hands the keys to the (UC) adversary, the (kp-fs-CCA) adversary posts a $\mathsf{corrupt}(t_\psi + \Delta_{\mathsf{max}}, \psi)$ query to obtain $\mathsf{sk}_p^{t_\psi+\Delta_{\mathsf{max}}}$. While $\mathcal{F}_{\mathsf{KeyMem}}$ at the time of corruption stores $\mathsf{sk}_p^{t_\psi}$, by assumption it will first apply Δ_{max} updates.
- When the ideal functionality receives an $(\textsc{encrypt}, \mathsf{pk}_p, t, m)$ query, if it does not reveal m to the simulator, $\mathcal{A}$ queries $\mathsf{challenge}(t, (\psi_{\mathsf{dummy}}, 0^{|m|}), (\psi, m))$ and returns c.

We begin by observing that this adversary does obey the rules of the kp-fs-CCA game. Specifically, the conditions for the game are as follows: a) A challenge ciphertext is not queried for decryption, and b) A party is not challenged after it has been corrupted. For a) challenge queries are performed when an ENCRYPT message is seen and due to the structure of $\mathcal{F}_{\mathsf{FwEnc}}$, the challenges will be issued for, at latest, the time $t_\psi + \Delta_{\mathsf{max}} - 1$. On corruption, the $\mathsf{corrupt}(k, \psi)$ query is made with $k = t_\psi + \Delta_{\mathsf{max}}$. As t_ψ is monotonically increasing and ENCRYPT is not called after corruption – and therefore no further challenge queries are issued – the corruption can occur only after all challenges. For b), we note that on corruption, $\mathcal{F}_{\mathsf{FwEnc}}$ will no longer query the simulator with ENCRYPT queries for this party, but only with LEAK-ENCRYPT queries. As challenge queries are

only issued on ENCRYPT queries, this party will no longer receive challenge queries.

Next, if $b = 0$, the execution perfectly matches a random ideal world execution with $\mathcal{S}_{\mathsf{FwEnc}}$. Specifically, if $b = 0$ the result of $\mathsf{challenge}(t, (\psi_{\mathsf{dummy}}, 0^{|m|}), (\psi, m))$ is $\mathsf{Enc}_{\mathsf{pk}_{\mathsf{dummy}}}(t, 0^{|m|})$. Furthermore, $\mathsf{decrypt}(t, c, \psi) = \mathsf{Dec}_{\mathsf{sk}_p^t}(t, c)$, that is, all points in which $\mathcal{A}$ intervenes in the UC execution, the execution is identical for $b = 0$.

Finally, we will argue that if $b = 1$, the statistical distance between the simulated UC execution and the UC execution of FWENC is negligible. Honest parties perform four operations in FWENC: A one-time key-generation, encryption, decryption, and update. The keys are supplied in kp-fs-CCA, and sampled from the same distribution as in the protocol. Initially, t is set to 0 for ψ upon key generation in both the protocol and the simulator. In both cases, pk is returned, sampled from the Gen algorithm. For encryption, regardless of whether Encrypt or DummyEncrypt is called by the functionality, as $\mathsf{challenge}(t, (\psi_{\mathsf{dummy}}, 0^{|m|}), (\psi, m)) = \mathsf{enc}_{\mathsf{pk}_p}(t, m)$, the ciphertext will be sampled from $\mathsf{enc}_{\mathsf{pk}_p}(t, m)$, the same distribution used in the protocol. For decryption queries, if it lies in the past, both the protocol and functionality will return $\perp$. The functionality will, if it supplied the ciphertext itself and the party is the intended recipient, return the corresponding plaintext. Otherwise it asks the simulator for decryption, which in turn makes a decrypt query. We note that by contrast, the protocol will *always* run $\mathsf{Dec}_{\mathsf{sk}_p^t}(t, c)$. If a decrypt query is made, we know that – since the ciphertext was not previously challenged (at least not with the same party and time slot) – the behaviour is identical. Otherwise, we know by the correctness of the underlying key-evolving encryption scheme that with overwhelming probability the decryption must return the same plaintext. For update, t_ψ is kept the same in the protocol and the simulated execution by incrementing it. While the secret key is not updated in the simulated execution, this update serves only to erase information – something the simulator does not care about. □

5.2.4 PRFs with Unpredictability Under Malicious Keys

Consider a PRF family $\{f_k\}_{k \in K}$ such that $f_k \colon X \to Y$ for all $k \in K$. The usual PRF security requires that any PPT distinguisher $\mathcal{D}$ with an oracle cannot tell the

difference between an oracle $f_k(\cdot)$ for a randomly selected k and a truly random function over $X \to Y$. The definition can be ported to the random oracle setting where both the function f_k as well as the distinguisher D have access to a random oracle $H(\cdot)$. Unpredictability under malicious key generation is an additional property that, intuitively, suggests the function does not have any "bad keys" that can eliminate the entropy of the input, a concept introduced in [DGKR18]. In the random oracle model, the property can be expressed as follows: for any PPT $\mathcal{A}$ and $x \in X, T \in \mathbb{N}$, the probability of the event $\Pr[f_k(x) < T \mid x \notin Q_H]$ equals $T/2^\kappa$ where $\mathcal{A}(1^\kappa) = k$ and Q_H is the set of queries of $\mathcal{A}$ to H.

We will employ the following construction. Let $H\colon \{0,1\}^* \to \mathbb{G}$ be a function mapping to a cyclic group $\mathbb{G}$ generated by g with a compact representation. We use an elliptic curve group based on the "elligator" curves [BHKL13] that have the property that a uniform element over $\mathbb{G}$ is indistinguishable from a random κ-bit string. We then define $f_k(m) \coloneqq H(m)^k$ for $k \neq 0$ and we show that it is a PRF with unpredictability under malicious key generation from X to $\{0,1\}^\kappa$. Indeed observe first that $(g^k, H(m), H(m)^k)$ is a DDH triple over the group $\mathbb{G}$. Thus, by the DDH assumption and the random oracle model, we can substitute all queries to the PRF by random group elements. Now observe that by the encoding properties of the curve these elements can be substituted by random strings over $\{0,1\}^\kappa$. Regarding the unpredictability under malicious key generation observe that in the random oracle model, $\Pr[H(x)^k < T] \leq \sum_{y<T} \Pr[H(x)^k = y] = T \cdot \Pr[H(x) = y^{1/k}] \leq T/2^\kappa$ in the conditional space $x \notin Q_H$.

5.2.5 Equivocal Commitments

We make use of a non-interactive equivocal commitment scheme [DG03], which is secure in the CRS model assuming hardness of discrete logarithms. For self-containment we include a high-level description, including some notation used in our proofs below.

Specifically, we will assume the existence of six algorithms, $\mathsf{init}_{\mathsf{comm}}$, comm, deComm, $\mathsf{simInit}_{\mathsf{comm}}$, simComm, and equiv. $\mathsf{init}_{\mathsf{comm}}$ generates a public key $\mathsf{pk}^{\mathsf{comm}}$ which is given as an argument to comm and deComm and will be part of the parameterisation of the CRS functionality. In addition to satisfying the traditional commitment properties of binding, hiding, and correctness, the scheme also satisfies equivocality. Specifically, $\mathsf{simInit}_{\mathsf{comm}}$ provides an equiv-

ocation key in addition to $\mathsf{pk}^{\mathsf{comm}}$. This equivocation key "breaks" the binding property – simComm can generate a commitment without a message and equiv can later create a witness matching any message for this commitment. We note that we do not require additional common properties, such as extraction or non-malleability, as these are provided by other components of Crypsinous' design, in particular the NIZK functionality.

We write $(\mathsf{cm}, r) \leftarrow \mathsf{comm}(m)$ to create the commitment cm for message m, and $\mathsf{deComm}(\mathsf{cm}, m, r) = \top$ if the decommitment to m and r verifies. Likewise, we write $\mathsf{cm} \leftarrow \mathsf{simComm}(\mathsf{ek})$ for simulating a commitment with equivocation key ek and $r \leftarrow \mathsf{equiv}(\mathsf{ek}, \mathsf{cm}, m)$ to equivocate, where $\mathsf{deComm}(\mathsf{cm}, m, r) = \top$. In all these, we leave the public key $\mathsf{pk}^{\mathsf{comm}}$ implicit, as it is assumed to be globally known via the CRS.

5.3 The Private Ledger

We next provide the complete description of the private ledger functionality that, as we prove, is implemented by Crypsinous. Privacy of Crypsinous is captured by the transactions which are returned from the functionality being *blinded* by a function blind. Crypsinous transactions are represented as a tuple of subtransactions, denoted $\mathsf{tx} = (\mathsf{stx}_1, \ldots, \mathsf{stx}_\ell)$. Each subtransaction is a pair of recipient ID and message, (id, m), where IDs usually correspond to real-world public keys. These IDs are generated by the private ledger functionality, allowing it to show the message on to the party which generated the corresponding ID, or to the adversary in case no honest party generated it. An exception is the symbol PUBLIC, which is used to denote subtransactions visible to all parties.

5.3.1 UC Specification

The private ledger behaves similarly to $\mathcal{G}^{\delta}_{\mathsf{DelayLedger}}$, but also tracks party registration (in order to notice disconnections and to exclude these from liveness), all READ requests are passed through blind. It is additionally parameterised by a stateful leakage procedure Lkg, which provides the simulator with information about who wins a simulated leadership election.

Functionality $\mathcal{G}_{\text{pLedger}}$

$\mathcal{G}_{\text{pLedger}}$ is a modification of $\mathcal{G}^{\delta}_{\text{DelayLedger}}$ which supports partially hiding transactions (which are divided into addressed *subtransactions*), a "leakage" function, the output of which is given to the adversary on request. It also supports ID generation, which are used to address subtransactions. It is parameterised by a mapping S_0 from parties to their initial stake.

For computing coin spendability, the mapping of ledger states M retains historical data, which is accessed through subscript: M_t refers to the state of M at the end of time t. This is used to ensure parties saw a transaction in their ledger state at the time they spent a coin originating in it.

State variables and initialisation values:

Variable	Description
$\Sigma \coloneqq \varepsilon$	Authoritative ledger state
$M \coloneqq \lambda\psi.\varepsilon$	Mapping of parties to ledger states
$U \coloneqq \varnothing$	Multiset of unconfirmed transactions
$\mathsf{Id} \coloneqq \varnothing$	Mapping from parties to tag, id pairs
$\mathsf{Reg} \coloneqq \varnothing$	Tracks registration of parties
$\mathfrak{C}_0 \coloneqq \varnothing$	Initial set of party ID, coin ID, and value

When receiving a message MAINTAIN-LEDGER *from a party ψ:*

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t
let $\mathsf{Reg} \leftarrow \mathsf{Reg} \cup (\psi, t)$
if $t = 0$ **then**
  let $\mathsf{coinId} \leftarrow (\text{GENERATE}, \text{COIN})$
  let $\mathfrak{C} \leftarrow \mathfrak{C} \cup \{(\mathsf{coinId}, S_0(\psi))\}$
  return $(\mathsf{coinId}, S_0(\psi))$
query $\mathcal{A}$ **with** (MAINT, ψ)

When receiving a message $(\text{SUBMIT}, \mathsf{tx})$ *from a party ψ:*

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t
assert $t > 0 \wedge (\psi, t) \in \mathsf{Reg}$
query $\mathcal{A}$ **with** $(\text{TRANSACTION}, \mathsf{blind}_{\mathcal{A}}(\mathcal{H}, \mathsf{Id}, (\bot, \mathsf{tx}, t, \psi)))$ and **receive the reply** txid,
  satisfying $(\mathsf{txid}, \cdot, \cdot) \notin \Sigma \cup U$, else **sampling from** $\{0,1\}^{\kappa}$
let $U \leftarrow U \cup \{(\mathsf{txid}, \mathsf{tx}, t, \psi)\}$

When receiving a message $(\text{GENERATE}, \mathsf{tag})$ *from a party ψ:*

query $\mathcal{A}$ **with** (GENERATE, ψ, tag) and **receive the reply** id,
satisfying $\nexists \psi' : (\cdot, \mathsf{id}) \in \mathsf{Id}(\psi')$, else **sampling from** $\{0,1\}^\kappa$
let $\mathsf{Id}(\psi) \leftarrow \mathsf{Id}(\psi) \cup \{(\mathsf{tag}, \mathsf{id})\}$
return id

When receiving a message READ *from a party* ψ*:*

assert liveness
if $\psi = \mathcal{A}$ **then**
send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
let lkgs $\leftarrow \{\ \mathsf{Lkg}(\psi', M(\psi'), \mathsf{Id}, \mathsf{Reg}, M, \mathfrak{C}_0, t) \mid \psi' \in \mathsf{online} \cap \mathcal{H}\ \}$
return $(\mathsf{map}(\mathsf{blind}_{\mathcal{A}}(\mathcal{H}, \mathsf{Id}), \Sigma), \mathsf{map}(\mathsf{blind}_{\mathcal{A}}(\mathcal{H}, \mathsf{Id}), U), \mathsf{lkgs})$
else
return $\mathsf{map}(\mathsf{blind}(\{\psi\}, \mathsf{Id}), M(\psi))$
return $\mathsf{map}(\mathsf{proj}_1, M(\psi))$

When receiving a message (EXTEND, Σ') *from* $\mathcal{A}$*:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
for txid **in** Σ' **do**
assert $\exists u, t', \psi : (\mathsf{txid}, u, t, \psi) \in U$
let $U \leftarrow U \setminus \{\mathsf{txid}, u, t', \psi\}$
let $\Sigma \leftarrow \Sigma \parallel (\mathsf{txid}, u, t, \psi)$

When receiving a message (ADVANCE, ψ, Σ') *from* $\mathcal{A}$*:*

if $\mathsf{map}(\mathsf{proj}_1, M(\psi)) \prec \Sigma' \prec \mathsf{map}(\mathsf{proj}_1, \Sigma)$ **then**
let $M(\psi) \leftarrow \mathsf{take}(\Sigma, |\Sigma'|)$.

Helper procedures:

procedure liveness
send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
if $\exists (\cdot, \cdot, t', \cdot) \in U : |t - t'| > \delta$ **then**
return $\bot$
else if $\exists (\mathsf{txid}, \mathsf{tx}, t', \psi) \in \Sigma : |t - t'| > \delta \wedge \exists \psi' \in \mathcal{H} : (\mathsf{txid}, \mathsf{tx}, t', \psi) \notin M(\psi')$ **then**
return $\bot$
else
return $\top$
procedure online
send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
return $\{\ p \mid p \in \mathcal{P}, \forall t' \in \mathbb{N} : t - \mathsf{onlineDelay} - 1 \leq t' < t \implies (p, t') \in \mathsf{Reg}\ \}$

function blind($\mathcal{P}$, Id, $(\cdot, \mathsf{tx}, \cdot, \cdot)$)
  let $\mathsf{tx}' \leftarrow \varepsilon$
  for (pk, m) **in** tx **do**
    if $\mathsf{pk} = \textsc{public} \vee \exists \psi \in \mathcal{P} \colon (\textsc{id}, \mathsf{pk}) \in \mathsf{Id}(\psi)$ **then**
      let $\mathsf{tx}' \leftarrow \mathsf{tx}' \parallel (\mathsf{pk}, m)$
    else
      let $\mathsf{tx}' \leftarrow \mathsf{tx}' \parallel (\bot, |m|)$
  return tx'
function $\mathsf{blind}_{\mathcal{A}}(\mathcal{H}, \mathsf{Id}, (\mathsf{txid}, \mathsf{tx}, t, \psi))$
  let $\mathsf{tx}' \leftarrow \varepsilon$
  for (pk, m) **in** tx **do**
    if $\mathsf{pk} \neq \textsc{public} \wedge \exists \psi \in \mathcal{H} \colon (\textsc{id}, \mathsf{pk}) \in \mathsf{Id}(\psi)$ **then**
      let $\mathsf{tx}' \leftarrow \mathsf{tx}' \parallel (\bot, |m|)$
    else
      let $\mathsf{tx}' \leftarrow \mathsf{tx}' \parallel (\mathsf{pk}, m)$
  return $(\mathsf{txid}, \mathsf{tx}', t, \psi)$

5.3.2 Leakage for Leader-Based Protocols

In our system, we permit the leakage $\mathsf{lkg}_{\mathsf{lead}}$, which effectively simulates the protocols leadership election and leaks the winning party. Specifically, for each time t, the adversary receives a set of parties that won the leadership election. This set is selected by sampling a random coin for each party, weighted by their stake using the same algorithm as in Ouroboros Praos [DGKR18]. We note that while this leakage is protocol-specific, it follows a general principle of leaking the elected leaders in a protocol. Specifically, honest parties will be selected by $\mathsf{lkg}_{\mathsf{lead}}$ with the probability of them winning a leadership election in Crypsinous. This probability is the same as in Ouroboros Genesis, and is the function ϕ_f of their stake, where ϕ_f is the independent aggregation function described in [DGKR18, BGK+18].

In addition to this, we note Zerocash-style protocols will allow an adaptively corrupting adversary to compute the serial number of coins *it sent* to an honest party after corrupting them. As the serial number is by necessity committing, the simulator must know when such adversarially sent coins are spent, to ensure the consistency of the simulation. For this reason, we also leak the points at

which adversarially sent coins are spent.

To formally capture both of these, defining how coins are distributed in the ideal world for any given ledger state is required. We capture this through a function $\mathfrak{C}_{\mathsf{ideal}}$, which takes the ledger state Σ, the ID mapping id, and the initial coins $\mathfrak{C}_0$ as inputs, returning a triple of spendable coins, spendable coins seen and owned by honest parties, and transactions spending adversarially created coins. Formally, the ideal-world semantics are described in Subsection 5.3.3.

For the full leakage, we assume t_i^{epStart} is the time ep_i starts, t_i^{epFreeze} is the time before which the stake for ep_i is frozen, and $ep(t)$ is the epoch time t lies in. We write $\Sigma^{<t}$ for $\mathsf{filter}(\lambda(\cdot,\cdot,t').\, t' < t, \Sigma)$. Then, the leakage is stateful, tracking past leakages in the variable L – it does not contradict this past leakage, and samples for each party whether this party won the current slot's leadership, given its available stake for this epoch.

procedure $\mathsf{Lkg}_{\mathsf{lead}}(\psi, \Sigma, \mathsf{Id}, \mathsf{Reg}, M, \mathfrak{C}_0, t)$
  let $(\mathfrak{C}, \mathfrak{C}_{\mathcal{H}}, \mathsf{spends}) \leftarrow \mathfrak{C}_{\mathsf{ideal}}(\Sigma, \mathsf{Id}, \mathfrak{C}_0, M)$
  let $\mathsf{epochFreeze} \leftarrow t^{\mathsf{epFreeze}}_{ep(t)}$
  let $(\mathfrak{C}^{ep}, \mathfrak{C}^{ep}_{\mathcal{H}}, \cdot) \leftarrow \mathfrak{C}_{\mathsf{ideal}}(\Sigma^{<\mathsf{epochFreeze}}, \mathsf{Id}, \mathfrak{C}_0, M)$
  if $(\psi, t) \notin L$ **then**
    let $v \leftarrow 0$
    let $v_t \leftarrow \sum_{(\cdot,v')\in\mathfrak{C}^{ep}} v'$
    for $\mathsf{coinId} \in \mathsf{Id}(\psi), (\mathsf{coinId}', v') \in \mathfrak{C}_{\mathcal{H}} \cap \mathfrak{C}^{ep}_{\mathcal{H}}$ **do**
      if $\mathsf{coinId} = \mathsf{coinId}'$ **then let** $v \leftarrow v + v'$
    let $\alpha_\psi \leftarrow v/v_t; L(\psi, t) \xleftarrow{*} \phi_f(\alpha_\psi)$
  return $(L(\psi, t), \mathsf{spends})$

In a preliminary step of our analysis we also utilise a leakage function leaking all information, $\mathsf{lkg}_{\mathsf{id}}$. This is effectively the identity function, simply returning the parameters passed to it. With this leakage the private ledger effectively becomes a "standard" $\mathcal{G}_{\mathsf{Ledger}}$ functionality, as the simulator still receives all information it would with the standard non-private ledger, with the exception of the transaction blinding on submission. In our security analysis we will forcibly disable this, by setting $\mathsf{blind}_{\mathcal{A}}(\cdot,\cdot,\mathsf{tx}) \coloneqq \mathsf{tx}$ instead.

5.3.3 Ideal-World Transaction Semantics

We consider ideal-world transactions starting with (PUBLIC, TRANSFER) to be *transfer transactions*. While it may appear sufficient to have ideal-world transfers appear as something like "give 0.05 of Alice's stake to Bob", our realisation of transfers using a Zerocash-like [BCG+14] design introduces some subtleties that need to be reflected in the ideal world. Specifically, we will require parties to specify *which* coins they are attempting to spend. Specifically, as in Zerocash, two coins are burned and two coins created, in any transfer. As a special case, as our protocol has no other minting functionality, we allow a zero-value coin to be burned in place of the second coin. Formally, the transactions have the following form: $((\text{PUBLIC}, \text{TRANSFER}), (\mathsf{pk}_r, \mathbf{c}_4), (\mathsf{pk}_s, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3))$, where $\mathbf{c}_i$ are ID/value pairs. This can be interpreted as "transfer the coins $\mathbf{c}_1$ and $\mathbf{c}_2$ to coins $\mathbf{c}_3$ and $\mathbf{c}_4$." It is worth noting that $\mathbf{c}_3$, while being a newly created coin, is not included in the component addressed to pk_r. It should be seen as a means of returning "change" from a transaction, corresponding to its real-world usage of Bitcoin and Zerocash transactions and should therefore also be addressed to the sending party. Coins are triples of an owner ID, a coin ID, and its value. Owner IDs must originate from the ledgers GENERATE interface (with the tag COIN), otherwise they are treated as invalid. Coin IDs are arbitrary (within $\{0,1\}^\kappa$) and are used to disambiguate coins with the same owner ID and value. Should a coin ID, owner ID, *and* value be reused, the coin it would create is ignored.

The function $\mathfrak{C}_{\mathsf{ideal}}$ is defined as follows:

function $\mathfrak{C}_{\mathsf{ideal}}(\Sigma, \mathsf{Id}, \mathfrak{C}_0, M)$

let $\mathfrak{C} \leftarrow \mathfrak{C}_0$

let $\mathfrak{C}_{\mathsf{spent}} \leftarrow \varnothing$

// $\mathfrak{C}_{\mathcal{H}}$ represents coins honest parties see.

let $\mathfrak{C}_{\mathcal{H}} \leftarrow \{ (\mathsf{idCoin}, v) \mid (\mathsf{idCoin}, v) \in \mathfrak{C}_0, \exists \psi \in \mathcal{H}\colon (\text{COIN}, \mathsf{idCoin}) \in \mathsf{Id}(\psi) \}$

// Map coins to the transaction it was created in.

let coinTx $\leftarrow \varnothing$

for $\mathbf{c} \in \mathfrak{C}_0$ **do let** $\mathsf{coinTx}(\mathbf{c}) \leftarrow$ GENESIS

// Adversarial coin IDs.

let $\mathsf{ids}_{\mathcal{A}} \leftarrow \varnothing$

// spends lists when adversarial coin IDs were spent.

let spends $\leftarrow \varnothing$

for $(\mathsf{txid}, \mathsf{tx}, t, \psi)$ **in** Σ **do**
  if $((\textsc{public}, \textsc{transfer}), (\mathsf{id}_r, \mathbf{c}_r), (\mathsf{id}_s, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)) \leftarrow \mathsf{tx}$ **then**
    $\forall i \in \{r, 1, 2, 3\}$: **let** $(\mathsf{ownId}_i, \mathsf{coinId}_i, v_i) \leftarrow \mathbf{c}_i$
    let $\mathsf{senderIds} \leftarrow \{(\textsc{coin}, \mathsf{ownId}_1), (\textsc{coin}, \mathsf{ownId}_2)\}$
    if $\{\mathbf{c}_1, \mathbf{c}_2\} \notin \mathfrak{C} \cup \{(\bot, 0)\}$ **then continue**
    if $\{\mathsf{coinId}_1, \mathsf{coinId}_2, \mathsf{coinId}_3, \mathsf{coinId}_r\} \notin \{0,1\}^\kappa$ **then continue**
    if $\psi \in \mathcal{H} \wedge (\textsc{id}, \mathsf{id}_s) \in \mathsf{Id}(\psi)$ **then let** $\mathsf{sender} \leftarrow \psi$
    else if $\psi \in \mathcal{A}$ **then let** $\mathsf{sender} \leftarrow \mathcal{A}$
    else continue
    if $\mathsf{sender} = \mathcal{A} \wedge \exists \psi \in \mathcal{H}$: $\mathsf{senderIds} \cap (\mathsf{Id}(\psi) \setminus \{(\textsc{coin}, \bot)\}) \neq \varnothing$ **then continue**
    if $\mathsf{sender} \neq \mathcal{A} \wedge \mathsf{senderIds} \notin \mathsf{Id}(\mathsf{sender}) \cup \{(\textsc{coin}, \bot)\}$ **then continue**
    if $\mathsf{sender} \neq \mathcal{A} \wedge (\textsc{coin}, \mathsf{ownId}_3) \notin \mathsf{Id}(\mathsf{sender})$ **then continue**
    if $\mathsf{sender} \neq \mathcal{A} \wedge \{\mathsf{coinTx}(\mathbf{c}_1), \mathsf{coinTx}(\mathbf{c}_2)\} \notin M_t(\psi) \cup \{\textsc{genesis}\}$ **then continue**
    if $v_r + v_3 \neq v_1 + v_2$ **then continue**
    let $\mathfrak{C}_{\mathsf{spent}} \leftarrow \mathfrak{C}_{\mathsf{spent}} \cup \{\mathbf{c}_1, \mathbf{c}_2\}$; $\mathfrak{C} \leftarrow \mathfrak{C} \cup \{\mathbf{c}_3, \mathbf{c}_r\} \setminus \mathfrak{C}_{\mathsf{spent}}$
    if $\mathsf{sender} = \mathcal{A}$ **then let** $\mathsf{ids}_{\mathcal{A}} \leftarrow \mathsf{ids}_{\mathcal{A}} \cup \{\mathsf{coinId}_r\}$
    for $i \in \{1, 2\}$ **where** $\mathsf{coinId}_i \in \mathsf{ids}_{\mathcal{A}}$ **do**
      let $\mathsf{spends} \leftarrow \mathsf{spends} \cup (\mathsf{txid}, i)$
    let $\mathsf{coinTx}(\mathbf{c}_3) \leftarrow \mathsf{txid}$; $\mathsf{coinTx}(\mathbf{c}_r) \leftarrow \mathsf{txid}$
    if $\exists \psi \in \mathcal{H}$: $\{(\textsc{coin}, \mathsf{ownId}_3), (\textsc{id}, \mathsf{id}_s)\} \subseteq \mathsf{Id}(\psi)$ **then**
      let $\mathfrak{C}_{\mathcal{H}} \leftarrow \mathfrak{C}_{\mathcal{H}} \cup \{\mathbf{c}_3\}$
    if $\exists \psi \in \mathcal{H}$: $\{(\textsc{coin}, \mathsf{ownId}_r), (\textsc{id}, \mathsf{id}_r)\} \subseteq \mathsf{Id}(\psi)$ **then**
      let $\mathfrak{C}_{\mathcal{H}} \leftarrow \mathfrak{C}_{\mathcal{H}} \cup \{\mathbf{c}_r\}$
    let $\mathfrak{C}_{\mathcal{H}} \leftarrow \mathfrak{C}_{\mathcal{H}} \setminus \mathfrak{C}_{\mathsf{spent}}$
return $(\mathfrak{C}, \mathfrak{C}_{\mathcal{H}}, \mathsf{spends})$

5.4 The Crypsinous Protocol

In this section we provide a detailed description of the Crypsinous UC protocol. The protocol has a similar structure as Ouroboros Genesis [BGK+18], but differs considerably in the leader election and the processing of transactions. As already discussed, the protocol assumes access to a global random oracle and clock, and functionalities for network, encryption, and NIZK.

5.4.1 High-Level Transaction Semantics

In the real world, the design looks slightly different than the ideal-world one presented in Subsection 5.3.3, following the approach of Zerocash [BCG+14]. Specifically, parties locally maintain, for each coin $\mathbf{c}$, nonces, $\rho_{\mathbf{c}}$ and commitment openings, $r_{\mathbf{c}}$, to their coins. In order to spend a coin, they reveal the deterministically derived serial number, $\mathsf{sn}_{\mathbf{c}}$, as well as prove the existence of a valid commitment, $\mathsf{cm}_{\mathbf{c}}$, somewhere in a Merkle tree of coin commitments. Like Zerocash, newly created coins are encrypted with the recipient party's public key, and the sending party is unable to spend them as it would require the recipient's private key to correctly generate the coin's serial number. The main difference is the design of addresses, corresponding to the ideal-world IDs. Parties generate a new coin public/secret key pair when given a GENERATE query, and update their secret key after spending a coin with it.

To become a leader at a time t, parties must prove knowledge of a path in a local Merkle tree of secret keys sk^{COIN}, labeled with t. This path is then erased by the party, to ensure leadership proofs cannot be re-made for past slots. This Merkle tree is created during key generation, with the coin's public key being derived from the Merkle tree's root, and the time of key generation. Each leaf is a PRF of the previous leaf, to reduce storage costs. We employ standard space/time trade-offs by keeping the top of the tree stored, recomputing parts of the bottom of the tree as needed. It is parameterised by the number of leaves R, which we leave as a system parameter, although we note it could also be defined as a per-user parameter.

A user's public key is derived from the root of the Merkle tree, root, and the time it was created, t. It is eligible for leadership so long as there are still paths in the tree to prove the existence of, after which the coin must be refreshed by spending it. We stress that this is a rare occurrence, as the assumption of honest majority relies on coins not only being held by honest parties, but also being eligible for leadership.

The protocol will take ideal transactions as an input and construct a corresponding Zerocash-style transaction in the real world. This transaction is then broadcast as usual in a blockchain protocol. On a READ request, the irrelevant information is not returned and only the information corresponding to the original ideal-world transaction is returned back to the requester. In addition to

transfers, we note that other types of transaction are accepted in the ideal world. We note that these are not validated, however, making the real-world equivalent far simpler to construct. Specifically, we encrypt each subtransaction with the public key of the party it is addressed to. On a READ request, the ciphertexts that the requesting party can decrypt are decrypted, and all others are replaced with $\perp$.

5.4.2 Protocol Overview

The protocol CRYPSINOUS assumes as hybrids two Δ-delay networks, $\mathcal{F}_{\text{Net}}^{\text{bc}}$ and $\mathcal{F}_{\text{Net}}^{\text{tx}}$, two non-interactive zero-knowledge functionalities $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$ and $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{XFER}}}$, a forward-secure encryption scheme $\mathcal{F}_{\text{FwEnc}}$, a global clock $\mathcal{G}_{\text{clock}}$, a random oracle $\mathcal{F}_{\text{RO}}$, a non-interactive equivocal commitment protocol, and a CRS used by the commitment scheme, to supply the commitment public key, $\mathcal{F}_{\text{CRS}}^{\text{init}_{\text{comm}}}$.

The protocol execution proceeds in discrete time intervals referred to as *slots*. As in Ouroboros Genesis, slots correspond directly to rounds given by $\mathcal{G}_{\text{clock}}$. In each slot *sl*, the parties execute a *staking procedure* to extend the blockchain. This proceeds similarly to Ouroboros Genesis, electing *leaders* to slots, with modifications to avoid revealing more information about the leader than necessary. We note that due to network-level attacks, the adversary is able to guess with good probability *which* party is the leader. Furthermore, due to serial numbers being revealed and being committing, the simulator must know when coins whose serial number the adversary could guess after corruption – specifically those sent by the adversary itself – were spent. This additional leakage can be avoided by a paranoid party, by it immediately transferring coins to itself on receipt. Furthermore, it is only an issue for parties which *may be corrupted*. In a hypothetical setting where the adversary committed to not corrupting a party, this party would no longer have leakage of this kind. Similar to Ouroboros Genesis, time is also divided into larger units, called epochs, with the distribution of stake considered for leadership purposes being frozen for each epoch.

We specify a concrete transaction system, based on Zerocash [BCG+14]. Parties hold *coins* with inherent value, and a fixed total value across the system (a restriction imposed for simplifying the analysis. Adding block rewards would be a straightforward extension). The Ouroboros Genesis leadership election is performed on a per-coin basis, with each coin competing separately. If any of

a party's coins win the election, the party proceeds to generate a new block, extending their current chain. The block itself is generated as in Ouroboros Genesis, although the validity of it is proved differently. Specifically, $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}_{\mathsf{LEAD}}}$ is used to produce a signature of knowledge of a coin that won the leadership election during a given slot. This proof is done in a Zerocash style, and involves renewing the coin in question. Specifically, the Zerocash serial number of the leading coin is revealed, and a new coin of the same value is minted. We also refer to this proof, together with its auxiliary information such as the spent serial number and newly created coin commitment, as a *leadership transaction*.

We note that Ouroboros Genesis requires the stakeholder distribution to be frozen to prevent grinding attacks. In order to allow a coin to be used for leadership proofs multiple times in an epoch, we introduce a new resistance mechanism against attacks of this type: The newly generated coin in a leadership transaction has its nonce deterministically derived from the nonce of the old coin. The leadership test itself utilises only this nonce from the coin as a seed – it follows that the leadership test for the derived coin is fixed along with the randomness of the epoch.

Once a block is created, the party broadcasts the new chain, extended with this block. Furthermore, the party broadcasts the leadership transaction separately, in order to ensure the newly created coin will eventually be valid, even if the consensus does not adopt the broadcast chain.

A chain proposed by any party might be adopted only if it satisfies the following two conditions: (1) it is valid according to a well defined validation procedure, and (2) the block corresponding to each slot has a signature of knowledge from a coin winning the corresponding slot.

To ensure the second property we need the implicit slot-leader lottery to provide its winners (slot leaders) with a certificate/proof of slot-leadership. For this reason, we implement the slot-leader election as follows: Each party ψ checks, for each of their coins $\mathbf{c}$, whether or not it is a slot leader, by locally evaluating a maliciously-unpredictable pseudo-random function, as described in Subsection 5.2.4, with entropy supplied by the epoch randomness η_{ep}, by being evaluated at the slot index sl and η_{ep}, seeded with the "winning coin's secret key" $\mathsf{root}_{\mathbf{c}} \parallel \rho_{\mathbf{c}}$. The generation of η_{ep} is similar to Ouroboros Genesis – it is initially supplied through the CRS, then for subsequent epochs, it is sampled in a maliciously-unpredictable way from "randomness contributions" ρ provided

by slot leaders over the course of the previous epoch.

Specifically, we will use the MUPRF construction of Subsection 5.2.4, for a given group $\mathbb{G}$. If the MUPRF output y is below a certain threshold $T_{\mathbf{c}}$ – which depends on $\mathbf{c}$'s stake – then ψ is an eligible slot leader; furthermore, he can generate a signature of knowledge of a valid coin which satisfies these conditions. In particular, each new block broadcast by a slot leader contains a NIZK proof π, signing the rest of the block content, with the knowledge of the nonce $\rho_{\mathbf{c}}$, $sk^{\text{COIN}}_{\mathbf{c},sl}$ for the slot sl the leadership transaction is for, proving that the nonce and secret key correspond to some unspent coin commitment $\text{cm}_{\mathbf{c}}$. The leadership transaction also *evolves* the coin that wins leadership – this is done in order to establish adaptive security, and is done by updating the coin nonce used: $\rho_{\mathbf{c}'} = \text{prf}^{\text{evl}}_{\text{root}_{\mathbf{c}}}(\rho_{\mathbf{c}})$. A new coin, in the same value, with this updated – and, crucially, deterministic – nonce is created and committed in the transaction. In particular, parties erase $\rho_{\mathbf{c}}$ and only maintain $\rho_{\mathbf{c}'}$ after the leadership proof is generated.

As in Ouroboros Genesis, it is possible for multiple parties, or no party to be a leader of any given slot. Our protocol behaves identically to Genesis in this regard, and we utilise the same chain selection rule in our protocol.

We next turn to the formal specification of the protocol Crypsinous. We follow closely the modular design of Ouroboros Genesis, beginning with a master protocol described below. Most of the subcomponents will be introduced throughout the rest of this section, however the details of transaction processing is omitted. Formally this is done through the fetch-information protocol, which fetches transactions and chains from the network, and updates the local ledger state accordingly.

Protocol Crypsinous

The Crypsinous protocol is divided into several sub-protocols. For simplicity, where these use the same state variable names, these variables are considered shared.

State variables and initialisation values:

Variable	Description
$\mathfrak{C}_{\text{free}} \coloneqq \varnothing$	Set of unbound secret keys
$\text{pk}^{\text{enc}} \coloneqq \perp$	Encryption public key

When receiving a message maintain-ledger *from a party ψ:*

run ledger-maintenance

When receiving a message (SUBMIT, tx) *from a party* ψ*:*

if $\mathsf{tx} = ((\textsc{public}, \textsc{transfer}), (\cdot,\cdot), (\mathsf{pk}^{\mathsf{enc}}, \cdot, \cdot, \cdot))$ **then**

return SUBMIT-XFER(tx)

else

return SUBMIT-GENERIC(tx)

When receiving a message (GENERATE, tag) *from a party* ψ*:*

if tag = COIN **then**

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

let $sk_t^{\textsc{coin}} \xleftarrow{*} \{0,1\}^{\ell_{\mathsf{prf}}}$

for i **in** $\{t+1, \ldots, t+R\}$ **do**

let $sk^{\textsc{coin}} t+i \leftarrow \mathsf{prf}^{\mathsf{evl}}_{sk_i^{\textsc{coin}}}(1)$

let $\mathsf{root}_{sk^{\textsc{coin}}} \leftarrow \mathsf{merkleRoot}\left(\left\{sk_t^{\textsc{coin}}, \ldots, sk_{t+R}^{\textsc{coin}}\right\}\right)$

let $pk^{\textsc{coin}} \leftarrow \mathsf{prf}^{\mathsf{pf}}_{\mathsf{root}_{sk^{\textsc{coin}}}}(t)$

let $\mathfrak{C}_{\mathsf{free}} \leftarrow \mathfrak{C}_{\mathsf{free}} \cup \{\mathsf{root}_{sk^{\textsc{coin}}}\}$

return $pk^{\textsc{coin}}$

else if tag = ID $\wedge\ \mathsf{pk}^{\mathsf{enc}} = \perp$ **then**

send KEYGEN **to** $\mathcal{F}_{\mathsf{FwEnc}}$ and **receive the reply** pk

let $\mathsf{pk}^{\mathsf{enc}} \leftarrow \mathsf{pk}$

return pk

else

return $r \xleftarrow{*} \{0,1\}^{\kappa}$

When receiving a message READ *from a party* ψ*:*

return READ-STATE

5.4.3 Real-World Transactions

Before giving further parts of the formal specification we introduce some necessary terminology and notation. Each party ψ stores a local blockchain $\mathcal{C} - \psi$'s local view of the blockchain. Such a local blockchain is a sequence of blocks B_i $(i > 0)$ where each $B \in \mathcal{C}$ has the following format: $B = (\mathsf{tx}_{\mathsf{lead}}, \mathsf{st})$; where $\mathsf{tx}_{\mathsf{lead}} = (\textsc{lead}, \vec{\mathsf{stx}}_{\mathsf{ref}}, \mathsf{stx}_{\mathsf{proof}})$ and $\mathsf{stx}_{\mathsf{proof}} = (\mathsf{cm}_{\mathbf{c}'}, \mathsf{sn}_{\mathbf{c}}, ep, sl, \rho, h, \mathsf{ptr}, \pi)$. Here, st is the encoded data of this block, h is the hash of the same data, sl and ep are the slot and epoch the block is for, respectively, $(\mathsf{cm}_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{comm}(pk^{\textsc{coin}} \parallel t \parallel v_{\mathbf{c}} \parallel \rho_{\mathbf{c}'})$ is the commitment of the newly-created coin, and $\mathsf{sn}_{\mathbf{c}} = \mathsf{prf}^{\mathsf{sn}}_{\mathsf{root}^{\textsc{coin}}_{sk}}(\rho_{\mathbf{c}})$ is the serial number of the coin $\mathbf{c}$, which is revealed to demonstrate the coin has not been

spent. We define $\rho = \mu^{sk_{sl}^{\text{COIN}}}$, where μ is $\mathcal{F}_{\text{RO}}$ evaluated at $\text{NONCE} \parallel \eta_{ep} \parallel sl; \rho$ is the randomness contribution to the next epoch's randomness, ptr is the hash of the previous block, and π is a NIZK proof of the statement LEAD, defined below. The component $\vec{\text{stx}}_{\text{ref}}$ consists of a (typically empty) vector of reference leadership transactions. These are processed *before* the leadership transaction itself is processed and serve to allow successive leadership proofs with the same coin, even when the selected chain switches.

Definition of LEAD. A tuple (x, w) is in $\mathcal{R}_{\text{LEAD}}$ if and only if all of the following hold:

- $x = (\text{cm}_{\mathbf{c}_2}, \text{sn}_{\mathbf{c}_1}, \eta, sl, \rho, h, ptr, \mu_\rho, \mu_y, \text{root})$
- $w = (\text{path}, \text{root}_{sk^{\text{COIN}}}, \text{path}_{sk^{\text{COIN}}}, t_{\mathbf{c}}, \rho_{\mathbf{c}_1}, \text{cm}_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v, r_{\mathbf{c}_2})$
- $pk^{\text{COIN}} = \text{prf}^{\text{pk}}_{\text{root}_{sk^{\text{COIN}}}}(t_{\mathbf{c}})$
- $\rho_{\mathbf{c}_2} = \text{prf}^{\text{evl}}_{\text{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$
- $\forall i \in \{1, 2\} : \text{deComm}(\text{cm}_{\mathbf{c}_i}, pk^{\text{COIN}} \parallel v \parallel \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \top$
- path is a valid Merkle tree path to $\text{cm}_{\mathbf{c}_1}$ in a tree with root root.
- $\text{path}_{sk^{\text{COIN}}}$ is a valid path to a leaf at position $sl - t_{\mathbf{c}}$ in a tree with root $\text{root}_{sk^{\text{COIN}}}$.
- $\text{sn}_{\mathbf{c}_1} = \text{prf}^{\text{sn}}_{\text{root}_{sk^{\text{COIN}}}}(\rho_{\mathbf{c}_1})$
- $y = \mu_y^{\text{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}} \parallel \rho_{\mathbf{c}}}; \rho = \mu_\rho^{\text{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}} \parallel \rho_{\mathbf{c}}}$
- $y < \text{ord}(G)\phi_f(v)$

Note that x of LEAD contains values sl, h, ptr that seemingly nothing is proven about. As the NIZK is non-malleable, this makes them effectively part of a signature of knowledge message.

Kinds of transactions. CRYPSINOUS handles three kinds of transactions: *Leadership transactions*, such as the above tx_{lead}, *transfer transactions* tx_{xfer}, and *general-purpose transactions*. Each of these is handled separately. The transfer transactions and general-purpose transactions correspond directly to ideal-world transactions with the same behaviour. Leadership transactions by contrast exist only in the real world.

General-purpose transactions in the ideal world consist of a vector of subtransactions, addressed either to everyone (PUBLIC), or to a specific party. The corresponding real-world transaction is a vector of the same subtransactions,

which are either directly the content of the ideal world transaction, in the case of a transaction addressed to PUBLIC, or an encryption of the content using $\mathcal{F}_{\text{FwEnc}}$, to the party specified as the recipient. Upon reading the state, parties attempt to decrypt ciphertexts and, failing that, replace it with $\perp$. To disambiguate transactions, we prefix generic transactions with the label GENERIC.

The implementation of transfer transactions is more involved, as we not only want to guarantee their privacy, but also their validity. To achieve this, we replace transaction which fall into the permissible ideal-world format – which we recall, is $\mathsf{tx}^{\text{ideal}}_{\text{xfer}} = ((\text{PUBLIC}, \text{TRANSFER}), (\mathsf{pk}_r, (\mathsf{id}_4, v_4)), (\mathsf{pk}_s, (\mathsf{id}_1, v_1), (\mathsf{id}_2, v_2), (\mathsf{id}_3, v_3)))$ – with a cryptographic construction hiding the respective information. We define a real transfer transaction to be: $\mathsf{tx}^{\text{real}}_{\text{xfer}} = (\text{TRANSFER}, \mathsf{stx}_{\text{proof}}, c_r)$, where $\mathsf{stx}_{\text{proof}} = (\{\mathsf{cm}_{\mathbf{c}_3}, \mathsf{cm}_{\mathbf{c}_4}\}, \{\mathsf{sn}_{\mathbf{c}_1}, \mathsf{sn}_{\mathbf{c}_2}\}, t, \mathsf{root}, \pi)$, c_r is a $\mathcal{F}_{\text{FwEnc}}$-encryption for the slot the transaction was submitted on, and $\mathsf{stx}_{\text{rcpt}} = (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_{\mathbf{c}_3})$ to pk_r. Similar to leadership transactions, $(\mathsf{cm}_{\mathbf{c}_3}, r_{\mathbf{c}_3}) = \mathsf{comm}(pk^{\text{COIN}}_{\mathsf{pk}_s} \parallel t \parallel v_{\mathbf{c}_3} \parallel \rho_{\mathbf{c}_3})$ and $(\mathsf{cm}_{\mathbf{c}_4}, r_{\mathbf{c}_4}) = \mathsf{comm}(pk^{\text{COIN}}_{\mathsf{pk}_r} \parallel t \parallel v_{\mathbf{c}_4} \parallel \rho_{\mathbf{c}_4})$; $\mathsf{sn}_{\mathbf{c}_1}$ and $\mathsf{sn}_{\mathbf{c}_2}$ are revealed to spend the coins $\mathbf{c}_1$ and $\mathbf{c}_2$, respectively, and π proves the statement XFER, defined below, specifically proving the existence of $\mathsf{cm}_{\mathbf{c}_1}$ and $\mathsf{cm}_{\mathbf{c}_2}$, in the Merkle tree of coin commitments with the root root, as well as various consistency properties. The use of $\mathcal{F}_{\text{FwEnc}}$ implies that parties will not be able to decrypt ciphertexts addressed to them indefinitely, however they are still required to respond with the corresponding ideal-world information to READ requests. As a result, when a transfer transaction is first seen and decrypted, the corresponding ideal world transaction is locally stored. Furthermore, parties maintain locally the information needed to spend coins they own – specifically $(pk^{\text{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$.

Definition of XFER. A tuple (x, w) is in $\mathcal{R}_{\text{XFER}}$ if and only if all of the following hold:

- $x = (\{\mathsf{cm}_{\mathbf{c}_3}, \mathsf{cm}_{\mathbf{c}_4}\}, \{\mathsf{sn}_{\mathbf{c}_1}, \mathsf{sn}_{\mathbf{c}_2}\}, t, \mathsf{root})$
- $w = (\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, \mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, pk^{\text{COIN}}_{\mathbf{c}_3}, pk^{\text{COIN}}_{\mathbf{c}_4}, (\mathsf{cm}_{\mathbf{c}_1}, \rho_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v_1, \mathsf{path}_1), (\mathsf{cm}_{\mathbf{c}_2}, \rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, v_2, \mathsf{path}_2), (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_3), (\rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_4))$
- $\forall i \in \{1, 2\} : pk^{\text{COIN}}_{\mathbf{c}_i} = \mathsf{prf}^{\mathsf{pk}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}}(1)$
  (if $i = 2, v_2 = 0$, this check may be skipped)
- $\forall i \in \{1, \ldots, 4\} : \mathsf{deComm}(\mathsf{cm}_{\mathbf{c}_i}, pk^{\text{COIN}}_{\mathbf{c}_i} \parallel v_i \parallel \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \top$
  (if $v_2 = 0$, this check may be skipped for $i = 2$)
- $v_1 + v_2 = v_3 + v_4$

- path_1 is a valid path to $\mathsf{cm}_{\mathbf{c}_1}$ in a tree with root root.
- path_2 is a valid path to $\mathsf{cm}_{\mathbf{c}_2}$ in a tree with root root,
  or $v_2 = 0$ and $\mathsf{sn}_{\mathbf{c}_2} = \mathsf{prf}^{\mathsf{zdrv}}_{\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$.
- $\mathsf{path}_{sk^{\mathrm{COIN}}_{\mathbf{c}_i}}$ is a valid path to a leaf at position t in $\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}_i}}$, for $i \in \{1, 2\}$.
- $\forall i \in \{1, 2\} : \mathsf{sn}_{\mathbf{c}_i} = \mathsf{prf}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}_i}}}(\rho_{\mathbf{c}_i})$
  (or, if $v_2 = 0$, this check may be skipped for $i = 2$)

5.4.4 Interacting with the Ledger

At the core of the CRYPSINOUS protocol is the process that allows parties to maintain the ledger. There are three types of processes that are triggered by three different commands provided that the party is already registered to all its local and global functionalities.

- The command (SUBMIT, tx) is used for sending a new transaction to the ledger. The party maps tx to a corresponding $\mathsf{tx}^{\mathsf{real}}$, which is stored in the party's local transaction buffer, and is multicast to the network.

- The command GENERATE is used for creating a new address, which can be used by other parties to transfer funds to this current party.

- The command READ is used for the environment to ask for a read of the current ledger state. On receipt, the party maps each transaction $\vec{\mathsf{st}}^{\lceil k}$ to its ideal-world equivalent, and returns this ideal-world chain.

- The command MAINTAIN-LEDGER triggers the main ledger update. A party receiving this command first fetches from its network all information relevant for the current round, then it uses the received information to update its local info – i.e., asks the clock for the current time t, updates its epoch counter ep, its slot counter sl, and its (local view of) stake distribution parameters, accordingly; finally it invokes the staking procedure unless it has already done so in the current round. If this is the first time that the party processes a MAINTAIN-LEDGER message then before doing anything else, the party invokes an initialisation protocol to receive the initial information it needs to start executing the protocol – in particular the genesis block.

The relevant sub-processes involved in handling these queries are detailed in the following sections. After introducing each of these basic ingredients, we conclude with a technical overview of the main ledger maintenance protocol LEDGER-MAINTENANCE, a detailed specification of the protocol READ-STATE for answering requests to read the ledger's state, and a detailed specification of the protocols SUBMIT-XFER and SUBMIT-GENERIC.

5.4.4.1 Party Initialisation

A party that has been registered with all its resources and setups becomes operational by invoking the initialisation protocol CRYPSINOUS-INIT upon processing its first command. As a first step the party receives its encryption key from $\mathcal{F}_{\mathsf{FwEnc}}$. It receives any initial stake it may have as a single coin from $\mathcal{F}_{\mathsf{Init}}$. Subsequently, protocol CRYPSINOUS-INIT proceeds as in Ouroboros Genesis, although it does not register any keys. This is managed instead by the ledgers GENERATE interface. Formally, the initialisation procedure is specified as:

Protocol CRYPSINOUS-INIT

The CRYPSINOUS initialisation procedure claims any initial stake and retrieves the genesis block.

State variables and initialisation values:

Variable	Description
$\mathfrak{C}_{\mathsf{free}} \coloneqq \varnothing$	Set of unbound secret keys
$\mathfrak{C} \coloneqq \varnothing$	Set of spendable coins
$\mathcal{C} \coloneqq \bot$	The currently selected chain
claimed $\coloneqq \bot$	If initial funds have been claimed

Prior to any other interaction, if $\mathcal{C} = \bot$:

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
let $ep \leftarrow ep(t)$
if $t = 0 \wedge \mathsf{init} = \bot$ **then**
  send CLAIM **to** $\mathcal{F}_{\mathsf{Init}}$ and **receive the reply** $((pk^{\mathrm{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}), sk^{\mathrm{COIN}})$
  let $\mathfrak{C} \leftarrow \mathfrak{C} \cup \left\{(pk_{\mathbf{c}}^{\mathrm{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\right\}, \mathfrak{C}_{\mathsf{free}} \leftarrow \mathfrak{C}_{\mathsf{free}} \cup \left\{sk^{\mathrm{COIN}}\right\}$
  let $\mathsf{init} \leftarrow \top$
else if $t > 0$ **then**
  send GENESIS **to** $\mathcal{F}_{\mathsf{Init}}$ and **receive the reply** $\mathfrak{G}$
  let $\mathcal{C} \leftarrow \mathfrak{G}$

5.4.4.2 The Staking Procedure

The next part of the ledger-maintenance protocol is the staking procedure which is used for the slot leader to compute and send the next block. A party ψ is an eligible slot leader for a particular slot *sl* in an epoch *ep* if one of ψ's coins, $\mathbf{c}$, is both eligible for leadership in *ep*, and a PRF-value depending on *sl* and the coin nonce $\rho_{\mathbf{c}}$ and secret key sk_t^{COIN} is smaller than a threshold value $T_{\mathbf{c}}$. We discuss when a coin is considered eligible for leadership and how its threshold is determined. A coin is eligible for leadership depending on when, and how, its corresponding commitment entered the chain. Specifically, if its corresponding commitment was created in a transfer transaction, it is valid in a similar way as transactions are considered for leadership in an epoch: If it is sufficiently old by the time the epoch starts, it is taken as part of the snapshot fixing the stake distribution for *ep*. Commitments originating from leadership transactions are always immediately eligible for leadership, as their nonce and secret key are deterministically derived. It is possible, although unusual, for a coin created in a leadership transaction in a fork to be used eligible for leadership in an unrelated fork of the chain. In this case, the coin is *still eligible*, as the originating leadership transaction will be added to $\vec{\text{stx}}_{\text{ref}}$.

Each coin $\mathbf{c}$'s value $v_{\mathbf{c}}$ induces a relative stake for the coin, $\alpha_{\mathbf{c}}$. We use the same function $\phi_f(\alpha_{\mathbf{c}})$ to determine the probability of a coin winning the leadership election, with the corresponding threshold, $T_{\mathbf{c}} = \text{ord}(\mathbb{G})\phi_f(\alpha_{\mathbf{c}})$. Due to the independent aggregation property of ϕ_f, the probability of a party winning the leadership election in CRYPSINOUS and in Genesis is initially the same, regardless of how stake is split between coins. One key difference, however, is that when a coin is *transferred* in CRYPSINOUS, it is *no longer eligible for leadership*. As a direct consequence, any stake transferred during an epoch must be considered adversarial for the given epoch.

The staking procedure evaluates two distinct MUPRFs for each eligible coin. If the output of one of these is under the target for some coin, the party is a slot leader and continues to create a new block B from their current transaction buffer. Aside of the main contents, the party assembles a leadership transaction and assigns it to the block. This transaction includes a NIZK proof of leadership – specifically of the statement LEAD – and acts as a signature of knowledge over the block content, as well as the pointer to the previous block. An updated

blockchain $\mathcal{C}$ containing the new block B is finally broadcast over the network. Formally, the staking procedure is specified as:

Protocol STAKING-PROCEDURE

The CRYPSINOUS staking procedure attempts to extend the current chain, and broadcasts any successful new blocks created. The group used in Subsection 5.2.4 is denoted G, and a mapping from random oracle outputs to corresponding group elements is assumed, also denoted by $G(x)$. Further, a collision-resistant hash function H is used to link blocks, and a family of pseudo-random functions labelled prf_b^a when operating on domain a with key b is used to deterministically derive random values.

State variables and initialisation values:

Variable	Description
$\mathfrak{C}_{\text{free}} \coloneqq \varnothing$	Set of unbound secret keys
$\mathfrak{C} \coloneqq \varnothing$	Set of spendable coins
$\mathcal{C} \coloneqq \perp$	The currently selected chain
$\mathsf{txBuf} \coloneqq \varepsilon$	Buffer of transactions to include
$\mathsf{doneLead} \coloneqq 0$	The lead time leadership was tested

On invocation:

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t
if $\mathsf{doneLead} = t \vee \mathcal{C} = \perp$ **then return**
let $\mathsf{doneLead} \leftarrow t$
for $(pk_{\mathbf{c}}^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}) \in \mathfrak{C}$ **do**
  if c is not eligible for leadership **then continue**
  send $(\text{QUERY}, \text{NONCE} \parallel \eta_{ep(t)} \parallel t)$ **to** $\mathcal{F}_{\text{RO}}$ and **receive the reply** μ_ρ
  send $(\text{QUERY}, \text{LEAD} \parallel \eta_{ep(t)} \parallel t)$ **to** $\mathcal{F}_{\text{RO}}$ and **receive the reply** μ_y
  retrieve $sk_{\mathbf{c},t}^{\text{COIN}}$, $\mathsf{root}_{\mathbf{c}}$, and $t_{\mathbf{c}}$ in $\mathfrak{C}_{\text{free}}$ corresponding to $pk_{\mathbf{c}}^{\text{COIN}}$
  let $\rho \leftarrow G(\mu_\rho)^{\mathsf{root}_{sk_{\mathbf{c}}^{\text{COIN}}} \parallel \rho_{\mathbf{c}}}; y \leftarrow G(\mu_y)^{\mathsf{root}_{sk_{\mathbf{c}}^{\text{COIN}}} \parallel \rho_{\mathbf{c}}}$
  if $y < \mathsf{ord}(G) \cdot \phi_f(v_{\mathbf{c}})$ **then**
    let $B \leftarrow \varepsilon$
    for $\mathsf{tx} \in \mathsf{txBuf}$ **do**
      if $\mathsf{validate}(\mathsf{tx}, \mathcal{C} \parallel B)$ **then**
        let $B \leftarrow B \parallel \mathsf{tx}$
    let $\mathsf{txBuf} \leftarrow \varepsilon$
    let $\mathsf{ptr} \leftarrow H(\mathcal{C}); h \leftarrow H(B)$

let $\rho_{\mathbf{c}'} \leftarrow \mathsf{prf}^{\mathsf{evl}}_{\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}}}}(\rho_{\mathbf{c}})$; $\mathsf{sn}_{\mathbf{c}} \leftarrow \mathsf{prf}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}}}}(\rho_{\mathbf{c}})$
let $(\mathsf{cm}_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{comm}(pk^{\mathrm{COIN}} \parallel v_{\mathbf{c}} \parallel \rho_{\mathbf{c}'})$
let $\vec{\mathsf{stx}}_{\mathsf{ref}}$ be the ordered leadership transactions made by ψ not in $\mathcal{C}$
let (root, path) be the root of $\mathbb{C}^{\mathsf{lead}}$ in $\mathcal{C}$, after applying all transactions in $\vec{\mathsf{stx}}_{\mathsf{ref}}$ and the path to $\mathsf{cm}_{\mathbf{c}}$ in this tree
let $\mathsf{path}_{\mathbf{c}}$ be the path to $sk^{\mathrm{COIN}}_{\mathbf{c},t}$ in a tree in $\mathfrak{C}_{\mathsf{free}}$
let $x = (\mathsf{cm}_{\mathbf{c}'}, \mathsf{sn}_{\mathbf{c}}, \eta_{ep}, sl, \rho, h, ptr, \mu_{\rho}, \mu_{y}, \mathrm{root})$
let $w = (\mathrm{path}, \mathrm{root}_{sk^{\mathrm{COIN}}}, \mathsf{path}_{\mathbf{c}}, t_{\mathbf{c}}, \rho_{\mathbf{c}}, \mathsf{cm}_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}, r_{\mathbf{c}'})$
send $(\textsc{prove}, x, w)$ **to** $\mathcal{F}^{\mathcal{R}_{\textsc{lead}}}_{\mathsf{NIZK}}$ and **receive the reply** π
let $\mathsf{tx}_{\mathsf{lead}} = (\textsc{lead}, \vec{\mathsf{stx}}_{\mathsf{ref}}, (\mathsf{cm}_{\mathbf{c}'}, \mathsf{sn}_{\mathbf{c}}, ep, sl, \rho, h, ptr, \pi))$
let $\mathcal{C} \leftarrow \mathcal{C} \parallel (\mathsf{tx}_{\mathsf{lead}}, B)$
let $\mathfrak{C} \leftarrow \left(\mathfrak{C} \setminus \left\{(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\right\}\right) \cup \left\{(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}'}, r_{\mathbf{c}'}, v_{\mathbf{c}})\right\}$
send $(\textsc{bcast}, \mathsf{tx}_{\mathsf{lead}})$ **to** $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Net}}$
send $(\textsc{bcast}, \mathcal{C})$ **to** $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Net}}$
break

From the staking procedure we construct the ledger maintenance protocol, which in addition to attempting to stake on each block, catalogues received transactions, ensures information and forward-security is up-to-date, as well as claiming initial stake.

Protocol LEDGER-MAINTENANCE

The ledger maintenance protocol LEDGER-MAINTENANCE organises received transactions and chains, claims initial stake and the genesis block, and participates in staking.

State variables and initialisation values:

Variable	Description
$\mathfrak{C}_{\mathsf{cnd}} \coloneqq \varnothing$	A set of not-yet spendable coins
$\mathsf{log} \coloneqq \varepsilon$	A log of received transaction information
$\mathcal{C} \coloneqq \perp$	The currently selected chain
$\mathsf{txBuf} \coloneqq \varepsilon$	Buffer of transactions to include
$t_{\mathsf{fwenc}} \coloneqq 0$	The "time" of $\mathcal{F}_{\mathsf{FwEnc}}$

On invocation:

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
if $t = 0 \vee \mathcal{C} = \perp$ **then**
  run CRYPSINOUS-INIT

if $t = 0$ **then return**

run newTx ← FETCH-INFORMATION

let txBuf ← txBuf ‖ newTx

send $(\text{BCAST}, \mathcal{C})$ **to** $\mathcal{F}_{\text{Net}}^{\text{bc}}$

for tx **in** txBuf **do**

  send $(\text{BCAST}, \text{tx})$ **to** $\mathcal{F}_{\text{Net}}^{\text{tx}}$

for tx **in** newTx **do**

  if $\text{tx} = (\text{TRANSFER}, \text{stx}_{\text{proof}} = (\text{cms}, \cdot, \cdot, \cdot), c_{\text{rcpt}})$ **then**

    send $(\text{DECRYPT}, c_{\text{rcpt}})$ **to** $\mathcal{F}_{\text{FwEnc}}$ and **receive the reply** m

    if $m = (\text{OK}, (pk_{\mathbf{c}}^{\text{COIN}}, t, \rho_{\mathbf{c}}, r_{\mathbf{c}}, \text{coinId}_{\mathbf{c}}, v_{\mathbf{c}})) \wedge \text{cm}_{\mathbf{c}} \in \text{cms}$ **then**

      if $\nexists sk_t^{\text{COIN}} \in \mathfrak{C}_{\text{free}}$ corresponding to $pk_{\mathbf{c}}^{\text{COIN}}$ **then continue**

      let $\mathfrak{C}_{\text{cnd}} \leftarrow \mathfrak{C}_{\text{cnd}} \cup \left\{(pk_{\mathbf{c}}^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\right\}$

      let log ← log ‖ $(\text{tx}, \text{RECEIVE}, (pk_{\mathbf{c}}^{\text{COIN}}, \text{coinId}_{\mathbf{c}}, v_{\mathbf{c}}))$

  else if $\text{tx} = (\text{GENERIC}, c_1, \ldots, c_n)$ **then**

    for c **in** $\{c_1, \ldots, c_n\}$ **do**

      if $c = (\text{PUBLIC}, m)$ **then continue**

      else if $c = (\text{PRIVATE}, c')$ **then**

        send $(\text{DECRYPT}, c')$ **to** $\mathcal{F}_{\text{FwEnc}}$ and **receive the reply** m

        if $m = (\text{OK}, m')$ **then let** log ← log ‖ $(\text{PLAINTEXT}, c', m')$

for $(sk_{\mathbf{c}}^{\text{COIN}}, t_{\mathbf{c}}) \in \mathfrak{C}_{\text{free}}$ **do**

  for $\exists c = (pk_{\mathbf{c}}^{\text{COIN}}, \ldots) \in \mathfrak{C}_{\text{cnd}}$ whose transaction is in $\mathcal{C}^{\lceil k}$ **do**

    let $\mathfrak{C}_{\text{cnd}} \leftarrow \mathfrak{C}_{\text{cnd}} \setminus \{c\}$; $\mathfrak{C} \leftarrow \mathfrak{C} \cup \{c\}$

  $\forall t' \leq t$ **erase** $sk_{\mathbf{c},t'}^{\text{COIN}}$ from $\mathfrak{C}_{\text{free}}$

while $t_{\text{fwenc}} < t - k$ **do**

  send UPDATE **to** $\mathcal{F}_{\text{FwEnc}}$

  let $t_{\text{fwenc}} \leftarrow t_{\text{fwenc}} + 1$

run STAKING-PROCEDURE

5.4.4.3 Submitting Transactions

Transactions submitted to the CRYPSINOUS protocol are, as previously discussed, first mapped to corresponding real-world transactions, which then get handled as standard ledger transactions by being broadcast over a multicast network, and being assembled into blocks. Specifically, transfer transactions are mapped to Zerocash-like transactions, where only the first coin received

to a given address is spent, and other transactions are mapped into encrypted components. The submitting procedure for transfer transactions is:

Protocol SUBMIT-XFER($\mathsf{tx}_{\mathsf{xfer}}$)

The submission procedure for transfer transactions builds a working zero-knowledge proof to authenticate the transfer. If this cannot be done, it falls back to submitting the transaction as "generic".

State variables and initialisation values:

Variable	Description
$\mathfrak{C} \coloneqq \varnothing$	Set of spendable coins
$\mathsf{log} \coloneqq \varepsilon$	A log of received transaction information
$\mathcal{C} \coloneqq \bot$	The currently selected chain

On invocation:

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

let $((\mathsf{pk}_r^{\mathsf{enc}}, (pk_{\mathbf{c}_4}^{\mathrm{COIN}}, \mathsf{coinId}_4, v_4)), (\mathsf{pk}_s^{\mathsf{enc}}, (pk_{\mathbf{c}_1}^{\mathrm{COIN}}, \mathsf{coinId}_1, v_1), (pk_{\mathbf{c}_2}^{\mathrm{COIN}}, \mathsf{coinId}_2, v_2), (pk_{\mathbf{c}_3}^{\mathrm{COIN}}, \mathsf{coinId}_3 v_3))) \leftarrow \mathsf{tx}_{\mathsf{xfer}}$

if $\mathsf{pk}_s^{\mathsf{enc}} \neq \mathsf{pk}^{\mathsf{enc}} \vee v_1 + v_2 \neq v_3 + v_4 \vee pk_{\mathbf{c}_3}^{\mathrm{COIN}} \notin \mathfrak{C}_{\mathsf{free}}$ **then**

  return SUBMIT-GENERIC($\mathsf{tx}_{\mathsf{xfer}}$)

let $\mathbf{c}_1$ and $\mathbf{c}_2$ be the first coins received at $(pk_{\mathbf{c}_1}^{\mathrm{COIN}}, \mathsf{coinId}_1)$ and $(pk_{\mathbf{c}_2}^{\mathrm{COIN}}, \mathsf{coinId}_2)$, respectively, according to log

let $\mathbf{c}_1, \mathbf{c}_2 \in \mathfrak{C}$ be the (potentially) evolved variant of $\mathbf{c}'_1, \mathbf{c}'_2$.

ensure $\mathbf{c}_1, \mathbf{c}_2$ values are v_1, v_2, and their transactions are in $\mathcal{C}^{\lceil k}$

  (if $v_2 = 0 \wedge pk_{\mathbf{c}_2}^{\mathrm{COIN}} = \bot$, ignore $\mathbf{c}_2$)

if the above failed **then**

  return SUBMIT-GENERIC($\mathsf{tx}_{\mathsf{xfer}}$)

retrieve $(pk_{\mathbf{c}_i}^{\mathrm{COIN}}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, v_{\mathbf{c}_i})$ from $\mathfrak{C}$ for $i \in \{1, 2\}$

retrieve $sk_{\mathbf{c}_i}^{\mathrm{COIN}}$ from $\mathfrak{C}_{\mathsf{free}}$ for $i \in \{1, 2\}$

let $\mathsf{root}_{sk_{\mathbf{c}_i}^{\mathrm{COIN}}}, \mathsf{path}_{sk_{t,\mathbf{c}_i}^{\mathrm{COIN}}}$ be the root and path to time t's sub-key in $sk_{\mathbf{c}_i}^{\mathrm{COIN}}$

if $pk_{\mathbf{c}_2}^{\mathrm{COIN}} = \bot \wedge v_2 = 0$ **then**

  let $\mathsf{sn}_{\mathbf{c}_2} \leftarrow \mathsf{prf}^{\mathsf{zdrv}}_{\mathsf{root}sk_{\mathbf{c}_1}^{\mathrm{COIN}}}(\rho_{\mathbf{c}_1})$

  let $\mathsf{root}_{sk_{\mathbf{c}_2}^{\mathrm{COIN}}}, \mathsf{path}_{sk_{t,\mathbf{c}_2}^{\mathrm{COIN}}}, \rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, \mathsf{path}_2 \leftarrow \bot$

else

  let $\mathsf{sn}_{\mathbf{c}_2} \leftarrow \mathsf{prf}^{\mathsf{sn}}_{\mathsf{root}_{sk_{\mathbf{c}_2}^{\mathrm{COIN}}}}(\rho_{\mathbf{c}_2})$

let $\rho_{\mathbf{c}_3} \leftarrow \mathsf{coinId}_3; \rho_{\mathbf{c}_4} \leftarrow \mathsf{coinId}_4$

let $(\mathsf{cm}_{\mathbf{c}_i}, r_{\mathbf{c}_i}) \leftarrow \mathsf{comm}(pk_{\mathbf{c}_i}^{\mathrm{COIN}} \parallel v_i \parallel \rho_{\mathbf{c}_i})$ for $i \in \{3, 4\}$

let $\mathsf{sn}_{\mathbf{c}_1} \leftarrow \mathsf{prf}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$

let root be the transfer Merkle tree root in $\mathcal{C}^{\lceil k}$

let $\mathsf{path}_1, \mathsf{path}_2$ be paths to $\mathsf{cm}_{\mathbf{c}_1}, \mathsf{cm}_{\mathbf{c}_2}$ in root, if still undefined

if $\mathsf{cm}_{\mathbf{c}_1}$ or $\mathsf{cm}_{\mathbf{c}_2}$ were not found in root **then**

  return SUBMIT-GENERIC($\mathsf{tx}_{\mathsf{xfer}}$)

let $x \leftarrow (\{\mathsf{cm}_{\mathbf{c}_3}, \mathsf{cm}_{\mathbf{c}_4}\}, \{\mathsf{sn}_{\mathbf{c}_1}, \mathsf{sn}_{\mathbf{c}_2}\}, t, \mathsf{root})$

let $w \leftarrow (\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}_1}}, \mathsf{path}_{sk^{\mathrm{COIN}}_{t,\mathbf{c}_1}}, \mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}_2}}, \mathsf{path}_{sk^{\mathrm{COIN}}_{t,\mathbf{c}_2}}, pk^{\mathrm{COIN}}_{\mathbf{c}_3}, pk^{\mathrm{COIN}}_{\mathbf{c}_4}, (\mathsf{cm}_{\mathbf{c}_1}, \rho_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v_1, \mathsf{path}_1), (\mathsf{cm}_{\mathbf{c}_2}, \rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, v_2, \mathsf{path}_2), (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_3), (\rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_4))$

send (PROVE, x, w) **to** $\mathcal{F}^{\mathcal{R}_{\mathrm{XFER}}}_{\mathrm{NIZK}}$ and **receive the reply** π

send (ENCRYPT, $\mathsf{pk}^{\mathsf{enc}}_r, t, (pk^{\mathrm{COIN}}_{\mathbf{c}_4}, t, \rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, \mathsf{coinId}_4, v_{\mathbf{c}_4})$) **to** $\mathcal{F}_{\mathsf{FwEnc}}$ and

  receive the reply c_{rcpt}

let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (\{\mathsf{cm}_{\mathbf{c}_3}, \mathsf{cm}_{\mathbf{c}_4}\}, \{\mathsf{sn}_{\mathbf{c}_1}, \mathsf{sn}_{\mathbf{c}_2}\}, \mathsf{root}, t, \pi)$

let $\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}} \leftarrow (\mathrm{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, c_{\mathsf{rcpt}})$

let $\mathsf{log} \leftarrow \mathsf{log} \cup \{(\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}}, \mathrm{SEND}, (\mathsf{pk}^{\mathsf{enc}}_s, (pk^{\mathrm{COIN}}_{\mathbf{c}_1}, \mathsf{coinId}_1, v_1), (pk^{\mathrm{COIN}}_{\mathbf{c}_2}, \mathsf{coinId}_2, v_2), (pk^{\mathrm{COIN}}_{\mathbf{c}_3}, \mathsf{coinId}_3 v_3)))\}$

erase $\mathbf{c}_{1,2}$: $\mathfrak{C} \leftarrow \mathfrak{C} \setminus \{(pk^{\mathrm{COIN}}_{\mathbf{c}_i}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, v_{\mathbf{c}_i}) \mid i \in \{1, 2\}\}$

let $\mathfrak{C}_{\mathsf{cnd}} \leftarrow \mathfrak{C}_{\mathsf{cnd}} \cup \{(pk^{\mathrm{COIN}}_{\mathbf{c}_3}, \rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_{\mathbf{c}_3})\}$

send (BCAST, $\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}}$) **to** $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Net}}$

return $\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}}$

Submitting "generic" transactions is comparatively straightforward:

Protocol SUBMIT-GENERIC(tx)

The generic transaction submission protocol encrypts non-public transactions using the recipient's public key.

On invocation:

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

let $\mathsf{tx}^{\mathsf{real}} \leftarrow$ GENERIC

for stx **in** tx **do**

  if $\mathsf{stx} = (\mathrm{PUBLIC}, m)$ **then**

    let $\mathsf{tx}^{\mathsf{real}} \leftarrow \mathsf{tx}^{\mathsf{real}} \parallel \mathsf{stx}$

  else if $\mathsf{stx} = (\mathsf{pk}^{\mathsf{enc}}_r, m)$ **then**

    send (ENCRYPT, $\mathsf{pk}^{\mathsf{enc}}_r, t, m$) **to** $\mathcal{F}_{\mathsf{FwEnc}}$ and **receive the reply** c

    let $\mathsf{tx}^{\mathsf{real}} \leftarrow \mathsf{tx}^{\mathsf{real}} \parallel (\mathrm{PRIVATE}, c)$

send (BCAST, $\mathsf{tx}^{\mathsf{real}}$) **to** $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Net}}$

return $\mathsf{tx}^{\mathsf{real}}$

5.4.4.4 Reading the State

The last command related to the interaction with the ledger is the read command READ that is used to read the current contents of the state. Note that in the ideal world, the result of issuing such a command is for the ledger to output a (long enough prefix) of the ideal-world state of the ledger, with parts the party does not have access to being hidden. As the format of real-world transactions differs, we need to invert the mapping from real transactions to the corresponding ideal transactions. For generic transactions, this is a little tricky, as the use of forward-secure encryption implies that the information associated with the transaction in the ideal world is erased in the real world. To circumvent this, parties maintain a log, recording information necessary to reconstruct the ideal-world representation of the transaction. The reconstruction process is fully specified as:

Protocol READ-STATE

The read protocol retrieves all information a party can see on their ledger's current chain, without the last k blocks.

State variables and initialisation values:

Variable	Description
$\mathsf{log} \coloneqq \varepsilon$	A log of received transaction information
$\mathcal{C} \coloneqq \bot$	The currently selected chain

On invocation:

run FETCH-INFORMATION
send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
let $\Sigma^{\mathsf{ideal}} \leftarrow \varepsilon$
for tx **in** $\mathcal{C}^{\lceil k}$ **do**
  if $\mathsf{tx} = (\textsc{transfer}, \mathsf{stx}_{\mathsf{proof}}, \mathsf{stx}_{\mathsf{rcpt}})$ **then**
    let $\mathsf{stx}_{\mathsf{chng}} \leftarrow \mathsf{stx}_{\mathsf{rcpt}} \leftarrow \bot$
    if $\exists \mathsf{stx}' : (\mathsf{tx}, \textsc{send}, \mathsf{stx}') \in \mathsf{log}$ **then**
      let $\mathsf{stx}_{\mathsf{chng}} \leftarrow (\psi, \mathsf{stx}')$
    if $\exists \mathsf{stx}' : (\mathsf{tx}, \textsc{receive}, \mathsf{stx}') \in \mathsf{log}$ **then**
      let $\mathsf{stx}_{\mathsf{rcpt}} \leftarrow (\psi, \mathsf{stx}')$
    let $\Sigma^{\mathsf{ideal}} \leftarrow \Sigma^{\mathsf{ideal}} \parallel ((\textsc{public}, \textsc{transfer}), \mathsf{stx}_{\mathsf{rcpt}}, \mathsf{stx}_{\mathsf{chng}})$
  else if $\mathsf{tx} = (\textsc{generic}, \mathsf{stx}_1, \ldots, \mathsf{stx}_n)$ **then**
    let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \varepsilon$

```
        for stx in tx do
            if stx = (⊤, m) then let tx^ideal ← tx^ideal ‖ (⊤, m)
            else if stx = (⊥, c) then
                if ∃m: (PLAINTEXT, c, m) ∈ log then let tx^ideal ← tx^ideal ‖ (ψ, m)
                else let tx^ideal ← tx^ideal ‖ ⊥
    let Σ^ideal ← Σ^ideal ‖ (tx^ideal)
return Σ^ideal
```

5.4.5 Transaction Validity

Transaction validity again differs in the real and ideal world, as the transactions themselves differ. The real-world transaction semantics (which formally runs in FETCH-INFORMATION) maintain three sets, the sets of coin commitments $\mathbb{C}^{\text{spend}}$, $\mathbb{C}^{\text{lead}}$ for spending and leadership, respectively, initialised to the initial set of coin commitments $\mathbb{C}_1$, and the set of spent serial numbers $\mathbb{S}$, initialised to $\varnothing$. A chain is processed transaction by transaction. Leadership transactions and transfer transactions are both validated, other transactions are ignored. A leadership transaction is valid if and only if all leadership transactions in $\vec{\text{stx}}_{\text{ref}}$ are valid adopted leadership transactions and the NIZK proof is valid with respect to the Merkle root of the current tree, with these adopted transactions inserted, as well as η_{ep}, and it has a greater slot number than the previous slot. Furthermore, the serial number sn revealed in it must not be in the current $\mathbb{S}$. The root used must either be the root of the predecessor block, or the root of a past leadership transaction's Merkle tree, with only this transactions commitment added to the tree. Finally, ptr must be the hash of the previous block and h must be the hash of the remaining transactions. After it is successfully validated, $\mathbb{S} \leftarrow \mathbb{S} \cup \{\text{sn}\}$, $\mathbb{C}^{\text{lead}} \leftarrow \mathbb{C}^{\text{lead}} \cup \{\text{cm}\}$, $\mathbb{C}^{\text{spend}} \leftarrow \mathbb{C}^{\text{spend}} \cup \{\text{cm}\}$.

Transfer transactions are likewise validated by checking the NIZK proof with respect to the public transaction component. Furthermore, it is checked that root was at some point the root of $\mathbb{C}^{\text{spend}}$ and that $\{\text{sn}_1, \text{sn}_2\} \cap \mathbb{S} = \varnothing$. If so, the effect is updating $\mathbb{S} \leftarrow \mathbb{S} \cup \{\text{sn}_1, \text{sn}_2\}$ and $\mathbb{C}^{\text{spend}} \leftarrow \mathbb{C}^{\text{spend}} \cup \{\text{cm}, \text{cm}_3\}$. Finally, at the start of an epoch, old enough spending coins are allowed for leadership proofs: $\mathbb{C}^{\text{lead}} \leftarrow \mathbb{C}^{\text{lead}} \cup \mathbb{C}^{\text{spend}}_{t-k}$, where $\mathbb{C}^{\text{spend}}_{t-k}$ is the set of spending coin commitments k slots before the start of the epoch.

If a leadership transaction is included normally in a block, or included

in $\vec{\mathsf{stx}}_{\mathsf{ref}}$ (that is, it is not *this block's* leadership transaction), it is considered an *adopted leadership transaction*. The validity criteria for these are different, requiring only that the proof is valid, the serial numbers are unspent, and the Merkle root was a valid root for $\mathbb{C}^{\mathsf{lead}}$ at some point. The effects of the transaction remain the same, although is is no longer the leader of a block. A block's transactions are validated *prior* to the leadership transaction, as this may depend on adopted leadership transactions. The Merkle tree root of $\mathbb{C}^{\mathsf{lead}}$ of any adopted leadership transactions chain's is saved and preserved. These are valid for other leadership transactions in the same epoch. Specifically, they are also valid for the leadership transaction of the block it is contained in.

5.5 Security Analysis

We split our security analysis of Crypsinous into two parts: In a first part, we show that Crypsinous realises a "non-private" version of $\mathcal{G}_{\mathsf{pLedger}}$ – specifically, we show that it realises $\mathcal{G}_{\mathsf{pLedger}}$ with lkg set to the identity function $\mathsf{lkg}_{\mathsf{id}}$ and $\mathsf{blind}_{\mathcal{A}}(\cdot,\cdot,\mathsf{tx})$ overridden to return tx; that is, the ledger leaks its entire content to the simulator, described in detail during the proof. We argue that the simulator $\mathcal{S}_1$ can simulate any real-world attacks on Crypsinous against a non-private $\mathcal{G}_{\mathsf{pLedger}}$. This first part already proves that our protocol satisfies all the properties of the public ledger, including chain quality, common prefix, and chain growth. In a second part, we argue that in addition to the above, it also satisfies privacy. This is done by instantiating lkg to $\mathsf{lkg}_{\mathsf{lead}}$, in which only the leaders of a given slot are leaked. For this case we provide a simulator $\mathcal{S}_2$ which is able, with access only to this restricted leakage to simulate the outputs of $\mathcal{S}_1$, to generate a view which is indistinguishable from $\mathcal{S}_1$.

5.5.1 Stage 1: Public Proof-of-Stake

Theorem 5.2. Crypsinous, *in the* $(\mathcal{W}^{\mathsf{PoS}}_{\mathsf{HonMaj}}(\mathcal{F}^{\mathcal{R}_{\mathsf{LEAD}}}_{\mathsf{NIZK}}, \mathcal{F}^{\mathcal{R}_{\mathsf{XFER}}}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{FwEnc}}, \mathcal{F}^{\Delta}_{\mathsf{Net}}), \mathcal{F}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{clock}})$*-hybrid world, UC-emulates* $\mathcal{G}_{\mathsf{pLedger}}$ *with* $\mathsf{lkg} = \mathsf{lkg}_{\mathsf{id}}$ *and* $\mathsf{blind}_{\mathcal{A}}(\cdot,\cdot,\mathsf{tx})$ *overridden to return* tx*, under the DDH assumption.*[3]

[3]We will be working under this assumption throughout the rest of the security analysis, and will typically leave it implicit. We will also be assuming the binding (under discrete log, which is implied by DDH) and hiding of our commitments, and the pseudo-randomness of our PRFs implicitly.

Simulator $\mathcal{S}_1$

For simplicity, we assume that $\mathcal{A}$ does not violate the requirements of $\mathcal{W}^{\text{PoS}}_{\text{HonMaj}}$. If it does, simulation is simpler, as the adversary relinquishes the ability to make leadership proofs.

State variables and initialisation values:

Variable	Description
$\mathcal{F}_{\text{Init}}$	Simulated CRYPSINOUS initialisation
$\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$	Simulated leadership NIZK
$\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{XFER}}}$	Simulated transfer NIZK
$\mathcal{F}_{\text{Net}}^{\text{tx}}$	Simulated transaction network
$\mathcal{F}_{\text{Net}}^{\text{bc}}$	Simulated chain network
$\mathcal{F}_{\text{FwEnc}}$	Simulated forward secure encryption
$\mathcal{F}_{\text{CRS}}^{\text{init}_{\text{comm}}} \coloneqq \varnothing$	Simulated commitment CRS
$\text{ek} \coloneqq \bot$	Equivocation key
ϕ_ψ	Simulated protocol for each party $\psi \in \mathcal{P}$
$\text{doneMaint} \coloneqq \varnothing$	Party, time pairs of when maintenance was done
$\mathcal{C}_{\text{last}} \coloneqq \bot$	The current "best" chain
$M \coloneqq \varnothing$	Mapping from ciphertexts to messages
$\text{pks} \coloneqq \varnothing$	Mapping from parties to their public keys
$\text{txs} \coloneqq \varnothing$	Set of all transactions ever submitted

On initialisation:

let $(\mathcal{F}_{\text{CRS}}^{\text{init}_{\text{comm}}}.s, \text{ek}) \xleftarrow{*} \text{simInit}_{\text{comm}}$

When receiving a message (MAINT, ψ) *from* $\mathcal{G}_{\text{pLedger}}$*:*

send READ **to** $\mathcal{G}_{\text{clock}}$ and **receive the reply** t

if $(\psi, t) \notin \text{doneMaint}$ **then**

run ϕ_ψ.LEDGER-MAINTENANCE

let $\text{doneMaint} \leftarrow \text{doneMaint} \cup (\psi, t)$

When receiving a message $(\text{TRANSACTION}, \cdot, \text{tx}, t, \psi)$ *from* $\mathcal{G}_{\text{pLedger}}$*:*

simulate sending $(\text{SUBMIT}, \text{tx})$ **to** ϕ_ψ and **receive the reply** txid

let $\text{txs} \leftarrow \text{txs} \cup \{\text{txid}\}$

return txid

When receiving a message $(\text{GENERATE}, \psi, \text{tag})$ *from* $\mathcal{G}_{\text{pLedger}}$*:*

simulate sending $(\text{GENERATE}, \text{tag})$ **to** ϕ_ψ and **receive the reply** id

if $\text{tag} = \text{ID} \wedge \psi \notin \text{pks}$ **then**

let pks(ψ) ← id

return id

When receiving a message CLAIM *from a corrupted party* ψ *for* $\mathcal{F}_{\text{Init}}$*:*

send MAINTAIN-LEDGER **to** $\mathcal{G}_{\text{pLedger}}$ **on behalf of** ψ

When receiving a message (BCAST, $\mathcal{C}$) *from a party* ψ *for* $\mathcal{F}_{\text{Net}}^{\text{bc}}$*:*

if $\mathcal{C}_{\text{last}} = \bot$ **then**

send GENESIS **to** $\mathcal{F}_{\text{Init}}$ and **receive the reply** $\mathfrak{G}$

let $\mathcal{C}_{\text{last}} \leftarrow \mathfrak{G}$

if $\mathcal{C}_{\text{last}} \prec \mathcal{C}^{\lceil k}$ **then**

let $\Sigma' \leftarrow \Sigma(\mathcal{C}^{\lceil k} \setminus \mathcal{C}_{\text{last}})$

send (EXTEND, Σ') **to** $\mathcal{G}_{\text{pLedger}}$

When receiving a message (TARGET, ψ, $\mathcal{C}$) *from* $\mathcal{A}$ *for* $\mathcal{F}_{\text{Net}}^{\text{bc}}$*:*

send (ADVANCE, ψ, $\Sigma(\mathcal{C}^{\lceil k})$) **to** $\mathcal{G}_{\text{pLedger}}$

Forward all other queries to their simulated functionalities.

Helper procedures:

function $\Sigma(\mathcal{C})$

let $\Sigma' \leftarrow \varepsilon$

for B **in** $\mathcal{C}$ **do**

for tx **in** B **do**

run ensureSubmitted(Σ', tx)

let $\Sigma' \leftarrow \Sigma' \parallel$ tx

return Σ'

function ensureSubmitted(Σ, tx)

if tx ∈ txs **then**

return

let tx′ ← ε

if ∃tx″: tx = (GENERIC, tx′) **then**

let tx′ ← ε

let tx′ ← (PUBLIC, GENERIC)

for c **in** tx′ **do**

if ∃m: c = (PUBLIC, m) **then**

let tx′ ← tx′ ‖ c

else

let tx′ ← tx′ ‖ decrypt(c)

else if $\exists \ldots : \mathsf{tx} = (\textsc{transfer}, (\{\mathsf{cm}_{\mathbf{c}_3}, \mathsf{cm}_{\mathbf{c}_4}\}, \{\mathsf{sn}_{\mathbf{c}_1}, \mathsf{sn}_{\mathbf{c}_2}\}, \mathsf{root}, t, \pi), c_{\mathsf{rcpt}})$ **then**

let $x \leftarrow (\{\mathsf{cm}_{\mathbf{c}_3}, \mathsf{cm}_{\mathbf{c}_4}\}, \{\mathsf{sn}_{\mathbf{c}_1}, \mathsf{sn}_{\mathbf{c}_2}\}, t, \mathsf{root})$

simulate sending $(\textsc{verify}, x, \pi)$ **to** $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$ and **receive the reply** b

if $\neg b \vee \nexists \Sigma' \prec \Sigma \colon \mathsf{root}(\Sigma') = \mathsf{root}$ **then**

continue

let $w \leftarrow \mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}.W((x, \pi))$

determine the corresponding ideal-world coins from w

ensure the change is received by the adversary[4]

if the ciphertext cannot be decrypted and understood correctly by the recipient **then**
modify the recipient to be a new adversarial ID

let tx' be the corresponding ideal-world transfer

send $(\textsc{submit}, \mathsf{tx}')$ **to** $\mathcal{G}_{\mathsf{pLedger}}$

answer the following TRANSACTION query with tx

function decrypt(c)

if $c \in M$ **then return** $M(c)$

else

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

for $\psi \in \mathcal{H}; t' \in t - \Delta_{\mathsf{max}} \ldots t$ **do**

simulate sending $(\textsc{decrypt}, t', c)$ **to** $\mathcal{F}_{\mathsf{FwEnc}}$ and **receive the reply** r

if $\exists m \colon r = (\textsc{ok}, m)$ **then**

let $M(c) \leftarrow (\mathsf{pks}(\psi), m)$

return $M(c)$

return $\perp$

Proof (sketch). The backbone of the proof of Theorem 5.2 is similar to the security proof of Ouroboros Genesis [BGK+18] with some surgical modifications; in particular, in Step 1 we argue that the usage of NIZKs, nonces, and key-private forward-secure encryption, can replace the usage of forward secure signatures, and in Step 2 we argue that the usage of NIZKs and MUPRFs can replace the usage of VRFs in Genesis. In a nutshell, this allows us to argue in Step 3 that leadership transactions in Crypsinous can be used to replace leadership proofs in Genesis. This allows us to leverage the security analysis from Ouroboros Genesis [BGK+18] in Step 4 for proving that Crypsinous implements, at the very

[4] Even if addressed honestly, this is an adversarial transaction, and the honest party will not see the change.

least, a non-private version of the ledger.

Transactions submitted to CRYPSINOUS are pre-processed, before being handled as a Genesis transaction would be and, on reading from the ledger, this pre-processing is partially inverted. This inversion being only partial is what will later be used to establish the privacy properties of CRYPSINOUS. In Step 5, we establish that this pre-processing and post-processing has the same effect as blinding a transaction in the ideal world, and that the validation predicate of CRYPSINOUS – which is run only against pre-processed transactions – is equivalent to its ideal-world counterpart. Finally, in Step 6, we argue that combined, these properties demonstrate realisation of $\mathcal{G}_{\mathsf{pLedger}}$ with $\mathsf{lkg} = \mathsf{lkg}_{\mathsf{id}}$ and overridden $\mathsf{blind}_{\mathcal{A}}$.

Step 1. The security properties guaranteed by $\mathcal{F}_{\mathsf{FwSig}}$ and used in [BGK+18], are those of forward-secure unforgeability, correctness, and authenticity. A proof of LEAD gives the former two properties and a notion of authenticity that is different to $\mathcal{F}_{\mathsf{FwSig}}$, but sufficient for how it is used in [BGK+18]. Non-malleable NIZKs, such as the ones used in our construction, can be interpreted as "signing" their public inputs with the knowledge of a witness [GM17]. In particular, if the witness itself contains a secret key known only to one party, a NIZK over such a witness effectively acts as a signature. In CRYPSINOUS, the usage of sk^{COIN} in the witness for leadership proof effectively acts as a signature over the rest of the block, providing unforgeability and correctness guarantees. Furthermore, as the statement LEAD has the same conditions as a leadership proof in [BGK+18], the desired authenticity property is also satisfied. This is not sufficient to emulate $\mathcal{F}_{\mathsf{FwSig}}$, however using sk_{sl}^{COIN} and $\rho_{\mathbf{c}}$ in the witness rectifies this. As honest parties update both sk_{sl}^{COIN} and $\rho_{\mathbf{c}}$ after the proof, and sk_{sl}^{COIN} and $\rho_{\mathbf{c}}$ are necessary to generate a new proof for the same slot, the adversary will be unable to create leadership proof for past slots. This is effective only so long as sk_{sl}^{COIN} and $\rho_{\mathbf{c}}$ cannot be retrieved from elsewhere. sk_{sl}^{COIN} is generated locally by an honest party, is never communicated by it (except to $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}_{\mathrm{LEAD}}}$, which guarantees its secrecy) and is erased by the honest party in the same slot.

Step 2. The property of VRF provability is directly captured by the correctness of NIZKs, and that of uniqueness is directly captured by non-malleability. Pseudorandomness is directly supplied by the security under malicious key genera-

tion of MUPRFs. Two VRF calls are embedded in the NIZK; the VRF is used to generate the randomness contribution ρ, and the VRF is used to check the target. While in Crypsinous the latter is not publicly revealed, it is still present and is verified by a verification of the NIZK. The NIZK is not as flexible as the VRF, in that it cannot be used to generate arbitrary VRF proofs at any time, however it can still be safely substituted, as the relation is stricter. The NIZK inputs in Ouroboros Crypsinous depend on the coin's secret key, while in Ouroboros Genesis, they depend on the *party's* secret key. As Ouroboros Genesis anticipates parties acting as multiple parties in the protocol, we can simply consider each Crypsinous coin as one Genesis party.

Step 3. A leadership transaction in Crypsinous can be made only if a coin passes the same threshold check as in Ouroboros Genesis. Due to the independent aggregation property of the threshold function, the probability of this happening for a party holding a specific value of (honest) stake is equal in Crypsinous and Genesis. Furthermore, the NIZK ensures the impossibility of creating a leadership transaction *without* winning this election in Crypsinous, while the VRF validation and block validity check enforce the same property in Genesis. The mechanism of "adopted" leadership transaction ensures this property is preserved, even by a party selecting a new local chain.

Due to the equivalent output distribution of VRFs and PRFs in Genesis and Crypsinous, respectively, the randomness contribution ρ is also equivalent.

Step 4. Given we can replace leadership proofs with leadership transactions in the $\mathcal{G}_{\mathsf{Ledger}}$ proof of [BGK+18], the rest of the proof can be carried out the same for Crypsinous. This establishes that, Crypsinous effectively runs an internal ledger. While the transactions posted to this ledger are not directly those posted to Crypsinous itself, we will establish their relationship and that this corresponds directly to the difference between the public and private ledger.

Step 5. Submitted transactions are pre-processed before being sent to the network, and transactions from the network are post-processed on a READ request in Crypsinous. For brevity, we will refer to the former mapping as f and the latter as f_{ψ}^{-1}. We define *consistency* of this mapping to be the following property: $f_{\psi}^{-1} \circ f = \mathsf{blind}(\{\psi\})$ – that is, READ requests return the same as f_{ψ}^{-1} of the READ in

the mapped ledger. Specifically, as the real-world validation predicate already operates on the mapped transactions, this predicate should behave the same as the ideal-world predicate over the original transactions.

For generic transactions this is straightforward: subtransactions addressed to PUBLIC are preserved and not affected by the mapping. Subtransactions addressed to a party ψ are encrypted with $\mathsf{pk}_p^{\mathrm{enc}}$ in the real world, and each party attempts to decrypt them on the inverse mapping. Specifically, subtransactions addressed to any other party will fail to decrypt and be replaced with $\perp$, while subtransactions which are correctly encrypted, will be replaced with (pk_p, M), where M is the originally encrypted plaintext. This matches the behaviour of blind exactly.

Transfer transactions. This leaves us with the consistency of mappings for transfer and leadership transactions. In addition to being standard transactions, transfer transactions induce a stakeholder distribution. They are intrinsically linked with leadership transactions in the real world, so we will consider these as well. The ledgers, both real and ideal, can be read as a sequence of transfer transactions and, in the real world, leadership transactions. We will prove by induction that validity is equivalent in the real and ideal world, as well as that the inverse mapping of the real-world transaction is the ideal transaction. First, we note the induction hypothesis: For every vector of transfer and (in the real world) leadership transactions in the real and ideal worlds, two sets of valid coins are induced: a) The set of valid ideal-world coins, where each coin has a party, ID (which the simulator sets to be the coin's public key $pk_{\mathbf{c}}^{\mathrm{COIN}}$) and value, and b) The set of valid real-world coins, which have the same attributes, as well as an associated coin secret key sk^{COIN}, a nonce $\rho_{\mathbf{c}}$, and a commitment randomness $r_{\mathbf{c}}$. The induction hypothesis is that these sets are equivalent, that is, the ideal set is equal to the real set without the secret key, nonce and randomness, and that in the vector of transactions, the same transfer transactions were considered valid in both worlds.

As a base case, this is guaranteed by $\mathcal{F}_{\mathsf{Init}}$, which creates the same distribution of coins in the real world as was given in the ideal world, selecting random $\rho_{\mathbf{c}}$ and $r_{\mathbf{c}}$ values. In the induction step, we increase the real-world transaction vector by one transaction. There are four cases, depending on whether the transaction is honest or adversarial, and whether it is a transfer or leadership transaction. We

will consider the honest cases first.

Honest leadership. In the case of an honest leadership transaction, the transaction is valid in the real world, as honest parties would not post an invalid transaction. It spends a coin, and recreates a coin of the same value. This is reflected by updating the set of real-world coins by replacing $\rho_{\mathbf{c}}$ and, $r_{\mathbf{c}}$ with new values $\rho_{\mathbf{c}'}$ and $r_{\mathbf{c}'}$. Trivially, this entails the induction claim.

Honest transfers. In the case of an honest transfer transaction, the ideal world transaction is valid iff the two spent coins were the first coins received at an ID owned by the sending party, the transaction is zero-sum and the address of the "change" coin is also owned by the same party. If these conditions do *not* hold, the honest party would ignore the request in the real world. If they do, the honest party is, by induction hypothesis, guaranteed to know the corresponding sk_t^{COIN}, $\rho_{\mathbf{c}}$ and $r_{\mathbf{c}}$-values of the coins that are spent, so it is able to generate a valid transaction and NIZK proof. Afterwards, in the real and ideal world, the coin is removed from the set of valid coins, and the newly created coins are not yet added, but will be added once the transaction has been confirmed. We conclude the induction hypothesis is maintained in an honest transfer transaction.

Adversarial transactions. To consider adversarial transactions, the simulator does not immediately add them to the buffer. Instead, the simulator locally stores them and waits until the adversary has them sufficiently deep in the chain that they must be added to the ideal world state. At this stage, the simulator adds them to the ideal-world buffer and immediately promotes them to the state. This allows the simulator to manage conflicting adversarial transactions, as it simply waits for the adversary itself to resolve the conflict. In particular, transactions attempting to spend the same coin, in either a leadership or transfer transaction, will be conflicting, as they would reveal the same serial number. Once an adversarial leadership transaction is confirmed in the same way, the adversary will control the same updated coins as in the honest case and will be unable to use the old coins again, as the transaction semantics prevents the reuse of the coins' serial numbers.

Adversarial transfers. As the simulator waits until it enters the state, we need only consider sufficiently deep, valid transactions in the real world, and

ensure the simulator can create a corresponding ideal world transaction. The real-world transaction will need to spend two valid coins, which can originate only from corrupted parties. It creates two new coins, addressed to any party, or potentially no party at all, of the same value. This directly corresponds to a legal adversarial transaction in the ideal world and, by induction hypothesis, all coins spent will be unused. The adversary cannot spend honest coins, as it does not know their secret key, with which to create a NIZK proof, cannot spend coins multiple times, as this would invalidly reveal the same serial number twice. Finally, it cannot spend non-existent coins, as it could not provide a Merkle path witness.

Equivalence. We conclude that real and ideal transactions induce the same set of valid coins and are valid in the same cases. The simulator delaying adversarial transactions in the ideal world is not visible to the environment in any way, as the buffer is only seen by the simulator itself and by the ideal-world coin computation (which does not care about the order of adversarial transactions until they enter the state). The set of coins induces a stakeholder distribution, as required by the proof of [BGK+18].

Finally, the inverse mapping of parties' views correspond to their ideal-world views. Specifically, if a party sees *anything* in the ideal world, it is the recipient or sender of a coin. In the former case it needs only to be able to supply $pk_{\mathbf{c}}^{\text{COIN}}$ and $v_{\mathbf{c}}$ in the ideal world – provided the coin has not since been spent. If the transaction was honest, the party will have seen them on decrypting its ciphertext and – if the coin has not been spent – can be found recorded in log. If the transaction is dishonest, either the ciphertext still correctly encrypts the coin, or, if it does not, the ideal transaction would not have been addressed to the honest party, but to the adversary instead. If the party is the sender, it recorded the sending in log and returns the contents from there. We conclude that honest parties' responses to READ requests in the real and ideal worlds match.

Step 6. The private ledger differs primarily from the standard ledger in that it a) applies blind to the output of READ requests, b) leaks less information to the adversary, and c) provides a mechanism for unique ID generation (which are used internally). Difference a) follows directly from the consistency demonstrated in Step 5. Furthermore, we are considering an overly permissive leak-

age predicate, $\mathsf{lkg}_{\mathsf{id}}$, which provides the adversary with the same information it would receive from the standard ledger satisfying b). Finally, Crypsinous allows ID generation; IDs are generated as either PRF outputs of a PRF seeded with a random, secret value, which will lead to unique IDs for honest parties with overwhelming probability. $\mathcal{F}_{\mathsf{FwEnc}}$ public keys, are guaranteed to be unique, while other IDs randomly sample from $\{0,1\}^\kappa$, which has a negligible probability of collision. We conclude that Crypsinous realises $\mathcal{G}_{\mathsf{pLedger}}$ with $\mathcal{S}_1$, under the leakage predicate $\mathsf{lkg}_{\mathsf{id}}$ and $\mathsf{blind}_{\mathcal{A}}(\cdot,\cdot,\mathsf{tx}) = \mathsf{tx}$. □

5.5.2 Stage 2: Private Proof-of-Stake

Theorem 5.3. Crypsinous, *in the* $(\mathcal{W}^{\mathsf{PoS}}_{\mathsf{HonMaj}}(\mathcal{F}^{\mathcal{R}_{\textsc{lead}}}_{\mathsf{NIZK}}, \mathcal{F}^{\mathcal{R}_{\textsc{xfer}}}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{FwEnc}}, \mathcal{F}^{\Delta}_{\mathsf{Net}}), \mathcal{F}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{clock}})$*-hybrid world, UC-emulates* $\mathcal{G}_{\mathsf{pLedger}}$ *with* $\mathsf{lkg} = \mathsf{lkg}_{\mathsf{lead}}$ *under the DDH assumption.*

Simulator $\mathcal{S}_2$

The simulator $\mathcal{S}_2$ behaves like $\mathcal{S}_1$, with a few key differences in how information about transactions is extracted and acted on. Let ℓ_{coin} be the encoded length of coin tuples.

State variables and initialisation values as in $\mathcal{S}_1$.

When receiving a message $(\textsc{transaction}, \cdot, \mathsf{tx}, t, \psi)$ *from* $\mathcal{G}_{\mathsf{pLedger}}$*:*

if $\exists \mathsf{tx}' : \mathsf{tx} = (\textsc{public}, \textsc{transfer}) \,\|\, \mathsf{tx}'$ **then**
  return $\mathsf{simXfer}(\mathsf{tx}', \psi)$
else
  return $\mathsf{simGen}(\mathsf{tx}, \psi)$

When receiving a message $(\textsc{maint}, \psi)$ *from* $\mathcal{G}_{\mathsf{pLedger}}$*:*

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t
send READ **to** $\mathcal{G}_{\mathsf{pLedger}}$ and **receive the reply** $(\cdot,\cdot,L)$
if $(\psi, t) \notin \mathsf{doneMaint}$ **then**
  run $\mathsf{simStake}(\psi, t, L)$
  let $\mathsf{doneMaint} \leftarrow \mathsf{doneMaint} \cup (\psi, t)$

When $\mathcal{A}$ *requests the corruption of* ψ*:*

corrupt ψ
send READ **to** $\mathcal{G}_{\mathsf{pLedger}}$ **through** ψ and **receive the reply** Σ
construct ϕ_ψ.log from Σ

if $\psi \notin \mathcal{F}_{\mathsf{FwEnc}}.K$ **then**

send KEYGEN **to** $\mathcal{F}_{\mathsf{FwEnc}}$ **on behalf of** ψ

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

let $\mathcal{F}_{\mathsf{FwEnc}}.T(\psi) = t + 1$

determine which leadership and transfer transactions were simulated as originating from ψ

disambiguate which coins won which leadership transactions

for unspent coin $\mathbf{c}$ belonging to ψ **do**

if $\mathbf{c}$ was created by an honest party **then**

let $\rho_{\mathbf{c}} \xleftarrow{*} \{0,1\}^{\kappa}$

let t be the time the coin creating transaction was submitted

let $r_{\mathbf{c}} \leftarrow \mathsf{equiv}(\mathsf{ek}, \mathsf{cm}_{\mathbf{c}}, pk_{\mathbf{c}}^{\mathrm{COIN}} \parallel t \parallel \rho_{\mathbf{c}} \parallel v_{\mathbf{c}})$

else

extract $(pk^{\mathrm{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$ by decrypting the corresponding ciphertext

if $\mathbf{c}$ is currently visible to ψ **then**

let $\phi_{\psi}.\mathfrak{C}_{\mathsf{cnd}} \leftarrow \phi_{\psi}.\mathfrak{C}_{\mathsf{cnd}} \cup \{(pk^{\mathrm{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}$

ensure $\phi_{\psi}.\mathfrak{C}_{\mathsf{free}}$, $\phi_{\psi}.\mathfrak{C}_{\mathsf{cnd}}$, and $\phi_{\psi}.\mathfrak{C}$ are consistent with a real execution:

check which coins are confirmed,

move them to $\mathfrak{C}$, and

erase them from $\mathfrak{C}_{\mathsf{cnd}}$

Other behaviour as in $\mathcal{S}_1$.

Helper procedures:

procedure $\mathsf{simXfer}((\mathsf{stx}_{\mathsf{rcpt}}^{\mathsf{ideal}}, \mathsf{stx}_{\mathsf{chng}}^{\mathsf{ideal}}), \psi)$

send READ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

if $\mathsf{stx}_{\mathsf{rcpt}}^{\mathsf{ideal}} = (\bot, \cdot)$ **then**

let $\mathsf{cm} \leftarrow \mathsf{simComm}(\mathsf{ek})$.

query $\mathcal{A}$ **with** $(\textsc{encrypt}, t, \ell_{\mathsf{coin}})$ and **receive the reply** $\mathsf{stx}_{\mathsf{rcpt}}^{\mathsf{real}}$,

satisfying $\mathsf{stx}_{\mathsf{rcpt}}^{\mathsf{real}} \notin \mathcal{F}_{\mathsf{FwEnc}}.M$, else **sampling from** $\{0,1\}^{\kappa}$,

on behalf of $\mathcal{F}_{\mathsf{FwEnc}}$

let $\mathcal{F}_{\mathsf{FwEnc}}.M(\mathsf{stx}_{\mathsf{rcpt}}^{\mathsf{real}}) \leftarrow (\bot, -1, \bot)$

else

let $(\mathsf{pk}_q^{\mathsf{enc}}, (pk^{\mathrm{COIN}}, \cdot, v)) \leftarrow \mathsf{stx}_{\mathsf{rcpt}}$

let $\rho \xleftarrow{*} \{0,1\}^{\ell_{\mathsf{prf}}}$

let $(\mathsf{cm}, r) \leftarrow \mathsf{comm}(pk^{\mathrm{COIN}} \parallel \rho \parallel v)$

simulate sending $(\textsc{encrypt}, \mathsf{pk}_q^{\mathsf{enc}}, t, (pk^{\mathrm{COIN}}, t, \rho, r, v))$ **to** $\mathcal{F}_{\mathsf{FwEnc}}$ and

receive the reply $\mathsf{stx}_{\mathsf{rcpt}}^{\mathsf{real}}$

let $\mathsf{cm}_2 \leftarrow \mathsf{simComm}(\mathsf{ek})$

let $\mathsf{sn}_1, \mathsf{sn}_2 \xleftarrow{*} \{0,1\}^{\ell_{\mathrm{prf}}}$

if $\rho_{\{1,2\}}$ were adversarially generated, and can be read **then**

use ρ_i to compute sn_i instead

let root be the Merkle root of $\phi_\psi.\mathcal{C}^{\lceil k}$

let $x \leftarrow (\{\mathsf{cm}_3, \mathsf{cm}_4\}, \{\mathsf{sn}_1, \mathsf{sn}_2\}, \mathrm{root})$

query $\mathcal{A}$ **with** $(\textsc{prove}, x)$ and **receive the reply** π,

satisfying $\pi \neq \bot \wedge (x, \pi) \notin \mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}_{\mathrm{XFER}}}.\{\Pi, \overline{\Pi}\}$, else **sampling from** $\{0,1\}^\kappa$,

on behalf of $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}_{\mathrm{XFER}}}$

let $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}_{\mathrm{XFER}}}.\Pi \leftarrow \mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}_{\mathrm{XFER}}}.\Pi \cup \{(x, \pi)\}$

let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (\{\mathsf{cm}, \mathsf{cm}_2\}, \{\mathsf{sn}_1, \mathsf{sn}_2\}, \mathrm{root}, t, \pi)$.

let $\mathsf{tx} \leftarrow (\textsc{transfer}, \mathsf{stx}_{\mathsf{proof}}, \mathsf{stx}_{\mathsf{rcpt}}^{\mathsf{real}})$

simulate sending $(\textsc{bcast}, \mathsf{tx})$ **to** $\mathcal{F}_{\mathsf{Net}}^{\mathsf{tx}}$ **on behalf of** ψ

return tx

procedure $\mathsf{simGen}(\mathsf{tx}^{\mathsf{ideal}}, \psi)$

send $\textsc{read}$ **to** $\mathcal{G}_{\mathsf{clock}}$ and **receive the reply** t

let $\mathsf{tx}^{\mathsf{real}} \leftarrow \textsc{generic}$

for stx **in** $\mathsf{tx}^{\mathsf{ideal}}$ **do**

if $\exists \mathsf{pk}_i^{\mathsf{enc}}, m\colon \mathsf{stx} = (\mathsf{pk}_i^{\mathsf{enc}}, m)$ **then**

send $(\textsc{encrypt}, \mathsf{pk}_i^{\mathsf{enc}}, t, m)$ **to** $\mathcal{F}_{\mathsf{FwEnc}}$ **through** ψ and

receive the reply c

let $\mathsf{tx}^{\mathsf{real}} \leftarrow \mathsf{tx}^{\mathsf{real}} \parallel (\textsc{private}, c)$

else if $\mathsf{stx} = (\textsc{public}, m)$ **then**

Let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\textsc{public}, m)$

else if $\exists \ell\colon \mathsf{stx} = (\bot, \ell)$ **then**

query $\mathcal{A}$ **with** $(\textsc{encrypt}, t, \ell)$ and **receive the reply** c,

satisfying $c \notin \mathcal{F}_{\mathsf{FwEnc}}.M$, else **sampling from** $\{0,1\}^\kappa$,

on behalf of $\mathcal{F}_{\mathsf{FwEnc}}$

let $\mathcal{F}_{\mathsf{FwEnc}}.M(c) \leftarrow (\bot, -1, \bot)$

let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\textsc{private}, c)$

simulate sending $(\textsc{bcast}, \mathsf{tx}^{\mathsf{real}})$ **to** $\mathcal{F}_{\mathsf{Net}}^{\mathsf{tx}}$ **on behalf of** ψ

return $\mathsf{tx}^{\mathsf{real}}$

procedure $\mathsf{simStake}(\psi, t, L)$

if $\psi \notin L$ **then return**

let cm $\leftarrow$ simComm(ek); ρ, sn $\xleftarrow{*}$ $\{0,1\}^{\ell_{\text{prf}}}$

query $\mathcal{A}$ **with** (ENCRYPT, t, ℓ_{coin}) and **receive the reply** c,

satisfying $c \notin \mathcal{F}_{\text{FwEnc}}.M$, else **sampling from** $\{0,1\}^{\kappa}$,

on behalf of $\mathcal{F}_{\text{FwEnc}}$

let $\mathcal{C}$, B, h, ptr, $epoch$, sl, root, η_{ep}, μ_ρ, μ_y, and stx_{ref} be defined as in an honest staking protocol execution by ψ

let $x \leftarrow$ (cm, sn, η_{ep}, sl, ρ, h, ptr, μ_ρ, μ_y, root)

query $\mathcal{A}$ **with** (PROVE, x) and **receive the reply** π,

satisfying $\pi \neq \bot \wedge (x,\pi) \notin \mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}.\{\Pi, \overline{\Pi}\}$, else **sampling from** $\{0,1\}^{\kappa}$,

on behalf of $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}$

let $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}.\Pi \leftarrow \mathcal{F}_{\text{NIZK}}^{\mathcal{R}_{\text{LEAD}}}.\Pi \cup \{(x,\pi)\}$

let $\text{stx}_{\text{proof}} \leftarrow$ (cm, sn, ep, sl, ρ, h, ptr, π)

let tx $\leftarrow$ (LEAD, $\vec{\text{stx}}_{\text{ref}}$, $\text{stx}_{\text{proof}}$)

simulate sending (BCAST, tx) **to** $\mathcal{F}_{\text{Net}}^{\text{tx}}$ **on behalf of** ψ

simulate sending (BCAST, $\mathcal{C} \parallel$ (tx, B)) **to** $\mathcal{F}_{\text{Net}}^{\text{bc}}$ **on behalf of** ψ

Proof (sketch). The leakage lkg_{lead} leaks only the leader of any given slot. We utilise a modified version of $\mathcal{S}_1$, which differs only in that it creates simulated transaction instead of real transactions and reconstructs a corrupted party's state when required. In Step 1, we argue that the simulated transactions are indistinguishable from real transactions and, in Step 2, we argue that the reconstructed party state is indistinguishable from a real party's state. Finally, in Step 3, we argue that the simulator $\mathcal{S}_2$ is indistinguishable from $\mathcal{S}_1$, although requiring less leakage from the private ledger functionality. As a result, the same security argument as for $\mathcal{S}_1$ holds with respect to $\mathcal{G}_{\text{pLedger}}$ with restricted leakage.

Step 1. There are three primitives that are simulated in simulated transactions: Commitments, NIZKs, and $\mathcal{F}_{\text{FwEnc}}$ encryptions. Due to the simulation security of NIZKs and the equivocality of the commitments, we know they are indistinguishable from real NIZKs and commitments respectively. For $\mathcal{F}_{\text{FwEnc}}$, the simulator hands the adversary the same information about the plaintext (namely, the length) as the functionality itself, leaving the adversary with no information to distinguish. As transactions consist of these primitives, and the simulator accurately knows the format and originating party of a transaction, it can create

a perfect simulated equivalent of the transaction and broadcast it on behalf of the same party.

Step 2. While the first simulator was effectively running the protocol for real parties, making corruption trivial, $\mathcal{S}_2$ must reconstruct the parties' local state in a way the adversary cannot distinguish from a real execution. Parties maintain four important state variables: the local chain $\mathcal{C}$, the local buffer txBuf, the set of coins $\mathfrak{C}$ (as well as $\mathfrak{C}_{\mathsf{free}}$ and $\mathfrak{C}_{\mathsf{cnd}}$), the log of transfer interactions, and ciphertext to plaintext mappings, log.

Maintaining $\mathcal{C}$ and txBuf is straightforward, as the network interactions directly dictate their contents and the network is not anonymous. This leaves as the only major issues the reconstruction of $\mathfrak{C}$, $\mathfrak{C}_{\mathsf{free}}$, $\mathfrak{C}_{\mathsf{cnd}}$, and log. When a real-world party's corruption is requested, the simulator corrupts the corresponding ideal-world party. This allows the simulator to extract when the party received, transfers in the ideal world, all of which are guaranteed to be unspent, as well as the plaintexts corresponding to the ciphertext of subtransactions addressed to the party.

At these points, a transfer, or generic transaction will have also been made in the real world. This transaction is either a real transaction, in which case the simulator can extract its content from its simulated $\mathcal{F}_{\mathsf{FwEnc}}$. The corrupted party can only be the recipient ψ_r of such transactions (as this is the only party which may read it). There is one commitment in the transaction that is created for a new coin of this party, and one encrypted $\mathcal{F}_{\mathsf{FwEnc}}$ message that encrypts the corresponding secret values used to control it. The simulator randomly samples $\rho_{\mathbf{c}} \xleftarrow{*} \{0,1\}^{\ell}_{\mathsf{prf}}$, and retrieves $pk_{\mathbf{c}}^{\mathrm{COIN}}$ and $v_{\mathbf{c}}$ from the corresponding ideal-world transaction.

As the ideal-world transaction is valid, we know $pk_{\mathbf{c}}^{\mathrm{COIN}}$ must be a valid ID for the corrupted party, in which case the simulator provided it and knows the corresponding secret key $sk_{\mathbf{c}}^{\mathrm{COIN}}$. It then opens the commitment $\mathsf{cm}_{\mathbf{c}}$ to $pk_{\mathbf{c}}^{\mathrm{COIN}} \parallel v_{\mathbf{c}} \parallel \rho_{\mathbf{c}}$, with the opening randomness $r_{\mathbf{c}}$. This allows the simulator to populate $\mathfrak{C}$, $\mathfrak{C}_{\mathsf{free}}$, and $\mathfrak{C}_{\mathsf{cnd}}$ with coins generated by transfer transactions, depending on their stage of confirmation. We further note that the $\mathcal{F}_{\mathsf{FwEnc}}$ ciphertext can now be opened to the appropriate encryption if necessary. Finally log is populated, by recording the corresponding log action for each of these transactions.

This almost completes the simulator, with the exception of how to handle

coins that were used in leadership proofs. Recall that the simulator is aware of which slots the newly-corrupted party was a leader. It is not, however, aware of which coin won in these slots. For each leadership proof of the corrupted party, the simulator computes the probability of each of the party's coins being the winning coin in the given slot, and samples from this distribution a single coin $\mathbf{c}$. It then ensures this coin is appropriately updated – computing $sk_{\mathbf{c}'}^{\text{COIN}} = \text{prf}_{sk_{\mathbf{c}}^{\text{COIN}}}^{\text{evl}}(1)$ and $\rho_{\mathbf{c}'} = \text{prf}_{sk_{\mathbf{c}}^{\text{COIN}}}^{\text{evl}}(\rho_{\mathbf{c}})$, opening $\text{cm}_{\mathbf{c}'}$, the commitment in the corresponding real-world leadership proof to $pk_{\mathbf{c}'}^{\text{COIN}} \parallel v_{\mathbf{c}} \parallel \rho_{\mathbf{c}'}$, with the resulting randomness being $r_{\mathbf{c}}$. This is added to $\mathfrak{C}$, with the preimage being removed. As the adversary cannot find the preimage of $sk_{\mathbf{c}'}^{\text{COIN}}$, or $\rho_{\mathbf{c}'}$, the adversary cannot perform consistency checks involving the previous coin, such as checking serial numbers match what they should.

As the state of the party handed to the simulator is correct, and any sampled value in it is either purely random, or originates from the equivocal commitment scheme, the adversary cannot distinguish the corrupted party's state from the real party's state.

Step 3. We conclude from Theorem 5.2, and our observations in Step 1 and Step 2, combined with the fact that $\mathcal{S}_1$ and $\mathcal{S}_2$ differ only in simulating transactions and corruption, that Theorem 5.3 holds. □

5.6 Performance Estimation

Coin transfers are modelled after Zerocash's [BCG+14] pour transactions. This enables us to reuse much of the existing implementation work invested on optimising the performance critical SNARK operations by the Zcash project in their Sapling upgrade [HBHW18].

Like Zerocash, our transfer transactions pour two old coins into two new coins. In contrast, a leadership transaction only updates a single coin. The additional costs incurred are two evaluations of a PRF to compute ρ_{c_2} and $sk_{\mathbf{c}_2}^{\text{COIN}}$ for updating the coin in a deterministic manner, two evaluations of MUPRF, and one range-proof to determine the winners of the leadership election lottery. We approximate ϕ_f using a linear function as in Bitcoin. The PRF is implemented using a SHA-256 compression function. The MUPRF requires variable base group exponentiations. As we require equivocal commitments, we replace the

SHA-256 coin commitments of Zerocash that require 83,712 constraints with the Pedersen commitments of Sapling [HBHW18] which require only approximately 2,542 constraints. Purely for performance reasons, we also replace the original SHA-256 Merkle tree of Zerocash with the Pedersen hash-based tree used in Sapling.

In total, see Table 5.2, the multiplication count of a leadership SNARK relation is smaller than a transfer relation by about 42K constraints. Furthermore, the number of constraints used by our transfer relations is within a small margin of those used in an equivalent Sapling transfer relation. While we have not focused on optimising this process as Sapling has, by parallelising the NIZK proofs, we emphasise that even unoptimised, Crypsinous would have a proving time only around double that of Sapling.

Primitive	Approx. constraints
SHA-256	27,904
Exponentiation (variable base)	3,252 [HBHW18, page 128]
Hidden range proof	256
Pedersen commitment	1,006 + 2.666 per bit[5]

Table 5.1: Number of multiplicative constraints in SNARK relations

We note in passing that the forward-secure encryption scheme is needed only for transfers and does not affect the SNARK relations we need to prove which is dominating performance. Likewise, the usage of a simulation secure NIZK will increase proving time and proof lengths. Nevertheless, in both cases, the performance penalty is not intrinsic to the POS setting and it would equally affect a POW-based protocol like Zerocash if one wanted to make it simulation-secure in the adaptive corruption setting.

A second performance concern may be the cost of maintaining and updating Merkle trees of secret keys. There is a trade-off here – larger trees are more effort to maintain and use, while smaller ones may have all their paths depleted and hence require a refresh in the sense of moving the funds to a new coin. For a reasonable value of $R = 2^{24}$, this is of little practical concern. Public keys are valid for 2^{24} slots – approximately five years – and employing standard space/time trade-offs, key updates take under 10,000 hashes, with less than 500 KiB storage

[5] `https://github.com/zcash/zcash/issues/2634`

Constraint count	$\mathcal{R}_{\text{XFER}}$	$\mathcal{R}_{\text{LEAD}}$
Check $pk_{\mathbf{c}_i}^{\text{COIN}}$	$2 \times 27,904$	$27,904$
Check $\rho_{\mathbf{c}_2}$, $sk_{\mathbf{c}_2}^{\text{COIN}}$		$2 \times 27,904$
Path for $\text{cm}_{\mathbf{c}_i}$	$2 \times 43,808$	$43,808$
(1 layer of 32)	$(1,369)$	$(1,369)$
Path for $\text{root}_{sk_{\mathbf{c}_i}^{\text{COIN}}}$		$34,225$
(1 layer of 24)		$(1,369)$
(leaf preimage)		$(1,369)$
Check $\text{sn}_{\mathbf{c}_i}$	$2 \times 27,904$	$27,904$
Check $\text{cm}_{\mathbf{c}_i}$	$4 \times 2,542$	$2 \times 2,542$
Check $v_1 + v_2 = v_3 + v_4$	1	
Ensure that $v_1 + v_2 < 2^{64}$	65	
Check y, ρ		$2 \times 3,252$
Check (approx.) $y < \text{ord}(G)\phi_f(v)$		256
Total	$209,466$	$201,493$

Table 5.2: Number of constraints per SNARK statement

requirement. The most expensive part of the process, key generation, still takes less than a minute on a modern CPU.

6 Privacy in Smart Contracts

This chapter is based on "Kachina – Foundations of Private Smart Contracts" [KKK21b], to appear at the 2021 IEEE Computer Security Foundations Symposium, primarily authored by Thomas Kerber, and co-authored by Aggelos Kiayias and Markulf Kohlweiss.

Decentralised computation, as provided by smart contracts is seemingly inherently limited to being entirely non-private, as it encompasses replicated computation. This apparent contradiction can be bypassed through the usage of cryptography – hiding information in plain sight – as has been done in Chapter 5, and as other privacy-preserving smart contract systems describe in Subsection 2.6.3. The approaches these different systems suggest are fractured however – each fitting a niche and solving a part of the larger problem. This larger problem, arbitrary decentralised private computation, is essentially fully distributed and universal, multi-party computation, which with current algorithms is too costly to run at a large scale. The motivating question behind this chapter is then:

> *Is it feasible to achieve a **privacy-preserving** and **general-purpose** smart contract functionality under the same availability and decentralisation characteristics exhibited by Nakamoto consensus?*

This chapter carves out a large class of distributed computations that we express as smart contracts, which we collectively refer to as "Kachina core contracts". In particular, this includes contracts with privacy guarantees, which can be implemented without additional trust assumptions beyond what is assumed for Nakamoto consensus[1]. This class allows us to express the protocol logic of dedicated privacy-preserving, ledger-based protocols such as

[1] The existence of a securely generated common reference string is also required, which by the results of Chapter 4 also reduces to Nakamoto consensus.

Zerocash [BCG+14] as smart contracts. Existing smart contract systems such as Zexe [BCG+20], Hawk [KMS+16], Zether [BAZB19], Enigma [ZNP15], Arbitrum [KGC+18], and zkay [SBG+19] can be expressed, preserving their privacy guarantees, as KACHINA contracts. These protocols mainly rely on either *zero-knowledge* or *signature authentication* for their security. KACHINA is flexible enough to allow contract authors to express each of these systems, together with a concise description of the privacy they afford. It does not supersede these protocols, but rather gives a common foundation on which one can build further privacy-preserving systems.

Distributed ledgers put forth a new paradigm for deploying online services beyond the classical client-server model. In this new model, it is no longer the responsibility of a single organisation or a small consortium of organisations to provide the platform for deploying relevant business logic. Instead, services can take advantage of decentralised, "trustless" computation to improve their transparency and security as well as reduce the need for trusted third parties and intermediaries.

We make four contributions to the area of privacy-preserving smart contracts:

a) We **model** privacy-preserving smart contracts.

b) We **realise a large class** of such contracts.

c) We **enable concurrent interactions** with smart contracts, without compromising on privacy.

d) We demonstrate a general methodology to **efficiently and composably build** smart contract systems.

Combined, they provide a method for both reasoning about privacy in smart contracts, and construct an expressive foundation to build smart contracts with good privacy guarantees upon.

Our model. We provide a universally composable model for smart contracts in the form of an ideal functionality that is parameterised to model contracts both with and without privacy, capturing a broad range of existing systems. The expressiveness and relative simplicity of our model lends itself to further analyses of smart contracts and their privacy. Moreover, existing privacy-preserving

systems benefit from the model as a means to define their security and contrast their security with other systems.

We consider a smart contract to be specified by a transition function Δ and a leakage function Λ, which parameterise the smart contract functionality $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$. Δ models the behaviour of the contract, were it to be run locally or by a trusted party. It is a program that updates a shared state, and has its inputs provided by and outputs returned to, the calling party. $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$ models network, ledger, and contract specific "imperfections" that also exist in the ideal world by interacting with a $\mathcal{G}_{\text{Ledger}}$-GUC functionality [CDPW07] and captures the fundamental ideal-world leakage through the parameterising function Λ.

Some combinations of Δ and Λ are not obviously realisable, in particular the more restricted the leakage becomes. They are able to capture existing smart contract systems however, both privacy-preserving and otherwise. For instance, a leakage function which leaks the input itself corresponds closely to Ethereum [Woo14], while a leakage function returning no leakage makes many transition functions hard or impossible to realise. This chapter focuses on a more interesting middle ground. By defining the ideal behaviour to interact with $\mathcal{G}_{\text{Ledger}}$, we avoid having to duplicate the complex adversarial influence of ledger protocols. We make few assumptions about this ledger, requiring only the common prefix property, and interfaces for submitting and reading transactions to be well defined.

Our protocol. We construct a practical protocol for realising many privacy-preserving smart contracts, utilising only non-interactive zero-knowledge. The primary goal of this protocol is to provide a sufficiently low-level and general purpose basis for further privacy-preserving systems, without requiring the underlying system to be upgraded with each new extension or change. We focus on the Nakamoto consensus setting of a shifting, untrusted set of parties. The protocol's core idea is to separate a smart contract's state into a *shared, on-chain, public* state, and an *individual, off-chain, private* state for each party. Parties then prove in zero-knowledge that they update the public state in a permissible way: That there exists a private state and input for which this update makes sense.

Dealing with concurrency in a privacy-preserving manner. There exists a fundamental conflict between concurrency and privacy that needs to be ac-

counted for to remain true to our objective of providing a smart contract functionality as decentralised as Nakamoto consensus. To illustrate, suppose an ideal smart contract is at a shared private state ω and two parties wish to each apply a function f and g respectively to this state. They wish (in this specific case) the result to be independent of the order of application – i.e. $f(g(\omega)) = g(f(\omega)) = \omega'$. In any implementation of the above in which parties do not coordinate, the first party (resp. the second) should take into account the publicly known encoding $[\omega]$ of ω and facilitate its replacement with an encoded state $[f(\omega)]$ (resp. $[g(\omega)]$) as it results from the application of the desired transition in each case. It follows that the encoded states $[f(\omega)]$, $[g(\omega)]$ must be publicly reconciled to a single encoded state $[\omega']$ which necessarily must leak some information about the transitions f and g. Being able to achieve this type of public reconciliation while retaining some privacy requires a mechanism that enables parties to predict transition conflicts and specify the expected leakage.

We achieve this through the novel concept of *state oracle transcripts*, which are records of which operations are performed on the contract's state, when interacting with it through oracle queries. These allow contract authors to optimise when transactions are in conflict: ensuring minimal leakage occurs while still allowing reorderings. We provide a mechanism for analyzing when reordering transactions is safe with respect to a user's individual private state, by specifying a sufficient condition for when transactions must be declared as dependencies.

Efficient modular construction. Kachina is designed to be deployed at scale: Previous works using zero-knowledge do not explicitly maintain a contract state. If such a state ω was modelled anyway, (for instance, as inputs to these systems), the zero-knowledge proofs involved would scale poorly, with a proving complexity of $\Theta(|\omega|)$ before any computation is performed. A naive approach to state cannot scale to handle systems with a large state – such as a privacy-preserving currency contract, without these being handled as special cases. Our abstracting of state accesses solves this problem.

Regardless of the size of our state, the state is never accessed directly, but only through oracles specified by the contract. As a result the complexity of what must be proven is under the full control of the contract author and can be optimised for. A proving complexity of $\Theta(|\mathcal{T}_\rho| + |\mathcal{T}_\sigma|)$ prior to performing any computation can be expected in Kachina, where $\mathcal{T}_\rho$ is the oracle transcript for

the private state and $\mathcal{T}_\sigma$ is the one for the public state. This constitutes a clear improvement, as the state of smart contracts deployed in practice may be very large, however transcripts, similar to the inputs and outputs of traditional public contracts, are generally short. This increase in efficiency allows us to construct an entire smart contract system, akin to Ethereum [Woo14], *as a* KACHINA *contract* in Section 6.7.

Not all contracts a user wishes to write will directly match the requirements for realising a smart-contract with the KACHINA core protocol. However, our model is sufficiently flexible to allow direct application of the transitivity of UC-emulation to solve this: If the originally specified "objective" contract (Δ, Λ) is not in the class of KACHINA core contracts, the author can find an equivalent (Δ', Λ') which is. The author can provide a proof that $\mathcal{F}_{\text{SC}}^{\Delta',\Lambda'}$ UC-emulates $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$ and, by the transitivity of UC-emulation, can use the KACHINA core protocol to realise (Δ, Λ). We facilitate such proofs by including adversarial inputs and leakages in our model, which allow the simulator limited control over the objective smart contract. This method to develop private smart contracts is illustrated in Figure 6.1. It is further showcased by the implementation of the salient features of Zerocash [BCG+14] as a KACHINA contract in Section 6.5 and the proof that it UC-emulates a much simpler ideal payments contract.

6.1 Technical Overview

We first informally establish our goals and core technical ideas of this chapter. We will discuss each of our contributions in turn and discuss how, combined, they present a powerful tool for constructing privacy-preserving smart contract systems.

Our model. We model smart contracts as *reactive state machines*, which users interact with by submitting transactions to a distributed ledger. A user submits a transaction, with the intention to issue some high-level command to the smart contract, for instance, to cast a vote, or withdraw funds. Once the transaction is confirmed by the distributed ledger, the user obtains information about the results of this high-level command: both whether it has been processed, and any information it may have computed using the contract's state.

As multiple users can interact with the same smart contract system concur-

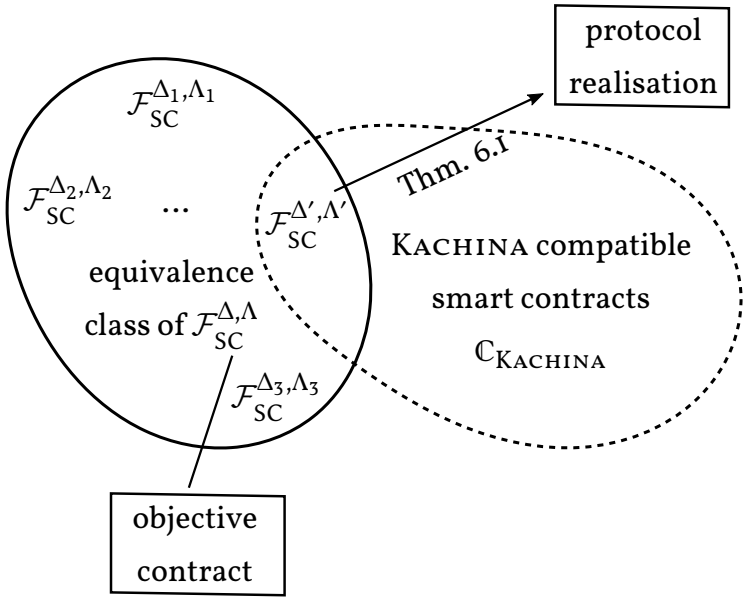


Figure 6.1: An overview of the KACHINA method to develop private smart contracts: 1) An intuitive description of the objective smart contract is developed in the form of $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$. 2) A KACHINA compatible $\mathcal{F}_{\text{SC}}^{\Delta',\Lambda'}$, from the set of all equivalent contracts $\mathcal{F}_{\text{SC}}^{\Delta_i,\Lambda_i}$ is selected and the equivalence proven. 3) Theorem 6.1 is applied to obtain its realisation.

rently, users cannot always predict the effect of their actions; a vote may end before a user's voting transaction is processed, for instance. As a result the user may not be able to predict the outcome of the command, or even if it can be carried out.

To capture privacy, the act of creating a transaction to post on the distributed ledger is the only point at which we permit privacy leakage. As a user may go offline at any point, any private information they reveal – a bid during an opening phase of an auction, for instance – must be revealed in the on-chain transaction itself. Formally, we model this with a *leakage function* Λ, which describes what information is leaked if a user, seeing a specific contract state, issues a specific command. This function can also fix choices that an interaction may make – for instance if the command is "send a coin to Bob", it may decide *which* coin to send to Bob. To give users full control over their privacy, even when these decisions are complex or randomised, we ask them to sign off on a description of the leakage before the transaction is broadcast. The leakage in KACHINA captures information which a user purposely decides to reveal, as the functionality they gain by doing so is worth whatever damage they take to their private information. It

is further worth noting that nothing prevents a malicious contract from finding clever ways to leak information without being observable. This highlights the importance of interacting only with trustworthy contracts and the importance of the leakage descriptor being accurate.

Similarly to the leakage function, the semantics of the contract itself is largely dictated by a *transition function* Δ. It describes how the state of a smart contract evolves given a command and a few auxiliary inputs (such as the choice of coin alluded to above).

The core protocol idea. The KACHINA core protocol restricts itself to contracts which divide their state into a public state σ and, for each party ψ, a private state ρ_ψ. These correspond to the shared ledger and a party's local storage respectively. Transition functions are over pairs (σ, ρ_ψ) instead of over *all* private states – a party may only change their own private state. Honest users maintain their own private state in accordance with the contracts' rules, while the contract must anticipate that dishonest parties may set it arbitrarily (this can be circumvented by committing to private states, as descripted in Subsection 6.6.1, although it comes at the cost of increased public state sizes and loss of anonymity).

A natural construction to achieve privacy in smart contracts utilising zero-knowledge proof systems is apparent: On creating a transaction, a user ψ evaluates the transition function against the current contract state (σ, ρ_ψ), resulting in a state (σ', ρ'_ψ). He creates a zero-knowledge proof that $\sigma \mapsto \sigma'$ is a valid transition of public states (that is, there exists a corresponding private state and input such that this transition takes place) and posts the proof and transition as a transaction. Locally, the user updates his private state to ρ'_ψ.

We can also clearly describe the leakage of this sketched protocol: The transition $\sigma \mapsto \sigma'$ is precisely the information which is revealed!

State oracles. The core protocol sketched above has two major problems:

1. Due to each transaction containing a proof of transition from one state to another, concurrent transactions will almost certainly fail once the state is changed.

2. The size of the statement being proved, and therefore the size of transactions, grows linearly with the overall size of the contract's state.

These drawbacks are especially notable in systems with many users and a high frequency of transactions: On Ethereum a transaction is almost certainly applied after many other transactions the author never knew about, nor should need to know about. The state the contract will be in once it executes a transaction, is something the transaction's author cannot predict accurately. In the naive system proofs only succeed in the state they were originally created for, as Figure 6.2 suggests. Instead of capturing a transition from $\sigma \mapsto \sigma'$, we would rather want to capture a (partial) function from states to successor states.

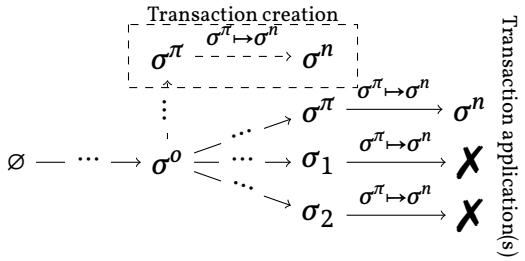


Figure 6.2: Direct state-transition based transactions can be applied only in the state σ^π they were proven for.

To solve these issues, we add a layer of indirection for accessing and updating contract states: Instead of the state being a direct input to the transition function, the contract has access to *oracles* operating on the public and private states. The contract makes queries to these oracles: functions which update the state and return information about it. To prove the interaction with the public state correct, users capture the queries they made and the responses they expect, in a sequence $((q_1, r_1), \ldots, (q_n, r_n))$: a *transcript* of oracle interactions. The user proves that, given the responses expected, they know an input which will make this series of queries.

Conversely, a user validating this transcript can verify this proof and evaluate the queries in turn against the public state, ensuring the responses match. This defines a partial function over public states, which is defined wherever the responses recorded in the transcript match the results obtained by evaluating the queries on the current state.

Selecting what queries a contract makes provides a great deal of control over the domain of the function: a query which has an empty response will always succeed! In limiting queries to returning only essential information, many conflicts can be avoided. Transcripts can also be concise about what changes are

made, assuming the queries are encoded in a sufficiently succinct language, such as most Turing-complete languages.

While not all conflicts are resolved through this as the responses may not match those expected, it allows the proof to focus on the *relevant* parts of the state, being compatible with more concurrent transactions, as pictured in Figure 6.3.

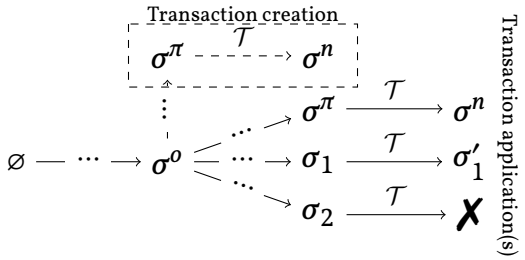


Figure 6.3: Oracle-transcript based transactions can be applied in any compatible state. The transcript $\mathcal{T}$ defines a partial function $\{\sigma^\pi \mapsto \sigma^n, \sigma_1 \mapsto \sigma'_1, \ldots\}$.

In order to be able to model partial transaction success, which is crucial for modelling transaction fees, we allow for a special query to be made, COMMIT. COMMIT queries mark checkpoints in a transaction's execution, such that if an error occurs after it, the execution up to this point is still meaningful. This effectively partitions the transcript into atomic segments. We primarily use this to construct transaction fees within a smart contract itself, the details of which can be seen in Subsection 6.7.5.

High-level usage. Even when using state oracles, this protocol is limited to contracts which have their state fit neatly into accessing only shared public state and local private state. The natural description of many contracts does not match this. For instance: a private currency contract is most directly described through a *shared private* state tracking the balances of all parties.

However, it *is* simple to express the Zerocash [BCG+14] protocol in terms of interactions with shared public and local private states. This provides a practical means to *achieve* what we can *describe* using a shared private state. It is important to have both the most natural description of a contract and the realisation. The former provides a good understanding of the features and security properties of a contract, while the latter realises it.

This idea is nothing but the notion of simulation-based security itself! We use multiple stages of UC-emulation: First moving from our objective contract

(a private payments contract) to a contract within the KACHINA constraints on state (a Zerocash contract), and second moving on to the KACHINA core protocol. Due to the transitivity of UC emulation, we may therefore use this "KACHINA method" to construct the objective of private payments. This process is outlined in Figure 6.1.

Our model is designed to facilitate this usage. Specifically for modelling objective contracts the model allows the adversary to provide an additional adversarial input to each transaction. This input allows the simulator to control some parts of the ideal behaviour similar to the simulator's influence on an ideal functionality, for instance to ensure ideal world addresses match real-world public keys.

6.2 Defining Smart Contracts

Smart contracts are typically implemented as replicated state machines. If a *replicated* state machine is the implementation, the natural model is that of the state machine itself. Inputs are drawn from a ledger of transactions and passed to this state machine.

This definition is unsuitable for privacy-preserving smart contracts: If the state machine's behaviour is known and its inputs are on a ledger, there is no privacy. A simple tweak can solve this: Inputs are replaced with identifiers on the ledger, with the smart contract functionality tracking what their corresponding inputs are.

6.2.1 Interactive Automata Interpretation

Smart contracts are a form of *reactive computation*: Parties supply an input to the contract, the latter internally performs a stateful computation and returns a result to the original caller. The result is returned asynchronously and may depend on interactions with other users. This is quite close to the concept of a trusted third party, although real-world systems have caveats:

- They *leak* information about the computation performed.
- They allow some *adversarial influence*, partly due to relying on the transaction ordering of an underlying ledger.

- They may carry some impure *execution context*: A transaction may depend on what the state is at the time it is created, for instance.

Often when talking about smart contracts, only the "on-chain" component is considered. This is insufficient for privacy, as by its nature, everything on-chain is public. We therefore model the off-chain component of the interaction as well. This can be as simple as placing inputs directly on the ledger, but can involve more complex pre-computation. Even without the need for privacy, the need to model off-chain computation of smart contracts had been observed [CKM+19] and we believe a formal model should account for it.

To represent a contract, we use a transition function, operating over the contract's state. We denote the initial state as $\varnothing$. Transition functions are deterministic, although limited nondeterminism can be simulated by including randomness in the execution context. Notably, such randomness is fixed on transaction creation, allowing the creator to input (potentially biased) randomness, which is subsequently used in the (replicated) execution of the contract's state machine. Potential uses include the creation of randomised ciphertexts or commitments. The transition function will also output if a transaction should be considered "confirmed" or not, with the latter indicating failure or only partial success, which dependant transactions should not build on.

A contract *transition function* Δ is a pure, deterministic function with the format $(\omega', c, y) \leftarrow \Delta(\omega, \psi, w, z, a)$, with the following inputs and outputs:

- The current state ω
- The calling party ψ
- The input w
- The execution context z
- The adversarial input a
- The successor state ω'
- The confirmation state c
- The output y

In addition to the transition function, it is necessary to capture what leakage an interaction with the contract has. The two are separated due to the asynchronous nature of smart contracts – a transaction is made and leaks information before the corresponding transition function is run on-chain.

The leakage is captured by a *leakage function*, which receives the same input, and in addition receives the creating user ψ's "view" ω of the contract as an input. $\omega = (\ell, U_\psi, T, \omega)$ consists of four items: a) The length of ψ's view of the ledger ℓ. b) ψ's unconfirmed transactions U_ψ. c) A map T from $\mathsf{tx} \in U_\psi$ to (ψ, w, z, a, D). These are Δ's inputs and the transaction's dependencies, which we will introduce shortly, D. d) The contract's state according to ψ's view of the ledger, ω.

This "view" may be used to avoid attempting double-spends by selecting a coin to spend which no other unconfirmed transaction uses, for instance. For this purpose the leakage function can also abort by returning $\perp$, refusing to create a transaction. The function returns a leakage value lkg, which is passed to the adversary, a description of the leakage which occurred, desc, a list of transactions to depend on, D, and the context z. While lkg may be arbitrary, it is important that desc provides an accurate and readable description of this leakage. Its primary purpose is to allow parties to decide *not* to go ahead with a transaction if they notice the leakage is more than expected. With complex contracts, anticipating what will be leaked should not be relied upon. The usage of a descriptor highlights that Λ should not be maliciously supplied and facilitates simulation, as shown in Section 6.5.

It is worth emphasising that the leakage discussed in this chapter is deliberate; this is not leakage observed over a network, which can be hard to identify, but is instead information which users accept to reveal. For instance, a leakage in Zerocash [BCG+14] is the length of the ledger at the time a transaction is created, with the security of the protocol guaranteeing that this – but nothing more – is revealed to an adversary.

The list of dependencies D is a list of transactions, which must occur in the same order before the newly created transaction can be applied. This can be used to enforce basic ordering constraints between transactions. Finally, the context z allows information about the state at the time of transaction creation to be passed to the transition function. This may include the current state, unconfirmed transactions, and a source of randomness. Its content is left arbitrary at this point.

A *leakage function* Λ is a pure, non-deterministic function with the format $(\mathsf{desc}, \mathsf{lkg}, D, z) \leftarrow \Lambda(\omega, \psi, w)$, with the following inputs and outputs:

- ψ's contract view ω
- The calling party ψ
- The input w
- The leakage descriptor desc
- The leaked data lkg
- The tx dependencies D
- The context z

We consider the pair (Δ, Λ) to define a smart contract. The ideal world interaction with a smart contract follows the below pattern:

1. A party submits a contract input w.

2. The corresponding context and leakage are computed.

3. The party agrees to the leakage description, or cancels (in the latter case, the transaction never takes place and no information is revealed).

4. The adversary is given (lkg, D) and provides the adversarial input a.

5. The submitting party can retrieve the output of Δ (if any), while other parties can interact with the modified state.

The level of privacy guaranteed depends greatly on the leakage function Λ: A leakage function which returns its input directly as leakage provides no privacy, while one which returns no leakage at all provides almost total privacy (notably the fact some interaction was made is still leaked). By tuning this, the privacy of Ethereum, Zerocash, and everything in between can be captured.

Our model relies on users querying the result of transactions manually – they are not notified of the acceptance of a transaction and can not modify it once made. If a transaction is not yet confirmed by the ledger, the user gets the result NOT-FOUND; if the transaction depends on failed transactions, $\perp$ is returned; and otherwise the result is provided by the contract itself (which may also inform of partial success).

6.2.2 UC Specification

The *ideal smart contract functionality* $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$ captures the notion of a contract as a leaky state machine whose inputs are drawn from a ledger. It is parameterised by the transition function Δ and the leakage function Λ, and it operates in a hybrid world with a global ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$. A candidate for such a ledger is $\mathcal{G}_{\mathsf{SimpleLedger}}$, as introduced in Subsection 2.4.3.1, although any compatible functionality is sufficient. Its privacy guarantees stem from only revealing explicitly leaked data, i.e. lkg, and only allowing the creator of a transaction to access the result.

Functionality $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$

The smart contract functionality $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$ allows parties to query a deterministic state machine determined by Δ and Λ in a ledger-specified order. The exact semantics of the call is subject to adversarial influence, which is provided some leakage, as defined in Λ.

State variables and initialisation values:

Variable	Description
$T \coloneqq \varnothing$	Mapping from transactions to their executing components.
$U_\psi \coloneqq \varepsilon$	Sequence of unconfirmed transactions, for all parties ψ

When receiving a message (POST-QUERY, w) *from an honest party* ψ*:*

let $\Sigma_\psi \leftarrow$ updateState(ψ)
let $\omega \leftarrow (|\Sigma_\psi|, U_\psi, \text{filter}(\lambda(\text{tx}, \cdot) : \text{tx} \in U_\psi, T), \text{execState}(\Sigma_\psi))$
let (desc, lkg, D, z) $\leftarrow \Lambda(\omega, \psi, w)$
if desc $= \bot$ **then**
  return REJECTED
send (LEAK, desc) **to** ψ and **receive the reply** b
if b **then**
  query $\mathcal{A}$ **with** (TRANSACTION, lkg, D) and **receive the reply** (tx, a),
    satisfying $T(\text{tx}) = \bot \wedge \text{tx} \neq \bot$, else **sampling from** $(\{0,1\}^\kappa, \bot)$
  let $T(\text{tx}) \leftarrow (\psi, w, z, a, D); U_\psi \leftarrow U_\psi \parallel \text{tx}$
  send (SUBMIT, tx) **to** $\mathcal{G}_{\text{Ledger}}$ **on behalf of** ψ
  return (POSTED, tx)
else
  return REJECTED

When receiving a message (CHECK-QUERY, tx) *from an honest party* ψ*:*

let $\Sigma_\psi \leftarrow$ updateState(ψ)
if tx $\in \Sigma_\psi$ **then**
  if $T(\text{tx}) = (\psi, \ldots)$ **then**
    return execResult(prefix(Σ_ψ, tx))
  else return $\bot$
else return NOT-FOUND

Helper procedures:

procedure updateState(ψ)
  send READ **to** $\mathcal{G}_{\text{Ledger}}$ **through** ψ and
    receive the reply Σ_ψ
  let $C \leftarrow$ execConfirmed(Σ_ψ)
  let $U'_\psi \leftarrow U_\psi$
  repeat
    let $U_\psi \leftarrow U'_\psi$
    for tx **in** U_ψ **do**

```
            let (..., D) ← T(tx)
            if D ∉ (C ∪ U_ψ) ∨ (D ∩ C) ⋢ Σ_ψ then
                let U'_ψ ← U'_ψ \ {tx}
    until U_ψ = U'_ψ
    return Σ_ψ
procedure execState(Σ)
    let (ω, ·, ·) ← exec(Σ) in return ω
procedure execResult(Σ)
    let (·, y, ·) ← exec(Σ) in return y
procedure execConfirmed(Σ)
    let (·, ·, C) ← exec(Σ) in return C
procedure exec(Σ)
    let ω ← ∅; y ← ⊥; C ← ∅
    for tx in dedup(Σ) do
        if T(tx) = ⊥ then
            query A with (INPUT, tx) and receive the reply x = (ψ, w, z, a, D),
                satisfying ψ ∉ H, else sampling from {NONE}
            if T(tx) = ⊥ then
                let T(tx) ← x
        y ← ⊥
        if T(tx) = NONE then continue
        let (ψ, w, z, a, D) ← T(tx)
        if D \ C ≠ ∅ ∨ D ⋢ Σ then continue
        let (ω', c, y) ← Δ(ω, ψ, w, z, a)
        if ω' ≠ ⊥ then let ω ← ω'
        if c then let C ← C ∪ {tx}
    return (ω, y, C)
```

6.3 The Kachina Protocol

As mentioned in Section 6.1, a naive construction divides a contract's state into a shared public state and a local private state for each party. Specifically, the ideal state ω is defined as the tuple (σ, ρ), where ρ consists of ρ_ψ for each party ψ. A user proves the validity of any public state transition – that there exists a private state

and input, such that this transition takes place. This clearly does not scale well, as it assumes that the ledger state does not change between the submission and processing of a transaction, and requires zero-knowledge proofs about potentially large states – hundreds of Gigabytes in systems like Ethereum [Eth19]!

In reality, a user's query may not be evaluated immediately and the ledger may change drastically in the meantime. Simply proving a direct state transition would lead to a high proportion of queries being rejected. To solve both problems, we require contracts to access their state through a layer of abstraction which both tolerates reordering interactions and allows for more efficient proofs. We further allow for partial transaction success, by introducing *transaction checkpoints*. Our primary purpose for this notion is to be able to capture the payment of transaction fees, such as gas. We detail our approach to do this in Subsection 6.7.5.

6.3.1 State Oracles and Transcripts

We introduce *state oracles* and *state oracle transcripts* to abstract interaction with a contract's state. We choose this abstraction primarily for its flexibility, and many other approaches are possible, such as byte-level memory accesses, or specific data structures such as set of unspent transactions. These can be seen as instances of state oracles. We make use of the notation $[a, b, c]$ to denote a list of a, b, and c, with the concatenation operator $\|$ and the empty list ε. We use the function last to retrieve the last element of a list and $L[i]$ to denote the ith element of the list L.

An example. To better motivate the need to abstract interactions with a contract's state, we will use a representative example smart contract, and discuss how different abstractions of its state will affect it.

Our example is a *sealed bid auction* contract[2], which we assume has some means of interacting with two on-chain assets, one public and one private. These may be constructed similarly as in Section 6.5, however should be holdable and spendable by other contracts. We do not go into detail of this construction; this idea is fleshed out in detail in Zether [BAZB19]. The auction is opened by the *seller* party and multiple *buyer* parties may bid on it. The auction has three

[2] This contract is designed to make a good example, not a good auction – we do not recommend using it as presented.

stages: Bidding, opening, and withdrawing. The auction contract allows for the following interactions:

- At initialisation, the seller transfers ownership of the public asset A to the auction contract.
- In Stage 1, buyers submit their bids, transferring some amount of the private asset B to the auction contract, which remains anonymous.
- In Stage 2, buyers *reveal* their bid. If the buyer's bid exceeds the currently maximum revealed bid, they reveal their committed asset, increase the maximum bid and they record themselves as the winning bidder. Otherwise, they withdraw their bid from the contract without revealing its value.
- In Stage 3, buyers withdraw any assets they own after the auction – either their (losing) bids, or the sold asset (for the highest bidder). The seller withdraws the highest bid, or the original asset if no bids were made.
- In Stage 1 and 2, the seller may advance the stage.

This contract needs to maintain in its state:

- The current stage the auction is in.
- A reference to the asset being sold.
- A set of bids made.
- The winning bid, its value, and who made it, during the reveal phase.
- A set of losing bids, which have not yet been withdrawn, during the reveal phase.
- Privately, a user remembers which bids are theirs and how to reveal them.

Suppose we adopted a naive approach to state transitions, and proved the transitioning from one state to another directly, with no abstraction of any kind. During the bidding phase it is easily possible for multiple users to attempt to bid simultaneously (especially considering the delay until transactions become confirmed by an underlying ledger). In this case, only one of these transactions will succeed – as soon as this transaction changes the state by adding its own bid, the proof of any other simultaneous transaction becomes invalid.

The simple abstraction of byte-level access would allow a buyer and a seller to withdraw concurrently, as their withdrawals affect different parts of the state. It does not do so well in allowing concurrent bids to be made, however. If the set is implemented with a linked list, for instance, two users attempting to add their own bid simultaneously will change the same part of the state: the pointer to the next element.

A smart abstraction should realise that whichever user bids first, the resulting set of bids is the same, even if its binary representation may not be. Even if the order of the interactions matters, a smart abstraction may allow concurrent interactions. When claiming the maximum bid in the auction, Alice may increase it to 5, while Bob may increase it to 7 concurrently. It should not matter to Bob's transaction if the maximum bid is currently 3, or 5 – although Alice's must be rejected if the bid is increased to 7 first.

General-purpose state oracles. The abstraction we propose is that of *programs*. Appending a value to a linked list can be encoded as a program which a) traverses to the end of the current list, b) creates a new cell with the input value, and c) links this from the end of the list. Formally, these programs are executed by a universal machine called a *state oracle* with access to the current (public or private) state α and potentially an additional *context* z.

Definition 6.1. A state oracle $\mathcal{O} = \mathcal{U}(\alpha_0, z)$, given an initial state α_0 and context z, is an interactive machine internally maintaining a state α, a transcript $\mathcal{T}$, and a vector of fallback states $\vec{\alpha}$ (initially set to the input α_0, ε, and $[\alpha_0]$, respectively), which permits the following interactions:

- Given a COMMIT query, set $\vec{\alpha} \leftarrow \vec{\alpha} \parallel [\alpha]$ and append COMMIT to $\mathcal{T}$.
- Given a query q while α is $\perp$, return $\perp$.
- Otherwise, given a query q, compute $(\alpha', r) \leftarrow q(\alpha, z)$. Update α to α', append (q, r) to $\mathcal{T}$ and return r.
- state($\mathcal{O}$) returns $(\vec{\alpha} \parallel [\alpha], \mathcal{T})$.

The context z is empty ($\varnothing$) for state oracles operating on the public state and is used in state oracles operating on the private state for fine-grained read-only access to the state during transaction creation, for instance, to allow private state oracles to read the public state. Specifically, the oracle operating on the private state can read both the public and private states for: a) the confirmed state at

the time the transaction was created (σ^o and ρ^o), and b) the *projected* state, an optimistic state generated at the time the transaction was created by executing all of the user's unconfirmed transactions. It is made of up the pair σ^π and ρ^π. This can be used to make sure new transactions do not conflict with pending ones: Selecting which coin to spend uses the confirmed state to ensure the coin *can be spent* and the projected state to ensure a coin is not *double spent*. The context is also used to provide a source of randomness η to the private state oracle. In total, the context of the private state oracle is $(\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta)$. The context to the public state oracle is empty ($\varnothing$) and we will sometimes omit it.

We say that the oracle *aborts* if it sets its state to $\perp$. The state will then be rolled back to a safe point, specifically to the last COMMIT where the state was non-$\perp$. Looking forward, we will decompose the transition function Δ into three components: An oracle operating on the public state σ, an oracle operating on ψ's private state ρ_ψ, and a "core" transition function Γ. This process is described in detail in Subsection 6.3.4, with an overview of the interactions of Γ with public and private state oracles given in Figure 6.4.

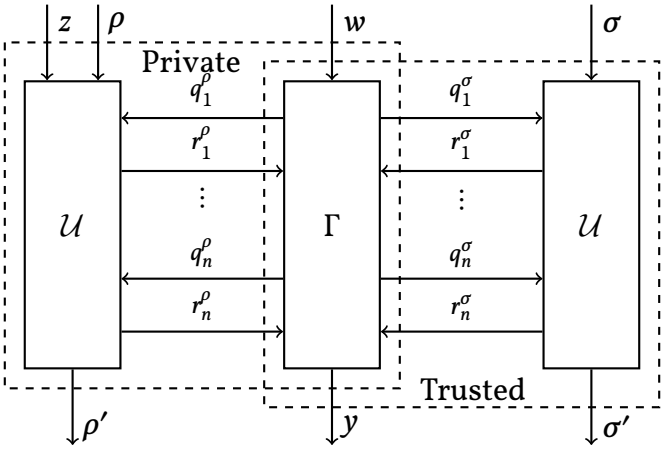


Figure 6.4: The interaction of the core contract Γ, with two universal machines $\mathcal{U}$, acting as state oracles over the public state σ and the private state ρ, together with the context z.

The notion of *oracle transcripts* is crucial in the functioning of KACHINA, as it provides a means to decouple the part of a transaction which is proven in zero-knowledge from both the public and private states entirely: We prove only that given some input and a sequence of responses recorded in the public state transcript, the smart contract must have made the recorded queries.

Revisiting our example. As an illustration, we show how our auction example interacts with state oracles. We define the auction's states more precisely first, where users are identified by public keys, denoted with pk:

- The current stage: $\mathsf{stage} \in \{1, 2, 3\}$
- A reference to the asset being sold and who is selling it: a, pk_s
- A set of bids made: S
- The winning bid, its value, and who made it: b, v, pk_b
- A set of not yet withdrawn losing bids: R
- Privately, a user remembers openings to their bids, the committed bid itself, and its value: $\mathsf{bidOpen}, \mathsf{bidComm}, v$

Overall, the public state is defined as $\sigma := (\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R)$ and the private state is defined as $\rho := (\mathsf{bidOpen}, \mathsf{bidComm}, v)$. The public state is initialised by the seller to $(1, \mathsf{pk}_s, a, \varnothing, 0, \varnothing, \varnothing, \varnothing)$.

The oracle queries corresponding to each interaction with the contract are given as closures, that is, sub-functions which make use of some of their parents local variables. To clarify where this is the case, we place such variables in the subscript of the function name. These functions are passed to either the public or private state oracle as the input q, as specified in Definition 6.1.

- Bidding: Given an asset opening $\mathsf{bidOpen}$, with value v, corresponding to an asset commitment $\mathsf{bidComm}$, which has been bound to the auction contract, Γ first makes the following public oracle query:

  function $\mathsf{makeBid}_{\mathsf{bidComm}}((\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S, R))$
  assert $\mathsf{stage} = 1$
  return $((\mathsf{stage}, \mathsf{pk}_s, a, b, v, \mathsf{pk}_b, S \cup \{\mathsf{bidComm}\}, R), \top)$

  Further, it makes the following private oracle query:

  function $\mathsf{recordBid}_{\mathsf{bidOpen},\mathsf{bidComm},v}(\cdot, \cdot)$
  return $((\mathsf{bidOpen}, \mathsf{bidComm}, v), \top)$

- Revealing: Given a public key to redeem the funds in case of losing the auction, Γ first makes a private oracle query to retrieve which bid is owned:

  function $\mathsf{retrieveBid}((\mathsf{bidOpen}, \mathsf{bidComm}, v), \cdot)$
  return $((\mathsf{bidOpen}, \mathsf{bidComm}, v), (\mathsf{bidOpen}, \mathsf{bidComm}, v))$

  Next, the contract makes a further private oracle query for the expected maximum bid, to determine if the buyer's bid is higher:

function projMax$(\rho, z = (\cdot, \cdot, \sigma^{\pi} = (\ldots, v', \ldots), \cdot, \cdot))$
  return (ρ, v')

If this query returns $v' < v$, the contract attempts to claim the maximum bid with the public oracle query[3]:

function claimMax$_{\text{bidOpen,bidComm},v,\text{pk}}(\sigma)$
  let $(\text{stage}, \text{pk}_s, a, b_o, v_o, \text{pk}_o, S, R) \leftarrow \sigma$
  assert $\text{bidComm} \in S \wedge v > v_o \wedge \text{stage} = 2$
  return $((\text{stage}, \text{pk}_s, a, \text{bidOpen}, v, \text{pk}, S \setminus \{\text{bidComm}\}, R \cup \{(b_o, \text{pk}_o)\}), \top)$

If the original value test fails, on the other hand, instead the contract transfers the ownership of bidComm via the underlying asset system to pk and runs the public oracle query:

function claimLoss$_{\text{bidComm}}((\text{stage}, \text{pk}_s, a, b, v_o, \text{pk}_o, S, R))$
  assert $\text{bidComm} \in S \wedge \text{stage} = 2$
  return $(\top, (\text{stage}, \text{pk}_s, a, b, v_o, \text{pk}_o, S \setminus \{\text{bidComm}\}, R))$

- Withdrawing: Given a public key pk, which the caller knows the corresponding secret key for, the contract will make an oracle query to determine which assets to transfer ownership of and to un-record them in a public oracle query:

  function withdraw$_{\text{pk}}((\text{stage}, \text{pk}_s, a, b, v, \text{pk}_b, S, R))$
    assert $\text{stage} = 3$
    if $\text{pk} = \text{pk}_s \wedge b \neq \varnothing$ **then**
      return $((\text{stage}, \varnothing, a, \varnothing, \varnothing, \text{pk}_b, S, R), (B, b))$
    else if $\text{pk} = \text{pk}_b \wedge a \neq \varnothing$ **then**
      return $((\text{stage}, \text{pk}_s, \varnothing, b, v, \varnothing, S, R), (A, a))$
    else if $\exists c\colon (c, \text{pk}) \in R$ **then**
      return $((\text{stage}, \text{pk}_s, a, b, v, \text{pk}_b, S, R \setminus \{(c, \text{pk})\}), (B, c))$

- Advancing the stage: The seller (given their public key pk) may advance the contracts stage from 1 or 2 to 2 or 3, respectively, with a public oracle query:

  function advanceStage$_{\text{pk}}((\text{stage}, \text{pk}_s, a, b, v, \text{pk}_b, S, R))$
    assert $\text{pk} = \text{pk}_s \wedge \text{stage} \in \{1, 2\}$
    return $((\text{stage} + 1, \text{pk}_s, a, b, v, \text{pk}_b, S, R), \top)$

[3]Note that the claim may fail if the maximum bid increased from the one projected at the time of transaction creation.

This example does not handle corner cases (such as buyers bidding multiple times) and is not intended for practical use. Instead, its purpose is to illustrate the advantages state oracles provide: The query an interaction will make and the response it will receive, are often not affected by other interactions. Concurrent bids do not conflict, for instance. The representation of data is also not crucial, as the state oracles may themselves interact with abstract data types.

We complete our example by specifying the core transition function Γ, under the assumptions that a means to call into a separate asset management system (a contract that permits transferring ownership of assets between public keys), such as presented in Subsection 6.7.4, exists. We also assume that a user's public key can be retrieved with a shared "identity" contract.

Transition Function Γ_{auction}

A simple private auction contract.

When receiving an input (BID, v)*:*

send (BIND, v, Γ_{auction}) **to** Γ_B and
  receive the reply (bidOpen, bidComm, v)
send $\text{makeBid}_{\text{bidComm}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** $\top$
send $\text{recordBid}_{\text{bidOpen,bidComm},v}$ **to** $\mathcal{O}_\rho$ and
  receive the reply $\top$

When receiving an input REVEAL*:*

send retrieveBid **to** $\mathcal{O}_\rho$ and
  receive the reply (bidOpen, bidComm, v)
send IDENTITY **to** Γ_{id} and **receive the reply** pk
send projMax **to** $\mathcal{O}_\rho$ and **receive the reply** v'
if $v' < v$ **then**
  send (ASSERTVALIDFOR, bidOpen, bidComm, v, pk, Γ_{auction}) **to** Γ_B
  send $\text{claimMax}_{\text{bidOpen,bidComm},v,\text{pk}}$ **to** $\mathcal{O}_\sigma$ and
    receive the reply $\top$
else
  send (UNBIND, bidOpen, pk) **to** Γ_B
  send $\text{claimLoss}_{\text{bidComm}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** $\top$

When receiving an input WITHDRAW*:*

send IDENTITY **to** Γ_{id} and **receive the reply** pk

send $\mathsf{withdraw}_{\mathsf{pk}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** (X, x)
if $X = A$ **then**
send $(\textsc{transfer}, x, \mathsf{pk})$ **to** Γ_A
else
send $(\textsc{unbind}, x, \mathsf{pk})$ **to** Γ_B

When receiving an input ADVANCE-STAGE*:*

send IDENTITY **to** Γ_{id} and **receive the reply** pk
send $\mathsf{advanceStage}_{\mathsf{pk}}$ **to** $\mathcal{O}_\sigma$ and **receive the reply** $\top$

Using transcripts. KACHINA relies on a few key observations on how transcripts relate to the original state oracle execution. To begin with, we define a few ways in which transcripts may be used.

Definition 6.2. A state oracle transcript $\mathcal{T}$ may be applied to a state α in a context z. We write $\vec{\alpha} \leftarrow \mathcal{T}(\alpha, z)$, or if $z = \varnothing$, $\vec{\alpha} \leftarrow \mathcal{T}(\alpha)$, the operation of which is defined through the following loop:

function $\mathcal{T}(\alpha, z)$
let $\mathcal{O} \leftarrow \mathcal{U}(\alpha, z)$
for (q_i, r_i) **in** $\mathcal{T}$ **do**
send q_i **to** $\mathcal{O}$ and **receive the reply** r
if $r \neq r_i$ **then return** $\bot$
let $(\vec{\alpha}, \cdot) \leftarrow \mathsf{state}(\mathcal{O})$
return $\vec{\alpha}$

If a transcript is malformed (that is, cannot be parsed into a sequence of query and response pairs), applying it will result in $[\alpha, \bot]$.

Observe that the application of a transcript mimics the execution of the original oracle, diverging only if it returns $\bot$ at some point. This allows users to replicate the effect other users' queries have on the public state, without knowing *why* these queries were made.

Lemma 6.1. *For all α, z, $\mathcal{T}$, where:*

let $\mathcal{O} \leftarrow \mathcal{U}(\alpha, z)$
for $(q, \cdot)$ ***in*** $\mathcal{T}$ ***do***
send q ***to*** $\mathcal{O}$
let $(\vec{\alpha}, \cdot) \leftarrow \mathsf{state}(\mathcal{O})$

let $\vec{\alpha}' \leftarrow \mathcal{T}(\alpha, z)$

Any prefix of $\vec{\alpha}'$ *not containing a* $\perp$ *will match the same length prefix of* $\vec{\alpha}$*.*

Definition 6.3. A sequence of transcripts and contexts $X = ((\mathcal{T}_1, z_1), \ldots, (\mathcal{T}_n, z_n))$ is applied by applying each transcript in order. We write $\mathcal{T}^*_X(\alpha)$, which has the recursive definition:

- $\mathcal{T}^*_\varepsilon(\alpha) \coloneqq \alpha$
- $\mathcal{T}^*_{X \parallel [(\mathcal{T},z)]}(\alpha) \coloneqq \mathcal{T}^*_X(\mathsf{last}(\mathcal{T}(\alpha, z)))$

Definition 6.4. A transcript $\mathcal{T} = ((q_1, r_1), \ldots, (q_n, r_n))$ (potentially including COMMIT messages) induces a transcript oracle $\mathcal{O}(\mathcal{T})$, which behaves as follows:

- Recorded COMMIT messages are ignored.
- For the ith query q'_i, return r_i if $q'_i = q_i$, otherwise abort by returning $\perp$ for this and all subsequent queries.
- When $\mathsf{consumed}(\mathcal{O})$ is queried, return $\top$ if exactly n queries were made, otherwise return $\perp$.

If in an interaction with the oracle, consumed holds, the transcript was *minimal* for this interaction.

If the transcript oracle $\mathcal{O}(\mathcal{T})$ does not abort when used as an oracle in some function, then it behaves identically to the original universal oracle that was used to generate the transcript. We use this fact to generate zero-knowledge proofs about transactions – we prove that each oracle query in a transcript was made and that the behaviour is correct, *given the responses the transcript claims*. We also prove that $\mathsf{consumed}(\mathcal{O})$ holds, ensuring the transcript does not just start with the queries an honest execution would make, but that it matches them exactly.

These are used together to define how a transaction is made and how it is applied: Alice generates a transcript for the oracle accesses her transaction will perform and proves this transcript both correct and minimal. She sends the transcript and proof to Bob, who is convinced by the proof of correctness and minimality, and can therefore reproduce the effect of the transaction by applying the transcript to the state directly.

Inherent conflicts. Abstracting the interaction with the state has many benefits, but it is not a panacea. Some conflicts are inherent and unavoidable – a contract may operate on a first-come first-serve basis, and no trick will ease the pain of coming second. A contract may also simply be badly designed, not making good use of the abstractions provided – at the most extreme, it can make only queries retrieving or setting the entire state, negating all benefit of using oracles.

6.3.2 Interaction between Smart Contracts

The example in Subsection 6.3.1, makes the natural assumption (in the setting of smart contracts), of being able to interact with other components – in this case with an asset system. Most interesting applications of smart contracts seem premised on such interactions. We consider how multiple contracts may interact in Subsection 6.7.3, however we stress that a full treatment is left as future work.

In particular, how various contracts can be independently proven secure and composed in a general system alongside other, potentially malicious contracts, is not handled in this chapter. Instead, where we assume interaction, we limit ourselves to a closed smart contract system with a small set of non-malicious contracts, such as the auction contract and the asset system in Subsection 6.3.1.

While it is tempting to delegate such interactions to the native compositionality and interactiveness of UC, this does not reflect the reality of smart contract interactions, where the executions of multiple contracts are atomically intertwined. While related issues of interaction with the environment have been considered in the literature, for instance in [CEK+16, CDT19], they do not fully address our scenario, in which multiple branches can be executed in projection. We therefore believe that studying the interaction and composition of smart contract transition and leakage functions requires further work, with this work providing a foundation.

6.3.3 The Challenge of Dependencies

If a transaction tx_1 moves funds from Alice to Bob and tx_2 moves funds from Bob to Charlie, the order $\mathsf{tx}_2 \ldots \mathsf{tx}_1$ may not be valid, if tx_2 relies on the funds Bob received from Alice. When a dependency like this is violated in interacting with the public state, attempting to apply the dependent transaction first will fail and

the transaction is rejected.

How such interactions affect a user's private state is more tricky to handle. While two different parties cannot conflict with each other on private state changes due to domain separation, parties may encounter *internal* dependencies.

A party starting with the private state ρ_1, makes a transaction tx_1 which advances their private state to ρ_2. Afterwards, they make the transaction tx_2, their private state ending up as ρ_3. If these transactions are made shortly after each other, tx_2 may be placed before tx_1 on the ledger. It is possible that tx_2 uses information from tx_1, such as a secret key, and that it makes no sense without it.

Should a user ignore the reordering and stick with the state ρ_3? This can introduce inconsistencies between the public state and private state. Should the user apply the private state transcript of tx_2 and hope for the best – but risk a catastrophic failure if it cannot be applied? Neither are ideal. Instead, we propose that tx_2 should publicly declare that it depends on tx_1, and rely on on-chain validation to ensure they are applied in the correct order.

If a user has a set of unconfirmed transactions U and is adding the new transaction tx in the ledger state, dependencies should ensure that any permutation of $U \cup \{\mathsf{tx}\}$ results in a consistent interaction with the user's private state – that is, result in a non-$\bot$ private state. Furthermore, this should even be the case if these transactions are only partially successful – regardless as to which COMMIT point was reached.

An overeager approach would be to ensure all unconfirmed transactions are dependencies, and are in the order that they were made. With domain separation and sufficiently abstract interactions it is likely that only few transactions actually depend on each other. This can be application specific, and to account for this we allow for contracts to specify a function dep to declare dependencies. We constrain how this function may behave, and provide the all-purpose fallback of all unconfirmed transactions.

For most practical cases that we have observed, private state oracles do not conflict or enter into complex dependencies with each other. Most often, their state management is simple: sampling and storing secrets. The formal machinery presented in this section is to allow this intuition that the transactions do not depend on each other to be justified in many cases.

Formal definition. The formal definition of dependency functions is complex; we begin by introducing some mathematical notations. In addition to this notation, we make use of the following functions: a) the higher-order function map. b) an index function, which returns the index of an element in a list, idx. c) the tuple projection functions proj_i, which return the ith element of a tuple. d) the list flattening function flatten, which, given a list of lists, returns a list of the inner lists concatenated. e) the function take, which returns the prefix of a list containing a specified number of items. f) the function zip, which combines n lists into a list of n-tuples.

Definition 6.5. For any finite set X, S_X is the set of all permutations of X, where each permutation is represented as a list.

Definition 6.6. The *subsequence relation* $X \sqsubseteq Y$ indicates that each element of the list X is present in Y, in the same order:

$$X \sqsubseteq Y \coloneqq X \subseteq Y \wedge (\forall a, b \in X.\, \text{idx}(X, a) < \text{idx}(X, b) \implies \text{idx}(Y, a) < \text{idx}(Y, b))$$

We define an expansion of transactions into useful components: As a transaction has no private data within it, we use this to refer to this data.

Definition 6.7. A transcript $\mathcal{T}$'s corresponding *commit-separated* transcript $\vec{\mathcal{T}}$ is a list of lists of query/response pairs, corresponding to splitting $\mathcal{T}$ at each COMMIT. We write $\vec{\mathcal{T}} = \text{split}(\mathcal{T}, \text{COMMIT})$.

Definition 6.8. A secret-expanded transaction is a tuple $(\tau, \vec{\mathcal{T}}, z, D)$, consisting of the transaction object τ, the commit-separated private state transcript $\vec{\mathcal{T}}$, the context z, and the dependencies D.

We define the format of transactions handled by the dependency function. We make use of "confirmation depth", the vector of which is denoted $\vec{c}$. This is a vector of natural numbers, representing how many parts of the corresponding commit-separated transcript executed successfully.

Definition 6.9. A list X of secret-expanded transactions' dependencies may be *satisfied* given a set of still unconfirmed transaction U and a list of confirmation depths $\vec{c}$, denoted by $\text{sat}(X, \vec{c}, U)$, which is defined formally below. Informally, it states that each transaction in X must be preceeded by its dependencies, in

order, and that each of these dependencies should have executed fully, rather than partially.

- $\mathsf{sat}(\varepsilon, \vec{c}, U) \coloneqq \top$
- $\mathsf{sat}(X \parallel (\cdot,\cdot,\cdot,D), \vec{c} \parallel \cdot, U) \coloneqq \mathsf{sat}(X, \vec{c}, U) \wedge (D \cap U) \sqsubseteq \mathsf{map}(\mathsf{proj}_1, X) \wedge \forall d \in D, \vec{\mathcal{T}}, z, D', i\colon (d, \vec{\mathcal{T}}, z, D') = X[i] \implies |\vec{\mathcal{T}}| = \vec{c}[i]$

We write $\mathsf{sat}^*(X, U)$ as a shorthand for the case where $\vec{c}$ are maximal, that is, $\vec{c}[i] = |\mathsf{proj}_2(X[i])|$.

We define what transcripts will actually be executed for a given sequence of confirmation levels.

Definition 6.10. The *effective sequence of transcripts* (denoted $ES(X, \vec{c})$), given a list of secret-expanded transactions and a list of confirmation depths of equal length, is the sequence of confirmed transcript parts, along with their contexts, defined as:
$ES(X, \vec{c}) \coloneqq \mathsf{flatten}(\mathsf{map}(\lambda((\cdot, \vec{\mathcal{T}}, z, \cdot), c)\colon \mathsf{map}(\lambda\mathcal{T}\colon (\mathcal{T}, z), \mathsf{take}(\vec{\mathcal{T}}, c)), \mathsf{zip}(X, \vec{c})))$

We write $ES^*(X)$ as a shorthand for the case where $\vec{c}$ are maximal: i.e. $\mathsf{proj}_i(\vec{c}) = |\mathsf{proj}_2(\mathsf{proj}_i(X))|$.

We define the central invariant the dependencies must preserve: The private state can always be advanced.

Definition 6.11. The dependency invariant $J(X, \rho)$, given a set X of secret-expanded transactions, states that any permutation of a subset of X's private state transcripts which have their dependencies satisfied can be successfully applied to ρ. $J(X, \rho) \coloneqq \forall Y \subseteq X, Z \in S_Y, \vec{c}\colon \mathsf{sat}(Z, \vec{c}, \mathsf{map}(\mathsf{proj}_1, X)) \implies \mathcal{T}^*_{ES(Z,\vec{c})}(\rho) \neq \bot$

Finally, we define the constraints on the dependency function.

Definition 6.12. A dependency function $\mathsf{dep}(X, \mathcal{T}, z)$ is a pure function taking as inputs a set of secret-expanded unconfirmed transactions X, a new private state transcript $\mathcal{T}$, and a new context z, returning a list of transaction objects. It must satisfy the following conditions:

1. If called with non-honestly generated transcripts or contexts, no constraints need to hold.

2. The result must be a subsequence of the transactions in X: $\mathsf{dep}(X, \mathcal{T}, z) \sqsubseteq \mathsf{map}(\mathsf{proj}_1, X)$

3. When adding a new transaction tx, with the corresponding private state transcript $\mathcal{T}$ (where its commit-separated form is $\vec{\mathcal{T}}$) and context z, the dependency invariant J is preserved: $\textbf{let } Y = X \parallel (\mathsf{tx}, \vec{\mathcal{T}}, z = (\cdot, \rho^o, \cdot, \cdot, \cdot), \mathsf{dep}(X, \mathcal{T}, z))$ $\textbf{in } \mathcal{T}^*_{ES^*(Y)}(\rho^o) \neq \bot \wedge J(X, \rho^o) \implies J(Y, \rho^o)$

The dependency function $\mathsf{dep}(X, \mathcal{T}, z) = \mathsf{map}(\mathsf{proj}_1, X)$ can always be used, as it maximally constraints the possible permutations which satisfy dependencies.

6.3.4 The Contract Class

The core Kachina protocol can realise a class of smart contracts, with each contract being primarily defined by a restricted transition function Γ. This transition function is given oracle access to the calling user's private state ρ_ψ and the shared public state σ, as described in Definition 6.1. In addition to these oracle accesses, Γ can make $(\textsc{commit}, y)$ queries, which a) send commit to both oracles, and b) record the value y in a vector of partial results $\vec{y}$. We write $\vec{y} \leftarrow \Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)$ as running the transition function against input w, with oracles $\mathcal{O}_\sigma$ and $\mathcal{O}_\rho$, returning the vector of partial results $\vec{y}$. The final output of Γ is appended to $\vec{y}$ when it terminates. The adversary can program its own private state oracle – it corresponds to local computation, after all! Two minor functions are also used to define the corresponding ideal contract:

- The leakage descriptor desc, which receives the time t, the sequence of secret-expanded unconfirmed transactions X, transcripts $\mathcal{T}_\sigma, \mathcal{T}_\rho$, original input w, and context z of new transactions as inputs and returns a description of what leakage this interaction will incur.

- A dependency function dep satisfying Definition 6.12.

Definition 6.13. $\mathbb{C}_{\text{Kachina}}$ is the set of all pairs $(\Delta_{\text{Kachina}}(\Gamma), \Lambda_{\text{Kachina}}(\Gamma, \mathsf{desc}, \mathsf{dep}))$, for any parameters Γ, desc and dep, satisfying Definition 6.12.

Δ_{Kachina} and Λ_{Kachina} operate as follows; we assume the set of honest parties $\mathcal{H}$ – in the ideal world, this is known by the functionality, while in the real world we assume each party ψ will use $\mathcal{H} = \{\psi\}$.

Transition Function $\Delta_{\text{KACHINA}}(\Gamma)$

The KACHINA transition function, running an internal transition function Γ with oracle access to the public contract state and the private state of the party making the query. The query has an associated context z, which the private state oracle may access, and an associated public state transcript $\mathcal{T}_\sigma$, which must be consistent with the current public state in order for the query to run successfully.

When receiving an input $((\sigma,\rho),\psi,w,(\mathcal{T}_\sigma,z),\cdot)$*:*

let $(\vec{\sigma},\cdot,\vec{\rho},\cdot,\vec{y}) \leftarrow \text{run-}\Gamma(\sigma,\rho[\psi],w,z,\psi\in\mathcal{H})$
let $\sigma' \leftarrow \sigma; y \leftarrow \bot; \vec{\mathcal{T}} \leftarrow \text{split}(\mathcal{T}_\sigma, \text{COMMIT}); C \leftarrow \top$
for $(\mathcal{T}',\sigma'',\rho',y')$ **in** $\text{zip}(\vec{\mathcal{T}},\vec{\sigma},\vec{\rho},\vec{y})$ **do**
  let $\sigma' \leftarrow \mathcal{T}'(\sigma')$
  if $\sigma' = \bot \vee \rho' = \bot \vee \sigma' \neq \sigma''$ **then**
    let $C \leftarrow \bot$
    break
  let $\sigma \leftarrow \sigma''; \rho[\psi] \leftarrow \rho'; y \leftarrow y'$
return $((\sigma,\rho),C,y)$

Helper procedures:

function $\text{run-}\Gamma(\sigma,\rho,w,z,h)$
  $\mathcal{O}_\sigma \leftarrow \mathcal{U}(\sigma,\varnothing); \mathcal{O}_\rho \leftarrow \mathcal{U}(\rho,z)$
  if $\neg h$ **then let** $\mathcal{O}_\rho \leftarrow z$
  $\vec{y} \leftarrow \Gamma_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(w)$
  $(\vec{\sigma},\mathcal{T}_\sigma) \leftarrow \text{state}(\mathcal{O}_\sigma); (\vec{\rho},\mathcal{T}_\rho) \leftarrow \text{state}(\mathcal{O}_\rho)$
  return $(\vec{\sigma},\mathcal{T}_\sigma,\vec{\rho},\mathcal{T}_\rho,\vec{y})$

Leakage Function $\Lambda_{\text{KACHINA}}(\Gamma, \text{desc}, \text{dep})$

The KACHINA leakage function reveals the public state transcript generated by Γ during the projected transition. This projected transition takes the state of the contract as the party currently sees it, and first replays all currently unconfirmed transactions from the same party. Both the initial (latest confirmed) contract state, as well as the projected state, and a randomness stream are considered the transaction's context.

When receiving an input $(\omega = (\ell,U,T,\omega = (\sigma^o,\rho^o)),\psi,w)$*:*

let $(\sigma^\pi,\rho^\pi) \leftarrow (\sigma^o,\rho^o[\psi])$

for u **in** U **do**
  let $(\psi', w', (\mathcal{T}_\sigma, z), \cdot, \cdot, D) \leftarrow T(u)$
  if $\mathcal{T}_\sigma(\sigma^\pi) = \bot$ **then**
    return $(\bot, \bot, \bot, \bot, \bot)$
  let $(\vec{\sigma}, \cdot, \vec{\rho}, \mathcal{T}, \cdot) \leftarrow \text{run-}\Gamma(\sigma^\pi, \rho^\pi, w', z, \psi' \in \mathcal{H})$
  let $\sigma^\pi \leftarrow \text{last}(\vec{\sigma}); \rho^\pi \leftarrow \text{last}(\vec{\rho})$
  let $X \leftarrow X \parallel (u, \mathcal{T}, z, D)$
let η be a randomness stream.
let $z \leftarrow (\sigma^o, \rho^o[\psi], \sigma^\pi, \rho^\pi, \eta)$
let $(\vec{\sigma}, \mathcal{T}_\sigma, \vec{\rho}, \mathcal{T}_\rho, \cdot) \leftarrow \text{run-}\Gamma(\sigma^\pi, \rho^\pi, w, z, \top)$
if $\text{last}(\vec{\sigma}) = \bot \vee \text{last}(\vec{\rho}) = \bot$ **then**
  return $(\bot, \bot, \bot, \bot, \bot)$
else
  let $D \leftarrow \text{dep}(X, \mathcal{T}_\rho, z)$
  return $(\text{desc}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z), \mathcal{T}_\sigma, D, (\mathcal{T}_\sigma, z))$

6.3.5 The Core Kachina Protocol

The construction of the core protocol itself is now fairly straightforward. We use non-interactive zero-knowledge to prove statements about transition functions interacting with an oracle. When creating a transaction, users prove that the generated transcript is consistent with the transition function and initial input. Instead of evaluating transactions, users apply the public (and, if available, private) state transcripts associated with them.

Formally, the relation $\mathcal{R}$ of the NIZK used is defined as follows, for any given transition function Γ: $((\mathcal{T}_\sigma, \cdot), (w, \mathcal{T}_\rho)) \in \mathcal{R}$ if and only if, where $\mathcal{O}_\sigma \leftarrow \mathcal{O}(\mathcal{T}_\sigma)$, and $\mathcal{O}_\rho \leftarrow \mathcal{O}(\mathcal{T}_\rho)$, $\text{last}(\Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)) \neq \bot$, and after it is run, $\text{consumed}(\mathcal{O}_\sigma) \wedge \text{consumed}(\mathcal{O}_\rho)$ holds. This is efficiently provable provided that $\mathcal{T}_\sigma$, w, and $\mathcal{T}_\rho$ are short, and Γ itself is efficiently expressible in the underlying zero-knowledge system.

Protocol Kachina

The Kachina protocol realises the ideal smart contract functionality when parameterised by a transition function Γ, a leakage descriptor desc, and a dependency function dep, such that the corresponding (Δ, Λ) pair is in $\mathbb{C}_{\text{Kachina}}$. It operates in the

$(\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}, \mathcal{G}_{\text{SimpleLedger}})$-hybrid model, where $\mathcal{R}$ is defined below.

$((\mathcal{T}_\sigma, \cdot), (w, \mathcal{T}_\rho)) \in \mathcal{R}$ if and only if, where $\mathcal{O}_\sigma \leftarrow \mathcal{O}(\mathcal{T}_\sigma)$ and $\mathcal{O}_\rho \leftarrow \mathcal{O}(\mathcal{T}_\rho)$, $\text{last}(\Gamma_{\mathcal{O}_\sigma, \mathcal{O}_\rho}(w)) \neq \bot$ and after it is run, $\text{consumed}(\mathcal{O}_\sigma) \wedge \text{consumed}(\mathcal{O}_\rho)$ holds.

State variables and initialisation values:

Variable	Description
$T \coloneqq \varnothing$	Mapping from transactions to their private state transcripts and contexts.
$Y \coloneqq \varnothing$	Mapping from transactions to their outputs.
$U \coloneqq \varepsilon$	Sequence of unconfirmed transactions.

When receiving a message (POST-QUERY, w) *from a party* ψ*:*

let $\Sigma \leftarrow \text{updateState}(\psi)$
let $(\sigma^o, \rho^o) \leftarrow \text{execState}(\Sigma)$
let $\sigma^\pi \leftarrow \sigma^o; \rho^\pi \leftarrow \rho^o; X \leftarrow \varepsilon$
for $u = (\mathcal{T}_\sigma, D, \cdot)$ **in** U **do**
  let $(\mathcal{T}_\rho, z) \leftarrow T(u)$
  let $\sigma^\pi \leftarrow \mathcal{T}_\sigma(\sigma^\pi); \rho^\pi \leftarrow \mathcal{T}_\rho(\rho^\pi, z)$
  let $X \leftarrow X \parallel (u, \text{split}(\mathcal{T}_\rho, \text{COMMIT}), z, D)$
let η be a randomness stream.
let $z \leftarrow (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta)$
let $(\vec{\sigma}, \mathcal{T}_\sigma, \vec{\rho}, \mathcal{T}_\rho, \vec{y}) \leftarrow \text{run-}\Gamma(\sigma^\pi, \rho^\pi, w, z)$
if $\text{last}(\vec{\sigma}) = \bot \vee \text{last}(\vec{\rho}) = \bot$ **then**
  return REJECTED
let $D \leftarrow \text{dep}(X, \mathcal{T}_\rho, z)$
send $(\text{LEAK}, \text{desc}(|\Sigma|, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z))$ **to** ψ and
  receive the reply b
if b **then**
  send $(\text{PROVE}, (\mathcal{T}_\sigma, D), (w, \mathcal{T}_\rho))$ **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and
    receive the reply π
  let $\text{tx} \leftarrow (\mathcal{T}_\sigma, D, \pi)$
  let $T(\text{tx}) \leftarrow (\mathcal{T}_\rho, z); Y(\text{tx}) \leftarrow \vec{y}; U \leftarrow U \parallel \text{tx}$
  send $(\text{SUBMIT}, \text{tx})$ **to** $\mathcal{G}_{\text{SimpleLedger}}$
  return $(\text{POSTED}, \text{tx})$
else
  return REJECTED

When receiving a message (CHECK-QUERY, tx) *from a party ψ:*

let $\Sigma \leftarrow$ updateState(ψ)
if tx $\in \Sigma$ **then return** execResult(prefix(Σ, tx))
else return NOT-FOUND

Helper procedures:

procedure updateState(ψ)
  send READ **to** $\mathcal{G}_{\text{SimpleLedger}}$ and **receive the reply** Σ
  let $C \leftarrow$ execConfirmed(Σ)
  let $U' \leftarrow U$
  repeat
    let $U \leftarrow U'$
    for tx $= (\cdot, D, \cdot)$ **in** U **do**
      if $D \notin (C \cup U) \vee (D \cap C) \not\subseteq \Sigma$ **then**
        let $U' \leftarrow U' \setminus \{\text{tx}\}$
  until $U = U'$
  return Σ
procedure execState(Σ)
  let $(\sigma, \cdot, \cdot) \leftarrow$ exec(Σ) **in return** σ
procedure execResult(Σ)
  let $(\cdot, y, \cdot) \leftarrow$ exec(Σ) **in return** y
procedure execConfirmed(Σ)
  let $(\cdot, \cdot, C) \leftarrow$ exec(Σ) **in return** C
procedure exec(Σ)
  let $\sigma \leftarrow \varnothing; \rho \leftarrow \varnothing; y \leftarrow \bot; C \leftarrow \varnothing$
  for tx $= (\mathcal{T}_\sigma, D, \pi)$ **in** dedup(Σ) **do**
    if tx $\in C$ **then continue**
    let $y \leftarrow \bot$
    send $(\text{VERIFY}, (\mathcal{T}_\sigma, D), \pi)$ **to** $\mathcal{F}^{\mathcal{R}}_{\text{NIZK}}$ and
      receive the reply b
    if $\neg b$ **then continue**
    if $D \setminus C \neq \varnothing \vee D \not\subseteq \Sigma$ **then continue**
    let $C \leftarrow C \cup \{\text{tx}\}$
    if $T(\text{tx}) \neq \bot$ **then**
      let parts $\leftarrow$ zip(split($\mathcal{T}_\sigma$, COMMIT), split(T(tx), COMMIT), Y(tx))
      for $(\mathcal{T}'_\sigma, \mathcal{T}_\rho, y')$ **in** parts **do**

```
            if T'_σ(σ) = ⊥ then
                let C ← C \ {tx}
                break
            let σ ← T'_σ(σ); ρ ← T_ρ(ρ); y ← y'
        else
            for T'_σ in split(T_σ, COMMIT) do
                if T_σ(σ) = ⊥ then
                    let C ← C \ {tx}
                    break
                let σ ← T'_σ(σ)
    return ((σ, ρ), y, C)
```

6.4 Security Analysis

The security of KACHINA is given through a standard UC security statement:

Theorem 6.1. *For any contract* $(\Delta, \Lambda) \in \mathbb{C}_{\text{KACHINA}}$*,* KACHINA *UC-emulates* $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$*, in the* $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$*-hybrid world, in the presence of* $\mathcal{G}_{\text{SimpleLedger}}$*.*

This is proven through a detailed case-analysis of any action an environment, in conjunction with the dummy adversary, may take. We define an invariant I between the real and ideal executions in the UC security statement, roughly encoding that "the real and ideal states are equivalent". This ranges from simple equivalences, such as them having the same ledger states, or the same NIZK proofs considered valid, to complex invariants, such as all unconfirmed honest transactions satisfying the sub-invariant J of Definition 6.11. This invariant is used to argue that the environment, in combination with a dummy adversary, cannot distinguishing between the real and ideal worlds. Specifically, for any action the environment takes, I is preserved, and from I holding, we can conclude that the information revealed to it, or the dummy adversary, is insufficient to distinguish the two worlds.

The simulator for KACHINA is quite straightforward; it simply creates simulated NIZK proofs for all honest transactions and forces the adversary to reveal witnesses to the simulated NIZK functionality in time for these to be input to the ideal smart contract. Fundamentally, the security proof relies on state tran-

scripts being interchangeable with full state oracles in the same setting, and this setting being enforced by both the protocol and functionality.

While a lot of factors must be formally considered, this is derived from receiving NIZK proofs as part of valid transactions, which prove precisely that if the preconditions for the transaction are met, then the update performed on the public state is the same. The private state is a little more tricky, but is guaranteed by the dependency invariant J holding for honest parties. This lets us similarly argue that the private state transcript will have the same effect as the ideal-world execution.

Proof. If an environment can distinguish between the ideal and real executions in presence of our simulator (see Subsection 6.4.1), then there must exist some polynomial sequence of interactions permitting it to distinguish with a non-negligible advantage. Broadly, each of the environment's actions falls into one of three categories: a) Honestly interacting with the protocol. b) Honestly interacting with the ledger. c) Commanding the adversary to perform some action in the real world. We will consider the responses the environment makes to queries given to the dummy adversary separately, in each case at the point where the query is made.

We will consider in parallel two random variables of the state of the ideal world execution and that of the real world execution at any time. We leave out of our analysis the "stack" of partial executions (as described in Subsection 2.3.4), except to show that the flow of each party – that is, when it is waiting for which query to be answered – is the same in both worlds. In particular, the state of the ideal world has the following functionalities' states as a part of it: 1. the state of the simulator, $\mathcal{S}$, 2. the state of the smart contract functionality, $\mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}$, and finally 3. the state of the ledger, $\mathcal{G}_{\mathsf{L}}^{i}$. In the real world, for each $\psi \in \mathcal{H}$, ψ's protocol state, which we refer to as ϕ_ψ, is part of the state, along with the (shared) NIZK hybrid functionality $\mathcal{F}_{\mathrm{NIZK}}^{\mathcal{R},r}$ and the real-world ledger, $\mathcal{G}_{\mathsf{L}}^{r}$. For convenience, we will often talk about these states as concrete variables and not random variables.

We will prove inductively that any action the environment takes will do two things: First, it will preserve an invariant I, which holds after the state of both worlds at any point during the two experiments. Second, if the invariant holds, the environment gains at most negligible advantage in distinguishing from its next action. To begin, we will specify the simulator, the invariant I, followed by

a few lemmas helpful in the proof. Finally, we will perform the induction itself.

6.4.1 The Simulator

The simulator for KACHINA has fairly little work to do. Firstly, it creates simulated transactions by creating a simulated NIZK proof and attaching it to the leakage x. Secondly, when presented with an unknown transaction and asked for the corresponding input, it attempts to extract the input from the simulated zero-knowledge functionality.

Simulator $\mathcal{S}_{\text{KACHINA}}$

The simulator $\mathcal{S}_{\text{KACHINA}}$ has two main points of interaction in the ideal world: First, it gets notified of the leakage of honest submissions, in the form of the new public state σ', and decides their format on the ledger. Second, it gets queried when an adversarial transaction is seen on the ledger, and must assign meaning to them. Furthermore, it simulates the non-global functionality $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$, which the adversary may interact with.

State variables and initialisation values:

Variable	Description
$\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$	Simulation of $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$

When receiving a message $(\text{TRANSACTION}, \mathcal{T}_\sigma, D)$ *from* $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$*:*

query $\mathcal{A}$ **with** $(\text{PROVE}, (\mathcal{T}_\sigma, D))$ and
  receive the reply π,
  satisfying $\pi \neq \bot \wedge (\cdot, \pi) \notin \mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.\Pi \wedge (x, \pi) \notin \mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.\overline{\Pi}$, else
  sampling from $\{0,1\}^\kappa$, **on behalf of** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$
let $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.\Pi \leftarrow \mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.\Pi \cup \{((\mathcal{T}_\sigma, D), \pi)\}$
return $((\mathcal{T}_\sigma, D, \pi), \varnothing)$

When receiving a message $(\text{INPUT}, (\mathcal{T}_\sigma, D, \pi))$ *from* $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$*:*

simulate sending $(\text{VERIFY}, (\mathcal{T}_\sigma, D), \pi)$ **to** $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ and **receive the reply** b
if $b \wedge \exists w, \mathcal{T}_\rho \colon \mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.W((\mathcal{T}_\sigma, D), \pi) = (w, \mathcal{T}_\rho)$ **then**
  return $(\mathcal{A}, w, (\mathcal{T}_\sigma, \mathcal{O}(\mathcal{T}_\rho)), \varnothing, D)$
else
  return NONE

Forward all queries to $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$ *to the simulated instance. Forward all queries to global functionalities directly.*

6.4.2 The Invariant I

Definition 6.14. The invariant I is the conjunction of all of the constraints below, over the state variables of a UC experiment on a pair of matching real and ideal worlds:

(1) The ledgers are indistinguishable:

$$\mathcal{G}^i_{\mathsf{L}}.\Sigma = \mathcal{G}^r_{\mathsf{L}}.\Sigma \wedge \forall \psi \in \mathcal{H}\colon \mathcal{G}^i_{\mathsf{L}}.M(\psi) = \mathcal{G}^r_{\mathsf{L}}.M(\psi)$$

(2) The simulated and real NIZKs consider the same statement/proof pairs valid and invalid:

$$\mathcal{S}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\Pi = \mathcal{F}^{\mathcal{R},r}_{\mathsf{NIZK}}.\Pi \wedge \mathcal{S}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.\overline{\Pi} = \mathcal{F}^{\mathcal{R},r}_{\mathsf{NIZK}}.\overline{\Pi}$$

(3) Real world witnesses have a corresponding ideal world witness:

$$\begin{aligned}
&\forall \mathcal{T}_\sigma, D, \pi\colon \exists \mathcal{T}_\rho, w\colon \mathcal{F}^{\mathcal{R},r}_{\mathsf{NIZK}}.W(((\mathcal{T}_\sigma, D), \pi)) = (\mathcal{T}_\rho, w) \implies \\
&\quad \mathcal{S}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W(((\mathcal{T}_\sigma, D), \pi)) = (\mathcal{T}_\rho, w) \vee \\
&\quad [\exists \psi \in \mathcal{H}, z = (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta)\colon \\
&\qquad \phi_\psi.T((\mathcal{T}_\sigma, D, \pi)) = (\mathcal{T}_\rho, z) \wedge \\
&\qquad \mathcal{F}^{\Delta,\Lambda}_{\mathsf{SC}}.T((\mathcal{T}_\sigma, D, \pi)) = (\psi, w, (\mathcal{T}_\sigma, z), \varnothing, D) \wedge \\
&\qquad \mathsf{run}\text{-}\Gamma(\sigma^\pi, \rho^\pi, w, z, \top) = (\cdot, \mathcal{T}_\sigma, \cdot, \mathcal{T}_\rho, \phi_\psi.Y((\mathcal{T}_\sigma, D, \pi)))]
\end{aligned}$$

(4) Recorded transactions are proven, and only adversarial witnesses are known by the simulator:

$$\begin{aligned}
&\forall \mathcal{T}_\sigma, D, \pi, \psi\colon \mathcal{F}^{\Delta,\Lambda}_{\mathsf{SC}}.T((\mathcal{T}_\sigma, D, \pi)) = (\psi, \ldots) \implies \\
&\quad ((\mathcal{T}_\sigma, D), \pi) \in \mathcal{F}^{\mathcal{R},r}_{\mathsf{NIZK}}.\Pi \wedge \\
&\quad (\psi \notin \mathcal{H} \iff ((\mathcal{T}_\sigma, D), \pi) \in \mathcal{S}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}.W)
\end{aligned}$$

(5) Honest parties record transactions correctly:

$$\begin{aligned}
&\forall \psi \in \mathcal{H}, \mathsf{tx}\colon \\
&\quad (\mathsf{tx} \in \phi_\psi.T \iff \mathsf{tx} \in \phi_\psi.Y \iff \mathcal{F}^{\Delta,\Lambda}_{\mathsf{SC}}.T(\mathsf{tx}) = (\psi, \ldots)) \wedge \\
&\quad (\mathsf{tx} \in \phi_\psi.U \implies \mathsf{tx} \in \phi_\psi.T)
\end{aligned}$$

(6) All recorded transactions respect dependencies and transcripts:

$$\begin{aligned}
&\forall \mathsf{tx} \in \mathcal{F}^{\Delta,\Lambda}_{\mathsf{SC}}.T\colon \\
&\quad \mathcal{F}^{\Delta,\Lambda}_{\mathsf{SC}}.T(\mathsf{tx}) = \textsc{none} \vee \\
&\quad (\exists \mathcal{T}, D, \pi\colon \mathsf{tx} = (\mathcal{T}, D, \pi) \wedge \\
&\qquad \mathcal{F}^{\Delta,\Lambda}_{\mathsf{SC}}.T(\mathsf{tx}) = (\cdot, \cdot, (\mathcal{T}, \cdot), \varnothing, D))
\end{aligned}$$

(7) Recorded as rejected transactions are disproven:

$$\forall \mathcal{T}_\sigma, D, \pi\colon \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T((\mathcal{T}_\sigma, D, \pi)) = \textsc{none} \implies ((\mathcal{T}_\sigma, D), \pi) \in \mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$$

(8) The dependency invariant J holds for all honest unconfirmed transactions: $\forall \psi \in \mathcal{H}$, let Σ be the longest prefix of $\mathcal{G}_{\text{L}}^{r}.M(\psi)$ such that $\Sigma \cap \phi_\psi.U = \varnothing$; define $X(u = (\cdot, D, \cdot)) \coloneqq \text{let } (\mathcal{T}, z) = \phi_\psi.T(u) \text{ in } (u, \text{split}(\mathcal{T}, \textsc{commit}), z, D)$ and $((\cdot, \rho), \cdot, C) \coloneqq \phi_\psi.\text{exec}(\Sigma)$. Then $J(\text{map}(X, \phi_\psi.U), \rho) \wedge \text{sat}(\text{map}(X, \phi_\psi.U), \phi_\psi.U) \wedge \forall (\cdot, D, \cdot) \in \phi_\psi.U\colon D \setminus C \setminus \phi_\psi.U = \varnothing$ holds.

(9) Transactions owned by an honest party and not in their view of the ledger, are considered unconfirmed, or can never be accepted: Let Σ be the longest prefix of $\mathcal{G}_{\text{L}}^{r}.M(\psi)$ such that $\forall \text{tx} \in \Sigma\colon \text{tx} \in \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$.

$$\forall \psi \in \mathcal{H}, \text{tx} \notin \Sigma\colon \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) = (\psi, \ldots) \implies$$
$$\text{tx} \in \phi_\psi.U \vee (\nexists \Sigma' \succ \mathcal{G}_{\text{L}}^{r}.M(\psi)\colon \text{tx} \in \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.\text{execConfirmed}(\Sigma' \parallel \text{tx})$$

(10) All results and state updates are consistent with the input and transcripts:

function transcriptConsistent$(\sigma_0, \rho_0, w, z, \mathcal{T}_\sigma, \mathcal{T}_\rho, Y)$
  let $(\vec{\sigma}, \cdot, \vec{\rho}, \vec{y}) \leftarrow \text{run-}\Gamma(\sigma_0, \rho_0, w, z, \top)$
  let $\vec{\mathcal{T}}_\sigma \leftarrow \text{split}(\mathcal{T}_\sigma, \textsc{commit})$
  let $\vec{\mathcal{T}}_\rho \leftarrow \text{split}(\mathcal{T}_\rho, \textsc{commit})$
  let $\sigma \leftarrow \sigma_0; \rho \leftarrow \rho_0$
  let parts $\leftarrow \text{zip}(\vec{\sigma}, \vec{\rho}, \vec{\mathcal{T}}_\sigma, \vec{\mathcal{T}}_\rho, \vec{y}, Y)$
  for $(\sigma', \rho', \mathcal{T}'_\sigma, \mathcal{T}'_\rho, y_1, y_2) \leftarrow$ parts **do**
    let $\sigma \leftarrow \mathcal{T}_\sigma(\sigma); \rho \leftarrow \mathcal{T}_\rho(\rho)$
    if $\sigma = \bot$ **then break**
    else if $\sigma \neq \sigma' \vee \rho \neq \rho' \vee y_1 \neq y_2$ **then return** $\bot$
  return $\top$

$$\forall \psi \in \mathcal{H}, \mathcal{T}_\sigma, \text{tx} = (\mathcal{T}_\sigma, \cdot, \cdot), w, z\colon \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) = (\psi, w, z, \varnothing, \cdot) \implies$$
$$[\exists \mathcal{T}_\rho\colon \phi_\psi.T(\text{tx}) = (\mathcal{T}_\rho, z) \wedge \phi_\psi.Y(\text{tx}) = \Gamma_{\mathcal{O}(\mathcal{T}_\sigma), \mathcal{O}(\mathcal{T}_\rho)}(w) \wedge$$
$$[\forall \sigma, \rho\colon \text{transcriptConsistent}(\sigma, \rho, w, z, \mathcal{T}_\sigma, \mathcal{T}_\rho, \phi_\psi.Y(\text{tx}))]]$$

(11) Execution results should be equivalent for prefixes *and* extensions of the ledger state containing no new adversarial transactions:

$$\forall \Sigma, \psi \in \mathcal{H}\colon ((\Sigma \prec \mathcal{G}_{\text{L}}^{r}.\Sigma \vee \mathcal{G}_{\text{L}}^{r}.\Sigma \prec \Sigma) \wedge \forall \text{tx} \in \Sigma\colon$$
$$\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) \neq \bot) \implies$$

$$
\begin{aligned}
&\text{let } ((\sigma^i, \rho^i), y^i, C^i) \leftarrow \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.\mathrm{exec}(\Sigma);\\
&\quad ((\sigma^r, \rho^r), y^r, C^r) \leftarrow \phi_\psi.\mathrm{exec}(\Sigma);\\
&\text{in } \sigma^i = \rho^i \wedge \rho^i[\psi] = \rho^r \wedge C^i = C^r \wedge\\
&\quad \text{if } \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T(\Sigma[-1]) = (\psi, \ldots) \text{ then } y^r = y^i \text{ else } y^r = \bot
\end{aligned}
$$

(12) Recorded transactions which are canonically preceeded by a (yet) unrecorded transaction, are honest and considered unconfirmed by their owner:

$$
\begin{aligned}
&\forall \mathsf{tx} \in (\mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T \cap \mathcal{G}_{\mathrm{L}}^{r}.\Sigma), \mathsf{tx}' \in (\mathcal{G}_{\mathrm{L}}^{r}.\Sigma \setminus \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T), \psi \in \mathcal{H}:\\
&\quad \mathrm{idx}(\mathcal{G}_{\mathrm{L}}^{r}.\Sigma, \mathsf{tx}') < \mathrm{idx}(\mathcal{G}_{\mathrm{L}}^{r}.\Sigma, \mathsf{tx}) \wedge \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T(\mathsf{tx}) = (\psi, \ldots) \implies\\
&\qquad \mathsf{tx} \in \phi_\psi.U \vee (\nexists \Sigma' \succ \mathcal{G}_{\mathrm{L}}^{r}.\Sigma\colon \mathsf{tx} \in \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.\mathrm{execConfirmed}(\Sigma' \parallel \mathsf{tx}))
\end{aligned}
$$

(13) The ledger is ahead of any party's ledger:

$$\forall \psi \in \mathcal{H}\colon \mathcal{G}_{\mathrm{L}}^{r}.M(\psi) \prec \mathcal{G}_{\mathrm{L}}^{r}.\Sigma$$

(14) The same transactions are unconfirmed in both worlds:

$$\forall \psi \in \mathcal{H}\colon \phi_\psi.U = \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.U_\psi$$

(15) NIZK proofs have witnesses:

$$
\begin{aligned}
&\forall x, \pi\colon (x, \pi) \in \mathcal{F}_{\mathrm{NIZK}}^{\mathcal{R},r}.\Pi \iff\\
&\quad \exists w\colon \mathcal{F}_{\mathrm{NIZK}}^{\mathcal{R},r}.W((x, \pi)) = w \wedge\\
&\quad \mathcal{S}.\mathcal{F}_{\mathrm{NIZK}}^{\mathcal{R}}.W((x, \pi)) \in \{w, \bot\} \wedge\\
&\quad (x, w) \in \mathcal{R}
\end{aligned}
$$

(16) Recorded transactions are either on the ledger, considered unconfirmed by an honest party, or can never be satisfied:

$$
\begin{aligned}
&\forall \mathsf{tx} \in \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T\colon \mathsf{tx} \in \mathcal{G}_{\mathrm{L}}.\Sigma \vee\\
&\quad (\exists \psi \in \mathcal{H}\colon \mathsf{tx} \in \phi_\psi.U \wedge \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T(\mathsf{tx}) = (\psi, \ldots)) \vee\\
&\quad (\nexists \Sigma' \succ \mathcal{G}_{\mathrm{L}}.\Sigma\colon \mathsf{tx} \in \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.\mathrm{execConfirmed}(\Sigma' \parallel \mathsf{tx}))
\end{aligned}
$$

Often many of these parts of the invariant are trivially preserved due to the state variables constrained in them being left unchanged. Such trivial cases will be omitted in our analysis.

6.4.3 Supporting Lemmas

Both $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$ and KACHINA have exec functions, which executes an entire ledger state given to it. Lemma 6.2 is a generalisation of invariant (11), and simply states that this execution will preserve the invariant and return the same values in the real and ideal world.

Lemma 6.2. *For any $\psi \in \mathcal{H}$, Σ where $\Sigma \prec \mathcal{G}_{\text{L}}^{r}.\Sigma \vee \mathcal{G}_{\text{L}}^{r}.\Sigma \prec \Sigma$, and after sending the message $\left(\text{EXTEND}, \Sigma \setminus \mathcal{G}_{\text{L}}^{r}.\Sigma\right)$ to $\mathcal{G}_{\text{L}}$, $((\sigma^i, \rho^i), y^i, C^i)$ is the result of running* $\text{exec}(\Sigma)$ *in $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$ and $((\sigma^r, \rho^r), y^r, C^r)$ is the result of running* $\text{exec}(\Sigma)$ *in ϕ_ψ, these interactions preserve I and the returned values are equivalent: $\sigma^i = \sigma^r \wedge \rho^i[\psi] = \rho^r$. If the last transaction* tx *in Σ is owned by ψ (i.e., $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) = (\psi, \ldots)$), then $y^i = y^r$, otherwise $y^r = \bot$.*

Proof. First, we consider the EXTEND call. This will only extend if Σ is longer than $\mathcal{G}_{\text{L}}.\Sigma$ – otherwise it extends with ε, which is a no-op. This call preserves I, as demonstrated in Subsection 6.4.4.

We prove the lemma by induction over Σ. In the base case, $\Sigma = \varepsilon$. The invariant is trivially satisfied and the returned values are equivalent (when $\varnothing$ is interpreted as public/private state pairs). In the induction step, we proceed by case analysis for the new transaction $\text{tx} = (\mathcal{T}, D, \pi)$:

Case 1. The $\text{tx} \in \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$ and all processed transactions so far have also been recorded (are in $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$). If so, then by (11), the return values are equivalent. Furthermore, this iteration does not change the state in the ideal world. By (4) and (7), we also know that the transaction is either in $\mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\Pi$, or $\mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$. As a result, no state changes will be made in the real-world execution either, trivially preserving I.

Case 2. $\text{tx} \notin \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$, but $((\mathcal{T}, D), \pi) \in \mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$. In this case, the real world will skip this transaction and set y to $\bot$. In the ideal world, the simulator will ensure that $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx})$ is set to NONE and equally this transaction is skipped, with y set to $\bot$. This affects and preserves the following invariants:

(3) As by (15), tx has no witness.

(4) As $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) = \text{NONE}$, not satisfying the precondition.

(5) As tx was not in $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$ in the induction hypothesis and is not associated with an honest party.

(6) By $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx})$ being NONE, satisfying the postcondition.

(7) Due to $((\mathcal{T}, D), \pi) \in \mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$.

(9) As $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) = \text{NONE}$, not satisfying the precondition.

(10) As tx was not in $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$ in the induction hypothesis, it cannot be in any $\phi_\psi.Y$, by (5).

(11) By the output equivalence part of the induction step holding.

(12) By tx being previously unrecorded, further restricting the quantification domain and $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) = \text{NONE}$, not satisfying the precondition.

(16) By the newly recorded transaction being in the ledger state, as this has been extended if necessary.

Case 3. $\text{tx} \notin \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$, but $((\mathcal{T}, D), \pi) \in \mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\Pi$. In this case, by (15) a witness must be recorded and by (3) this witness must be accessible to the simulator. As a result, the simulator will ensure that $T(\text{tx})$ is set to $(\mathcal{A}, w, (\mathcal{T}_\sigma, \mathcal{O}(\mathcal{T}_\rho)), \varnothing, D)$. As this is an adversarial transactions, the ρ-value of the adversary is not constrained and neither is the output y-value. As a result, to show the execution equivalence holds, it suffices to show that both worlds will have the same σ-value after this new transaction. In the real world, the commit-separated form of $\mathcal{T}_\sigma$ is applied to σ in parts, with the last non-$\perp$ state being adopted. In the ideal world, the $\mathcal{T}_\sigma$ is recomputed and the parts compared with those passed as inputs. The confirmation depth is derived from how many parts match before the computed and input transcripts diverge, or the result is $\perp$. The ideal world runs $\text{run-}\Gamma(\sigma, \mathcal{T}'_\sigma, w, \mathcal{O}(\mathcal{T}_\rho), \perp)$. Since $((\mathcal{T}_\sigma, D), (\mathcal{T}_\rho, w)) \in \mathcal{R}$ (by (15)), we know that the public state oracle in $\text{run-}\Gamma$ can be replaced with $\mathcal{O}(\mathcal{T}_\sigma)$, up to the confirmation depth, after which the executions may diverge. As a result, the σ returned in the ideal world – $\vec{\sigma}$ indexed at the confirmation depth – matches that returned in the real world. As $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$ is set, the following parts of the invariant are affected and preserved:

(3) By the left hand side of the disjunction already being satisfied.

(4) By the transaction being recorded in the NIZK, and the simulator knowing its witness.

(5) As tx was not in $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$ in the induction hypothesis and is not associated with an honest party.

(6) By the newly recorded transaction satisfying the postcondition.

(7) By the newly recorded transaction not being recorded as rejected.

(9) By the newly recorded transaction not being honestly owned, not satisfying the precondition.

(10) As tx was not in $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$ in the induction hypothesis, it cannot be in any $\phi_\psi.Y$, by (5).

(11) By the output equivalence part of the induction step holding.

(12) By tx being previously unrecorded, further restricting the quantification domain and $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T(\text{tx}) = (\mathcal{A}, \ldots)$, not satisfying the precondition.

(16) By the newly recorded transaction being in the ledger state, as this has been extended if necessary.

Case 4. The transaction has not been previously seen – that is, $\text{tx} \notin \mathcal{F}_{\text{SC}}^{\Delta,\Lambda}.T$ and $((\mathcal{T}, D), \pi) \notin \mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\Pi \cup \mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$. In this case, both the real and ideal worlds will attempt the same NIZK verification (simulated in the ideal world). By (2), they will both query the adversary for a NIZK witness in the same way, handing off execution. By the induction hypothesis, I holds as the point of execution transfer and, as the query made is the same in both worlds, the environment gains no means to distinguish.

As NIZK verification is the first thing done in both worlds and NIZK verification is agnostic as to which party is verifying, this is equivalent to the environment *first* manually verifying the same statement/proof pair. As will be shown in Subsection 6.4.4, this preserves the invariant and returns the same result in both worlds. Therefore, Case 4 is equivalent to either Case 2 (if the NIZK verification failed), or Case 3 (if the NIZK verification succeeded), as if the NIZK verification were done externally beforehand, the statement/proof pair *must* be either in $\mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\Pi$, or in $\mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$.

Case 5. $\mathsf{tx} \in \mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T$, however (11) cannot be applied, as other transactions have since been added. By (12), we know that tx belongs to an honest party ψ' and that $\mathsf{tx} \in \phi_{\psi'}.U$. We will use (8) to argue that, where $(\mathcal{T}_\rho, z) = \phi_{\psi'}.T(\mathsf{tx})$, either $\mathcal{T}_\rho(\rho^i[\psi'], z) \neq \bot$, or the transaction is skipped in both worlds.

First, we consider the possibility that $\mathsf{tx} \notin \phi_{\psi'}.U$. By (9) we know that tx cannot ever be confirmed by a suffix of the ledger state referred to in the invariant. As this is a prefix of $\mathcal{G}_{\mathsf{L}}^r.\Sigma$, such that it contains no unrecorded transactions, the current induction is necessarily a suffix of it. As a result, we know that the ideal world execution will fail. As transactions are rejected in both worlds under the same conditions – due to dependencies not being satisfied – we can conclude that these transactions are also skipped in the real world, preserving I as no state variables are changed and satisfying all conditions by the induction hypothesis. We will now focus on the case that $\mathsf{tx} \in \phi_{\psi'}.U$

Next, we determine that, given $\mathsf{tx} \in \phi_{\psi'}.U$, the longest prefix Σ^* referred to in (8) is a prefix of the ledger state Σ we are currently performing induction over. We know it to be a prefix of $\mathcal{G}_{\mathsf{L}}.M(\psi')$, such that this prefix contains none of the transactions in $\phi_{\psi'}.U$. As $\mathsf{tx} \in \phi_{\psi'}.U$ and is either a prefix or extension of $\mathcal{G}_{\mathsf{L}}^r.\Sigma$, of which $\mathcal{G}_{\mathsf{L}}^r.M(\psi')$ is itself a prefix by (13), we can conclude that $\Sigma^* \prec \Sigma$.

To apply (8), we are only concerned with the party's private state ρ, we can observe all transactions in $\Sigma \setminus \Sigma^*$ are either not owned by ψ', will not be accepted in any context, or are in $\phi_{\psi'}.U$. We can ignore the first possibility, as the real world execution of them will not affect ρ, regardless. The second can also be ignored, as these will be skipped by the ideal world execution and, by induction hypothesis, by the real world execution as well. Next, we consider which of the transaction in $\Sigma \setminus \Sigma^*$ owned by ψ' have been successfully processed. exec provides replay protection, ensuring that each unconfirmed transaction has been processed at most once. By induction hypothesis, the sequence A of such transactions that have results associated for this party is the same in both worlds. As exec will not set the state to $\bot$, we know that there exists a confirmation depth vector $\vec{c}$, such that $\mathcal{T}^*_{ES(\mathsf{map}(\phi_{\psi'}.T, A), \vec{c})}(\rho^*[\psi']) \neq \bot$ is the result of applying these transactions in the real world. Here, ρ^* is taken to be the private state corresponding to the prefix Σ^* in the ideal world.

Now that tx is processed, we know by (8) that its dependencies are either in confirmed, and in order, in Σ^*, or in $\phi_{\psi'}.U$. In either case, tx is skipped in both worlds if it is a replayed transaction.

As $\mathsf{tx} \in \phi_{\psi'}.U$ and is not a replay, $B = A \parallel \mathsf{tx}$ is a permutation of a subset of $\phi_{\psi'}.U$. As a result, by (8), we know that $\mathcal{T}^*_{ES(\mathsf{map}(\phi_{\psi'}.T,B),\vec{c} \parallel \cdot)}(\rho^*[\psi']) \neq \bot$. As we have previously established this holds for A, by definition of $\mathcal{T}^*$, this implies that applying $\mathcal{T}_\rho$ to $\rho[\psi']^i$ to any confirmation depth is non-$\bot$, where $\rho[\psi']^i$ is the same as the ideal world private state for the induction hypothesis, by repeated application of (10). Likewise, (10) allows us to conclude that σ^i will also be non-$\bot$, as the update applied to it will be equivalent to applying $\mathcal{T}_\sigma$ to the same confirmation depth, which by definition of confirmation depth is not $\bot$.

If the transaction is skipped in both worlds, the induction hypothesis still applies. Otherwise, up to the confirmation depth, applying both public and private state transcript parts is non-$\bot$. As previously noted in Subsection 6.3.1, this is equivalent to partial oracle executions to this confirmation depth and therefore the ideal and real world states match. Likewise, (10) applies (as by (5), $\phi_{\psi'}.Y(\mathsf{tx})$ is set) and we know the ideal-world result $y^i = \phi_{\psi'}.Y(\mathsf{tx})[c]$, where c is the confirmation depth.

If $\psi = \psi'$, by (5), $\phi_\psi.T$ is defined and, as a result, the same update is carried out to ρ in the real world, as to $\rho^i[\psi]$ in the ideal world. Furthermore, it will return the same result y^r as the ideal world, as $\mathsf{proj}_c(\phi_\psi.Y(\mathsf{tx})) = y^i$. If $\psi \neq \psi'$, the ideal world update does not affect $\rho^i[\psi]$ and the correctness of the returned private state is guaranteed by the induction hypothesis. $y^r = \bot$ is returned, which satisfies the requirements. Finally, in both cases, if the confirmation depth is maximal, the transaction is added to C, ensuring the returned C is the same in both worlds. Neither world makes any state updates, trivially preserving I. □

Lemma 6.3. *If I holds, then for all $\psi \in \mathcal{H}$, running* updateState(ψ) *in $\mathcal{F}^{\Delta,\Lambda}_{\mathrm{SC}}$ and running* updateState *in ϕ_ψ preserves I.*

Proof. To begin with, both worlds retrieve the same value Σ / Σ_ψ from $\mathcal{G}_{\mathsf{L}}$, due to (1). As seen in Subsection 6.4.4, this preserves I. Next, by Lemma 6.2, both worlds receive the same value C and the execConfirmed call preserves I. The worlds now iterate over $\phi_\psi.U$ and $\mathcal{F}^{\Delta,\Lambda}_{\mathrm{SC}}.U_\psi$, respectively, which by (14) are equal in value. The operations performed are almost identical, with the exception of the real world deconstructing $u = (\cdot, D, \cdot)$ for each $u \in U$, while the ideal world extracts $(\ldots, D) = \mathcal{F}^{\Delta,\Lambda}_{\mathrm{SC}}.T(u)$ instead. By (6), if $u \in \mathcal{F}^{\Delta,\Lambda}_{\mathrm{SC}}.T$, the two are equivalent and, by (5), as $u \in \phi_\psi.U$, it is also in both $\phi_\psi.T$ and $\mathcal{F}^{\Delta,\Lambda}_{\mathrm{SC}}.T$. We conclude that both worlds perform the same operations. Updated is only $\phi_\psi.U$ and $\mathcal{F}^{\Delta,\Lambda}_{\mathrm{SC}}.U_\psi$, respectively. The

following parts of the invariant are affected and preserved:

(5) By reducing the scope of $\phi_\psi.U$.

(8) This consists of three sub-parts: The satisfaction of J, that of sat and that $D \backslash C \backslash U = \varnothing$. The first is trivial: J makes a statement about all permutations of subsets. A smaller initial set merely reduces the scope of the quantifiers. The second holds due to updateState ensuring that if a transaction is removed from U, any transactions that depend on it are also removed, with the remaining transactions being in the same order as before. As a result, a previously satisfied transaction is either removed itself, or still satisfied, as it does not depend on any removed transactions and dependencies still in U being in the same order as before. Finally, $D \backslash C \backslash U = \varnothing$ is also preserved due to the recursive removal. Specifically, if $D \notin (C \cup U)$ the corresponding transaction is removed. As a result, only transactions satisfying this condition will remain.

(9) As the removed transactions either fail confirmation directly (it depends on a transaction rejected in Σ_ψ, or a different transaction order than got enforced), or depends on a transaction which fails. In either case, any state Σ', of which Σ_ψ is a prefix, cannot accept these for the same reasons.

(12) As in (9).

(14) By equal update. □

6.4.4 Proof of Theorem 6.1

We proceed with the main inductive proof of Theorem 6.1. We consider the base case of the system initialisations in the real and ideal worlds. The induction hypothesis is that after $k < 2^\kappa$ interactions with any environment, the state of both worlds satisfy the invariant I and the environment has not gained a non-negligible advantage in distinguishing. We will assume, without loss of generality, the adversary being a dummy adversary. We provide a concrete list of actions the environment may take before taking the induction step. We note that as at any point the environment cannot distinguish, we can assume that it takes the same action in both worlds without loss of generality.

Base Case.

Proof. Most base cases hold either due to equal initialisation of variables constrained to be equal, or due to initialisation leaving forall quantifiers to quantify over the empty set. The former is the case for: (1), (2), (5), and (14). The latter is the case for: (3), (4), (6), (7), (9), (10), (12), and (16). The remaining hold for the following reasons:

(8) At initialisation, the only prefix of $\mathcal{G}_{\mathsf{L}}^{r}.M(\psi)$ is ε. $\phi_{\psi}.\text{execState}(\varepsilon) = (\varnothing, \varnothing)$. The base case therefore holds iff $J(\varnothing, \varnothing)$ holds. This in turn holds iff $\mathcal{T}_{\varepsilon}^{*}(\varnothing) \neq \bot$, or $\varnothing \neq \bot$.

(11) At initialisation, the only ledger state Σ which satisfies the condition that $\forall \mathsf{tx} \in \Sigma: \mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}.T(\mathsf{tx}) \neq \bot$ is ε. For this, as both worlds are initialised to equivalent contract states, the outputs of exec will be equal.

(13) By the reflexivity of $\prec$. □

Induction Step.

Proof. We observe that the environment is capable of the following queries:

- $\forall \psi \in \mathcal{H}, w$:[4] Sending $(\textsc{post-query}, w)$ to $\mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}$ or Kachina.
- $\forall \psi \in \mathcal{H}, \mathsf{tx}$: Sending $(\textsc{check-query}, \mathsf{tx})$ to $\mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}$ or Kachina.
- $\forall \psi \in \mathcal{P}, \mathsf{tx}$: Sending $(\textsc{submit}, \mathsf{tx})$ to $\mathcal{G}_{\mathsf{L}}$.
- $\forall \psi \in \mathcal{P}$: Sending read to $\mathcal{G}_{\mathsf{L}}$.
- $\forall \Sigma'$: Sending $(\textsc{extend}, \Sigma')$ to $\mathcal{G}_{\mathsf{L}}$.
- $\forall \psi, \Sigma'$: Sending $(\textsc{advance}, \psi, \Sigma')$ to $\mathcal{G}_{\mathsf{L}}$.
- $\forall \psi \in \mathcal{P} \setminus \mathcal{H}$: Sending $(\textsc{prove}, x, w)$ to $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$.
- $\forall \psi \in \mathcal{P}$:[5] Sending $(\textsc{verify}, x, \pi)$ to $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}$.

We will prove that I is preserved across any of these queries and that they reveal the same information in both worlds.

[4] We omit without loss of generality the environment's ability to make honest queries with corrupted parties. The environment may simulate running the honest protocol to replicate these.

[5] Technically, as in prove, the environment can only instruct corrupted parties to verify. As verification for honest parties preserves the invariant as well, and is a useful lemma, we prove the more general statement.

Case (POST-QUERY, w)**.** We proceed by sub-case analysis. We identify the following cases: 1. The transaction is rejected by the contract. 2. The transaction is rejected by the user. 3. The transaction is posted. In all cases, updateState is first run. By Lemma 6.3, this preserves the invariant and also ensures that the returned value $\Sigma_\psi = \mathcal{G}_\mathsf{L}.M(\psi)$ is the value returned in both worlds (by (1)). In the ideal world, Λ is called. The real world largely emulates the same, computing most of the same values identically. Of note are the values σ^o and $\rho^o/\rho^o[\psi]$, which are computed in both worlds using execState(Σ_ψ). By Lemma 6.2, this preserves the invariant and returns the same values.

The only place where the two worlds diverge in their computation is in handling the unconfirmed transactions – the ideal world executes run-Γ and updates σ^π, ρ^π, and X according to the confirmation depth, while the real world partially applies $\mathcal{T}_\sigma$ and $\mathcal{T}_\rho$ to the confirmation depth. Before we go into the main three cases, we will argue that, if the transaction is *not* rejected by the contract, then these two approaches will yield the same result and that they will reject equally.

To begin with, in the ideal world the confirmation depth is derived from the number of transcript parts matching between the newly generated and input transcripts. As a transcript application is non-$\bot$ if and only if it can be generated in the same way in the current state, this ensures that the confirmation depth matches in the two worlds.

Furthermore, we observe that in the real world, the final value of ρ^π *cannot* be $\bot$ – to begin with, updateState guarantees that $\Sigma_\psi \cap \phi_\psi.U = \varnothing$. This in turn, along with (8) ensures that $J(X)$ holds, as well as that $\mathrm{sat}^*(X, U)$ holds. It follows $\mathcal{T}^*(\rho^o) \neq \bot$, where $\mathcal{T}^*$ performs the same repeated applications of $\mathcal{T}_\rho(\rho, z)$ as the loop in the main protocol, using the same values. Furthermore, by (3) and (4), we can conclude that the transcripts $\mathcal{T}_\rho$ and contexts z are the same in both worlds. By (10) , we can conclude that the final ρ^π values are also the same in both worlds. Furthermore, as $\mathcal{T}_\rho$ and z are equal in both worlds and, by (6) both D and the sequence X are also equal in real and ideal worlds. Subsequently, σ, ρ, and D are computed equivalently in both worlds.

We now consider the main case analysis: If the contract rejects the transaction in the ideal world, the returned description is $\bot$. This happens if and only if $\mathrm{last}(\vec{\sigma}) = \bot \vee \mathrm{last}(\vec{\rho}) = \bot$, the same condition as the real world protocol has for rejecting the transaction before querying the user. If the transaction is rejected, no variables are modified, preserving I, and the same value is returned in both

worlds, giving the environment no means to distinguish.

If the contract does not automatically reject the query, the leakage descriptor is computed equally in both worlds and sent to the party to acknowledge. The party has the opportunity to accept the described leakage, or cancel the transaction. At the point of handing over execution to the environment, no state has been modified, trivially preserving I and, as the same leakage descriptor is given, it has no means to distinguishing.

In the case of the environment subsequently cancelling the transaction, both worlds immediately return with REJECTED, again trivially preserving I and giving no means to distinguish.

Finally, if the environment accepts the leakage, both worlds obtain the transaction identifier tx: The simulator ensures that the real-world adversary is queried for the same NIZK proof as it is in the real-world and that the transaction format matches that of real-world transactions. At the time of the proof query, no state has been modified, trivially preserving I. As the same statement is queried for, the environment gains no information to distinguish.

Subsequently, both worlds record the transaction's information (in $\mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}.T$ and $\phi_\psi.T$) and note it as unconfirmed (in $\mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}.U$ and $\phi_\psi.U$). In the real world, the result is further recorded in $\phi_\psi.Y$. The following parts of I are affected and preserved (including the PROVE query):

(2) By $((\mathcal{T}_\sigma, D), \pi)$ being added to both worlds' Π equally.

(3) As $\phi_\psi.T$, $\mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}.T$, and $\phi_\psi.Y$ are appropriately set to satisfy the RHS of the disjunction.

(4) As for the newly added transaction, $\psi \in \mathcal{H}$ and $((\mathcal{T}_\sigma, D), \pi) \notin \mathcal{S}.\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R}}.W$ (by the uniqueness of statement/proof pairs).

(5) As the newly added transaction is added to all of $\phi_\psi.T$, $\phi_\psi.U$, and $\mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}.T$, where it is associated with ψ.

(6) As the newly added transaction does consist of transcript, dependencies and proof, and the former two are recorded in $\mathcal{F}_{\mathsf{SC}}^{\Delta,\Lambda}.T$ correctly and $\mathcal{S}$ returns $\varnothing$ for a.

(7) As the newly recorded transaction is not recorded as NONE.

(8) By J being preserved when appending a new transaction, J holds after the induction step (as ρ remains unaffected). sat holds by induction hypothesis and as $D \sqsubseteq U \setminus \{\mathsf{tx}\}$. For the new transaction, $D \setminus C \setminus \phi_\psi.U = \varnothing$ as $D \subseteq \phi_\psi.U$; for previously transactions this still holds, as $\phi_\psi.U$ is expanded.

(9) As the newly added transaction is also unconfirmed by the owning party.

(10) By $\mathcal{T}_\sigma$, $\mathcal{T}_\rho$, and $\vec{y}$ having been extracted from run-Γ, operating in the context of w, z, and some σ, ρ, with these values being recorded in the corresponding state variables (except σ and ρ). As the transcripts are transcripts of oracle evaluations against w and $\vec{y}$ is the result of Γ operating with these oracles, executing $\Gamma_{\mathcal{O}(\mathcal{T}_\sigma),\mathcal{O}(\mathcal{T}_\rho)}(w)$ has the same effect. Furthermore, as the transcripts accurately reproduce the state change in the original state context, by definition of transcript execution, if the sequence of transcripts up to the confirmation depth can be applied to be non-$\bot$, they are indistinguishable from making the original queries to the state oracle. Combined with the sequence of queries made depending only on w and the state oracle itself, we can conclude that the transcript applications are the same as executing against the state oracles up to the confirmation depth, regardless of which initial state the transcript could be successfully applied to.

(11) As a new transaction has been recorded, we must now additionally consider transaction sequences Σ which contain this new transaction at some point. We cannot directly use Lemma 6.2, however we can make use of its induction: If we can show that any Σ *ending* with the new transaction tx satisfies the execution equivalences, then induction from Lemma 6.2 can apply on that as a base case (in particular, the precondition for Case 5 applies for all subsequent transactions). The execution equivalence holds for this new base case, as we know that this new transaction is both honest and considered unconfirmed for this party. Therefore, the argument for Case 5 holds for tx itself as well. As the execution equivalence defined in Lemma 6.2 is the same as that of (11), this part of the invariant is preserved.

(12) By the newly added transaction being unconfirmed, it satisfies any quantification where tx is set to it. By now being recorded, the range of quantifications for tx′ is restricted, relaxing the condition.

(14) By equal update.

(16) By the newly recorded transaction being considered unconfirmed by an honest party.

Finally, both worlds submit to the ledger the same transaction tx, which is simply sent to the adversary. At this point I holds as argued above and, as the same transaction is sent, the environment cannot distinguish. Finally, (POSTED, tx) is returned, giving the environment no information to distinguish for the same reasons.

Case (CHECK-QUERY, tx)**.** After running updateState, Lemma 6.3 preserves the invariant, but also ensures that $\Sigma_\psi = \mathcal{G}_\mathsf{L}.M(\psi)$, where Σ_ψ is the value returned in both worlds (by (1)).

We consider three cases: 1. $\mathsf{tx} \notin \Sigma_\psi$, 2. $\mathcal{F}_{\mathrm{SC}}^{\Delta,\Lambda}.T(\mathsf{tx}) = (\psi, \ldots)$, and 3. otherwise. In Case 1, both worlds return NOT-FOUND without updating any state, not allowing the environment to distinguish and preserving I. In Case 2, both worlds run execResult(prefix(Σ_ψ, tx)), preserving I according to Lemma 6.2, and returning the same value in both worlds, giving the environment no information to distinguish. Finally, in Case 3, only the real world runs execResult, while the ideal world returns $\bot$. As previously in updateState the sub-function execConfirmed was run, we know that all NIZK-verifications performed in this exec call have previously been made – as a result the call modifies no state and preserves I. Furthermore, by Lemma 6.2, it returns $\bot$, as in the ideal world, giving the environment no information to distinguish.

Case (SUBMIT, tx)**.** In both worlds tx is handed to the adversary and no other action is taken. As the same information is relayed, the environment cannot distinguish and, as no state is changed, I is preserved.

Case READ**.** By (1), both worlds will return the same result; therefore the environment cannot distinguish. As no state is changed, I is preserved.

Case (EXTEND, Σ')**.** As nothing is returned, the environment gains no information allowing it to distinguish. By (1), the updates done are the same in both worlds. The parts of the invariant affected and preserved are the following:

(1) By equal update.

(11) Extending $\mathcal{G}_{\mathsf{L}}.\Sigma$ further constrains the possible Σ values quantified over.

(12) Without loss of generality, we can assume single-transaction appends to $\mathcal{G}^r_{\mathsf{L}}.\Sigma$. If a new unrecorded transaction is added, it (at first) does not precede any transactions, leaving the quantification unchanged and relaxing the non-existence quantifier. If a recorded honest transaction is added, then by (9) and (13), this transaction satisfies the conditions.

(13) By the append-only nature of EXTEND.

(16) By relaxing the constraint.

Case $(\textsc{advance}, \psi, \Sigma')$**.** As nothing is returned, the environment gains no information allowing it to distinguish. By (1), the updates done are the same in both worlds. The parts of the invariant affected and preserved are the following:

(1) By equal update.

(8) Without loss of generality, we can assume single-transaction advances. If $\mathcal{G}_{\mathsf{L}}.M(\psi) \cap \phi_\psi.U \neq \varnothing$, or the newly added transaction $\mathsf{tx} \in \phi_\psi.U$, this is preserved as the longest prefix remains equal. Otherwise, Σ in the induction step is that of the induction hypothesis, with one transaction $\mathsf{tx} \notin \phi_\psi.U$ appended. If tx is not owned by ψ, by (5), $\phi_\psi.T(\mathsf{tx}) = \bot$ and therefore $\mathsf{execState}(\Sigma \parallel \mathsf{tx})$ returns the same ρ as $\mathsf{execState}(\Sigma)$, preserving the invariant. If tx is owned by ψ, by (9), this transaction will be rejected, likewise returning the same ρ. Furthermore, as $D \setminus C \setminus U$ is already $\varnothing$ for all dependency lists D and extending Σ can only lead to C growing, this condition remains satisfied.

(9) By further restricting all-quantification and non-existance quantification.

(13) By condition that $\mathcal{G}^r_{\mathsf{L}}.M(\psi) \prec \mathcal{G}^r_{\mathsf{L}}.\Sigma$.

Case $(\textsc{prove}, x, w)$**.** In the ideal world, this query is handled by the simulated functionality $\mathcal{S}.\mathcal{F}^{\mathcal{R}}_{\mathsf{NIZK}}$. If $(x, w) \notin \mathcal{R}$ the call returns immediately with $\bot$ in both worlds and no variables are modified, giving the environment no information to distinguish and preserving I. Otherwise, the adversary is immediately queried with $(\textsc{prove}, x)$ in both worlds. Again, at this point no variables have been modified, preserving I, and the information handed to the adversary is the same in

both worlds, giving the environment no information to distinguish. The adversary will eventually respond with a proof π, which is verified against constraints in both worlds and randomly sampled if it does not meet them. By (2), the constraints are identical in both worlds. Finally Π and W are set and π returned in both worlds, giving no distinguishing information to the environment. The following parts of the invariant are affected and preserved:

(2) By equal update.

(3) By equal insertion into $\mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.W$ and $\mathcal{S}.\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}.W$.

(4) By relaxing the constraint.

(9) As the possible results of executing transactions consisting of unrecorded statement/proof pairs is constrained – the environment can no longer decide if they should be processed or not.

(11) As in (9).

(12) As in (9).

(15) As only members of $\mathcal{R}$ are recorded.

(16) As in (9).

Case (VERIFY, x, π)**.** The flow for verification is only slightly more complex than that for proving. At a high level, the adversary may be given a chance to produce a last-moment witness for the statement being verified. If it refuses to do so, the proof is recorded as definitively invalid. We consider three sub-cases: 1. The statement/proof pair is recorded as either valid or invalid. 2. The adversary returns a valid witness. 3. The adversary does not return a valid witness.

In Case 1, VERIFY returns the same value in both worlds by (2), giving the environment no means to distinguish. Case 2 is equivalent to the adversary first sending a PROVE query for the given statement, supplying it with the corresponding proof, and then running the VERIFY query. We therefore refer to Case 1 and the case of PROVE. In Cases 1 and 2, no state is changed, preserving I. Finally, for Case 3, $\mathcal{F}_{\text{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$ is updated equally in both worlds and $\perp$ is returned in both worlds, giving the environment no information to distinguish. In this case, the following parts of the invariant are affected and preserved:

(2) By equal update.

(7) By relaxing the condition on $\mathcal{F}_{\mathsf{NIZK}}^{\mathcal{R},r}.\overline{\Pi}$.

(9) As the possible results of executing transactions consisting of unrecorded statement/proof pairs is constrained – the environment can no longer decide if they should be processed or not.

(11) As in (9).

(12) As in (9).

(16) As in (9). □

As the environment cannot distinguish with non-negligible probability between the real and ideal world in any single action if I is preserved and, as I is preserved with overwhelming probability across each action by the environment and holds at protocol initialisation, we conclude that the environment cannot win the UC game. □

6.5 A Case Study: Private Payments

To demonstrate the versatility of Kachina, we take a closer look at the (private) token contract, which is prone to the scalability issues Kachina addresses. Public token contracts are well understood and standardised [VB15], with the typical implementation being to maintain a mapping of "addresses" (hashes of public keys) to balances in the contract's public state. We write the first *provably private* token contract to demonstrate the expressive power of Kachina.

A private token contract also implies that currency is not a primitive – it can be built as a contract, a key factor in simplifying our model, as it does not need to encode currency as a special case. It provides an asset to build contracts around in the first place, as well as a means of denial-of-service mitigation, through transaction fees. Bad fee models have resulted in devastating DoS attacks [Wil16], highlighting the necessity of well-chosen transaction fees.

We detail how to construct a fee model in Subsection 6.7.5. The fundamental idea of this construction is to embed the transition function Γ in a wrapper which performs the following steps:

1. In the private state oracle, estimate the cost of transaction fees.
2. Given an input gas price and this estimate, pay these fees using a designated currency contract.
3. Commit this as a partial execution success.
4. Execute Γ with a modified $\mathcal{O}_\sigma$, which deducts from available gas for each operation and aborts if this runs out.
5. Transfer any remaining gas back to the transaction author.

6.5.1 Indirect Construction

Following the design of Zerocash [BCG+14], we write a contract that maintains the necessary Zerocash secrets: coin randomnesses, commitment openings, and secret keys. The private state oracle computes the off-chain information required to make a Zerocash transaction: Merkle-paths to your own commitments, the selection of randomness for new coins, and the encryption of the secret information of these coins. This information is handed to the central, provable core of the contract, which computes a coin's serial number, verifies the Merkle-path, and verifies the integrity of the transaction. Finally, the serial number and new commitment are sent to the public state oracle, which ensures the former is new and adds the latter to the current tree.

This design is not self-evidently correct and is not the objective itself. Specifying what goal it achieves, in terms of an ideal leakage and transition function, allows us to build a clean ideal world, with a clear private token contract. This ideal world is constructed in two steps: First showing that the Zerocash contract UC-emulates it, and second showing that the Zerocash contract is in turn UC-emulated by KACHINA.

6.5.2 Ideal Private Payments

To simplify the external interface, we only use single denomination coins. The same approach can be applied to the full Zerocash protocol, with some caveats on coin selection and leakage.

We formally specify the private token contract through its transition and leakage functions, Δ_{pp} and Λ_{pp}. The contract supports the following inputs:

- INIT, giving a party a unique public key

- (SEND, pk), sending a coin to the public key pk
- MINT, creating a new coin for the calling party
- BALANCE, returning the current balance

Transition Function Δ_{pp}

The state transition function for a private payments system. Parties have associated public keys and balances. The payments system allows for parties without a public key to generate one, and for parties to transfer and mint single-denomination coins, as well as query their own balance.

State variables and initialisation values:

Variable	Description
$K \coloneqq \varnothing$	Mapping of parties to public keys
$B \coloneqq \lambda\psi.\, 0$	Mapping of parties to their spendable coins

When receiving an input $(\omega, \psi, \text{INIT}, \cdot, \text{pk})$*:*

if $\omega.K(\psi) = \bot$ **then**
  while $\exists \psi' : \text{pk} = \omega.K(\psi') \vee \text{pk} \in \{\varnothing, \bot\}$ **do**
    let $\text{pk} \xleftarrow{*} \{0,1\}^\kappa$
  let $\omega.K(\psi) \leftarrow \text{pk}$
  return $(\omega, \top, \text{pk})$
else
  return $(\bot, \bot, \bot)$

When receiving an input $(\omega, \psi, (\text{SEND}, \text{pk}), \cdot, a)$*:*

if $\psi \notin \mathcal{H} \wedge a \neq \varnothing$ **then**
  let $\text{pk}' \leftarrow a$
  assert $\nexists \psi' \in \mathcal{H} : \text{pk}' = \omega.K(\psi')$
else if $\omega.K(\psi) \neq \bot$ **then let** $\text{pk}' \leftarrow \omega.K(\psi)$
else return $(\bot, \bot, \bot)$
if $\omega.B(\text{pk}') > 0$ **then**
  let $\omega.B(\text{pk}') \leftarrow \omega.B(\text{pk}') - 1$
  let $\omega.B(\text{pk}) \leftarrow \omega.B(\text{pk}) + 1$
  return $(\omega, \top, \top)$
else return $(\bot, \bot, \bot)$

When receiving an input $(\omega, \psi, \text{MINT}, \text{pk}, \cdot)$*:*

let $\omega.B(\text{pk}) \leftarrow \omega.B(\text{pk}) + 1$
return $(\omega, \top, \top)$

When receiving an input $(\omega, \psi, \textsc{balance}, B, \cdot)$*:*

return $(\omega, \top, B)$

Leakage Function Λ_{pp}

Each operation on Δ_{pp} has minimal leakage, revealing only which operation was performed, and, in the case of a transfer, the time and the recipient – if and only if the recipient is corrupted.

When receiving an input $(\omega = (\ell, U, T, \omega), \psi, w)$*:*

let $\omega^\pi \leftarrow \omega$
let $B^- \leftarrow 0$
for u **in** U **do**
  let $(\cdot, w', z, a, \cdot, \cdot) \leftarrow T(u)$
  if $w' = (\textsc{send}, \cdot)$ **then let** $B^- \leftarrow B^- + 1$
  let $(\omega^\pi, \cdot) \leftarrow \Delta_{\text{pp}}(\omega^\pi, \psi, w', z, a)$
if $w = \textsc{init}$ **then**
  if $\omega.K(\psi) = \omega^\pi.K(\psi) = \bot$ **then**
    return $(\textsc{init}, \textsc{init}, \varepsilon, \varnothing)$
  else return $(\bot, \bot, \bot, \bot)$
else if $\exists \text{pk}\colon w = (\textsc{send}, \text{pk})$ **then**
  let $c \xleftarrow{*} \{0,1\}^\kappa$
  if $\omega.B(\omega.K(\psi)) - B^- > 0 \wedge \omega.K(\psi) = \omega^\pi(\psi) \neq \bot$ **then**
    let $\text{lkg} \leftarrow t$
    if $\nexists \psi' \in \mathcal{H}\colon \text{pk} = \omega.K(\psi')$ **then let** $\text{lkg} \leftarrow (\ell, \text{pk})$
    return $((\textsc{send}, \ell, \text{pk}), \text{lkg}, \varepsilon, \varnothing)$
  else return $(\bot, \bot, \bot, \bot)$
else if $w = \textsc{mint} \wedge \omega.K(\psi) \neq \bot$ **then**
  return $(\textsc{mint}, \textsc{mint}, \varepsilon, \omega.K(\psi))$
else if $w = \textsc{balance} \wedge \omega.K(\psi) \neq \bot$ **then**
  return $(\textsc{balance}, \textsc{balance}, \varepsilon, \omega.B(\omega.K(\psi)) - B^-)$
else
  return $(\bot, \bot, \bot, \bot)$

6.5.3 The Zerocash KACHINA Contract

The contract implementing Zerocash, which we will use to realise the private token contract, follows its source protocol closely, albeit with single denomination coins.

Transition Function Γ_{zc}

The state transition function for a Zerocash-based token contract. In blue are parts run in the public state oracle, in red are parts run in the private state oracle.

Public state variables and initialisation values:

Variable	Description
$\text{cms} \coloneqq \varnothing$	Public coin commitment set
$\text{sns} \coloneqq \varnothing$	Public serial number set
$\vec{R} \coloneqq \varepsilon$	Vector of commitment Merkle tree roots
$\vec{M} \coloneqq \varepsilon$	Vector of encrypted messages

Private state variables and initialisation values:

Variable	Description
$i \coloneqq 0$	Index of $\vec{M}$ processed.
$\vec{C} \coloneqq \varepsilon$	Vector of coins available.
$K_e \coloneqq \bot$	Encryption secret key.
$K_z \coloneqq \bot$	Zero-knowledge secret key.

When receiving an input INIT*:*

send INIT **to** $\mathcal{O}_\rho$ and **receive the reply** pk

return pk

When receiving an input $(\text{SEND}, (\text{pk}_z, \text{pk}_e))$*:*

send $(\text{SEND}, \text{pk}_e)$ **to** $\mathcal{O}_\rho$ and

receive the reply $(p, r, K_z, p', r', \text{rt}, \text{path}, M)$

assert path is a valid Merkle tree path with root rt, to the element

$\text{comm}_r((\text{prf}^{\text{pk}}_{K_z}(1), p))$

let $\text{sn} \leftarrow \text{prf}^{\text{sn}}_{K_z}((p, r))$

let $\text{cm} \leftarrow \text{comm}_{r'}(pk_z, p')$

send $(\text{SPEND}, \text{sn}, \text{rt})$ **to** $\mathcal{O}_\sigma$

send (MSG, M) **to** $\mathcal{O}_\sigma$

send (MINT, cm) **to** $\mathcal{O}_\sigma$

return $\top$

When receiving an input MINT*:*

send MINT **to** $\mathcal{O}_\rho$ and **receive the reply** cm

send (MINT, cm) **to** $\mathcal{O}_\sigma$

return $\top$

When receiving an input BALANCE*:*

send BALANCE **to** $\mathcal{O}_\rho$ and **receive the reply** B

return B

When receiving a private oracle query INIT*:*

assert $\rho^\pi.K_e = \bot \wedge \rho^\pi.K_z = \bot$

let $\rho.K_z \xleftarrow{*} \{0,1\}^\kappa$

let $(\rho.K_e, \mathsf{pk}_e) \leftarrow \mathsf{keyGen}(1^\kappa)$

return $(\mathsf{prf}^{\mathsf{pk}}_{\rho.K_z}(1), \mathsf{pk}_e)$

When receiving a private oracle query (SEND, pk_e)*:*

let $\rho^o \leftarrow \mathsf{update}(\rho^o, \sigma^o)$

let $\rho^\pi \leftarrow \mathsf{update}(\rho^\pi, \sigma^\pi)$

let $\rho \leftarrow \mathsf{update}(\rho, \sigma^\pi)$

assert $(\rho^o.\vec{C} \cap \rho^\pi.\vec{C}) \neq \varepsilon$

let $(p, r) \leftarrow (\rho^o.\vec{C} \cap \rho^\pi.\vec{C})[0]$

let $\rho.\vec{C} \leftarrow \rho.\vec{C} \setminus \{(p, r)\}$

let rt $\leftarrow \mathsf{merkleroot}(\sigma^o.\mathsf{cms})$

let path $\leftarrow \mathsf{merklepath}(\mathsf{comm}_r((\mathsf{prf}^{\mathsf{pk}}_{\rho^o.K_z}(1), p)), \mathsf{rt})$

let $(p', r') \xleftarrow{*} \{0,1\}^\kappa \times \{0,1\}^\kappa$

let $M \leftarrow \mathsf{enc}((r', p'), \mathsf{pk}_e)$

let $K_z \leftarrow \rho^o.K_z$

return $(p, r, \rho^o.K_z, p', r', \mathsf{rt}, \mathsf{path}, M)$

When receiving a private oracle query MINT*:*

assert $\rho^o.K_e \neq \bot \wedge \rho^o.K_z \neq \bot$

let $(p, r) \xleftarrow{*} \{0,1\}^\kappa \times \{0,1\}^\kappa$

let cm $\leftarrow \mathsf{comm}_r(\mathsf{prf}^{\mathsf{pk}}_{\rho^o.K_z}(1), p)$

let $\rho.\vec{C} \leftarrow \rho.\vec{C} \parallel (p, r)$

return cm

When receiving a private oracle query BALANCE*:*

let $\rho^o \leftarrow \mathsf{update}(\rho^o, \sigma^o)$

let $\rho^\pi \leftarrow \mathsf{update}(\rho^\pi, \sigma^\pi)$

return $|\rho^o.\vec{C} \cap \rho^\pi.\vec{C}|$

When receiving a public oracle query (SPEND, sn, rt)*:*

assert sn $\notin \sigma$.sns

assert rt $\in \sigma.\vec{R}$

let σ.sns $\leftarrow \sigma$.sns $\cup$ {sn}

When receiving a public oracle query (MSG, M)*:*

let $\sigma.\vec{M} \leftarrow \sigma.\vec{M} \parallel M$

When receiving a public oracle query (MINT, cm)*:*

let σ.cms $\leftarrow \sigma$.cms $\cup$ {cm}

let $\sigma.\vec{R} \leftarrow \sigma.\vec{R} \parallel$ merkleroot(σ.cms)

Helper procedures:

function update(ρ, σ)

let $\vec{N} \leftarrow \sigma.\vec{M}[\rho.i{:}]; \rho.i \leftarrow \max(\rho.i, |\sigma.\vec{M}|)$

for $M \in \vec{N}$ **do**

if $\exists r, p\colon (r, p) = \mathrm{dec}(M, \rho.K_e)$ **then**

if $\mathrm{comm}_r((\mathrm{prf}^{\mathsf{pk}}_{\rho.K_z}(1), p) \notin \sigma$.cms **then continue**

if $\mathrm{prf}^{\mathsf{sn}}_{\rho.K_z}(p) \in \sigma$.sns **then continue**

let $\rho.\vec{C} \leftarrow \rho.\vec{C} \parallel (r, p)$

return ρ

function $\mathrm{dep}_{\mathrm{zc}}(X, \mathcal{T}, z)$

return ε

function $\mathrm{desc}_{\mathrm{zc}}(t, \cdot, \cdot, \cdot, w, \cdot)$

if w = INIT **then return** INIT

else if $\exists$pk: w = (SEND, pk) **then return** (SEND, t, pk)

else if w = MINT **then return** MINT

else if w = BALANCE **then return** BALANCE

else return $\perp$

Lemma 6.4. Γ_{zc} *and* $\mathrm{dep}_{\mathrm{zc}}$ *satisfy Definition 6.12, and therefore the pair* $(\Delta_{\mathrm{zc}}, \Lambda_{\mathrm{zc}}) \coloneqq (\Delta_{\textsc{Kachina}}(\Gamma_{\mathrm{zc}}), \Lambda_{\textsc{Kachina}}(\Gamma_{\mathrm{zc}}, \mathrm{desc}_{\mathrm{zc}}, \mathrm{dep}_{\mathrm{zc}}))$ *is in the set* $\mathbb{C}_{\textsc{Kachina}}$.

Proof (sketch). Transcripts generated by run-Γ fall into three categories: They set a private key (initialisation), they insert a coin (minting), or they remove a coin and insert some number of coins (sending).

Consider first a new initialisation transaction. It does not affect the behaviour of unconfirmed minting and sending transactions, as these do not use the current private state's secret key. Furthermore, it cannot co-exist with another unconfirmed initialisation transaction, as this would initialise the private keys, ensuring an abort, which violates the preconditions of dependencies.

If the new transaction is a minting or balance transaction, this functions independently of other transactions, not having any requirements on the current private state. Likewise for sending transactions, the state transcript itself only depends on $\rho^{\{o,\pi\}}$, not the dynamic ρ. The only thing varying is which coins get added and removed from the set of available coins, but this information is not directly used – its purpose is to reduce the necessary re-computation the next time around. □

We can observe that (with some help from the simulator) the ideal Zerocash contract, given by $(\Delta_{zc}, \Lambda_{zc}) = (\Delta_{\text{KACHINA}}(\Gamma_{zc}), \Lambda_{\text{KACHINA}}(\Gamma, \text{desc}_{zc}, \text{dep}_{zc}))$, is equivalent to the ideal private payments contract $(\Delta_{pp}, \Lambda_{pp})$. Formally, we instantiate two instances of $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$, as presented in Subsection 6.2.2 and show that any attack against $(\Delta_{zc}, \Lambda_{zc})$ can be simulated against $(\Delta_{pp}, \Lambda_{pp})$.

6.5.4 Security analysis

We can observe that (with some help from the simulator), the ideal Zerocash contract, given by $(\Delta_{zc}, \Lambda_{zc}) = (\Delta_{\text{KACHINA}}(\Gamma_{zc}), \Lambda_{\text{KACHINA}}(\Gamma, \text{desc}_{zc}, \text{dep}_{zc}))$, is equivalent to the ideal private payments contract $(\Delta_{pp}, \Lambda_{pp})$. Formally, we instantiate two instances of $\mathcal{F}_{\text{SC}}^{\Delta,\Lambda}$, as presented in Subsection 6.2.2, and show that any attack against $(\Delta_{zc}, \Lambda_{zc})$ can be simulated against $(\Delta_{pp}, \Lambda_{pp})$.

Theorem 6.2. *$\mathcal{F}_{\text{SC}}^{\Delta_{pp},\Lambda_{pp}}$ is UC-emulated by $\mathcal{F}_{\text{SC}}^{\Delta_{zc},\Lambda_{zc}}$ in presence of $\mathcal{G}_{\text{SimpleLedger}}$.*

Corollary 6.1. *$\mathcal{F}_{\text{SC}}^{\Delta_{pp},\Lambda_{pp}}$ is UC-emulated by* KACHINA*, parameterised by Γ_{zc}, dep_{zc}, and desc_{zc}, in the $\mathcal{F}_{\text{NIZK}}^{\mathcal{R}}$-hybrid world, in the presence of $\mathcal{G}_{\text{SimpleLedger}}$.*

This proof can also be carried out via invariants. Here the invariant tracking is simple: The real and ideal world have the same coins owned by the same users at any time. Our simulator, described below, has a lot of book-keeping to do, mostly to conjure up fake commitments and encryptions for the real-world adversary, and replicating them in the real world.

Simulator $\mathcal{S}_{\text{ZC}}$

The fully detailed Zerocash simulator.

State variables and initialisation values:

Variable	Description
$B \coloneqq \varnothing$	Unspent adversarial coins.
$K \coloneqq \varnothing$	Honest public/private key pairs.
$T \coloneqq \varnothing$	Mapping of transactions to created coin commitments.

When receiving a message $(\textsc{transaction}, x, D)$ *from* $\mathcal{F}_{\text{SC}}^{\Delta_{\text{pp}},\Lambda_{\text{pp}}}$*:*

if $x = \textsc{init}$ **then**
  let $\mathcal{T}_\sigma \leftarrow \varepsilon$
  query $\mathcal{A}$ **with** $(\textsc{transaction}, \mathcal{T}_\sigma, D)$ and
    receive the reply $(\mathsf{tx}, \cdot)$,
    satisfying $T(\mathsf{tx}) = \bot \wedge \mathsf{tx} \neq \bot$, else
    sampling from $(\{0,1\}^\kappa, \bot)$
  let $(K_e, \mathsf{pk}_e) \leftarrow \mathsf{keyGen}(1^\kappa)$
  let $K_z \xleftarrow{*} \{0,1\}^\kappa$
  let $\mathsf{pk}_z \leftarrow \mathsf{prf}^{\mathsf{pk}}_{K_z}(1)$
  let $K \leftarrow K \cup \{((K_z, K_e), (\mathsf{pk}_z, \mathsf{pk}_e))\}$
  let $T(\mathsf{tx}) \leftarrow \varnothing$
  return $(\mathsf{tx}, (\mathsf{pk}_z, \mathsf{pk}_e))$
else if $x = (\textsc{send}, t, (\mathsf{pk}_z, \mathsf{pk}_e))$ **then**
  let $p \xleftarrow{*} \{0,1\}^\kappa$; $r \xleftarrow{*} \{0,1\}^\kappa$
  let $\mathsf{sn} \xleftarrow{*} \mathsf{prf}^{\mathsf{sn}}_{\{0,1\}^\kappa}(\{0,1\}^\kappa \times \{0,1\}^\kappa)$
  let $\mathsf{rt} \leftarrow \mathsf{root}(t)$
  let $\mathsf{cm} \leftarrow \mathsf{comm}_r(\mathsf{pk}_z, p)$
  let $B \leftarrow B \cup \{(\mathsf{pk}_z, \mathsf{pk}_e, \mathsf{cm})\}$
  let $M \leftarrow \mathsf{enc}((r, p), \mathsf{pk}_e)$
  let $\mathcal{T}_\sigma \leftarrow ((\textsc{spend}, \mathsf{sn}, \mathsf{rt}), \varnothing) \parallel ((\textsc{msg}, M), \varnothing) \parallel$
    $((\textsc{mint}, \mathsf{cm}), \varnothing)$
  query $\mathcal{A}$ **with** $(\textsc{transaction}, \mathcal{T}_\sigma, D)$ and
    receive the reply $(\mathsf{tx}, \cdot)$,
    satisfying $T(\mathsf{tx}) = \bot \wedge \mathsf{tx} \neq \bot$, else
    sampling from $(\{0,1\}^\kappa, \bot)$
  let $T(\mathsf{tx}) \leftarrow \{\mathsf{cm}\}$
  return $(\mathsf{tx}, \varnothing)$

else if $x = (\textsc{send}, t)$ **then**
  let $(\cdot, \mathsf{pk}) \xleftarrow{*} \mathsf{keyGen}(1^\kappa)$
  let $\mathsf{rt} \leftarrow \mathsf{root}(t)$
  let $\mathsf{cm} \xleftarrow{*} \mathsf{comm}_{\{0,1\}^\kappa}(\mathsf{prf}^{\mathsf{pk}}_{\{0,1\}^\kappa}(1), \{0,1\}^\kappa)$
  let $\mathsf{sn} \xleftarrow{*} \mathsf{prf}^{\mathsf{sn}}_{\{0,1\}^\kappa}(\{0,1\}^\kappa \times \{0,1\}^\kappa)$
  let $M \xleftarrow{*} \mathsf{enc}((\{0,1\}^\kappa, \{0,1\}^\kappa), \mathsf{pk})$
  let $\mathcal{T}_\sigma \leftarrow ((\textsc{spend}, \mathsf{sn}, \mathsf{rt}), \varnothing) \parallel ((\textsc{msg}, M), \varnothing) \parallel$
    $((\textsc{mint}, \mathsf{cm}), \varnothing)$
  query $\mathcal{A}$ **with** $(\textsc{transaction}, \mathcal{T}_\sigma, D)$ and
    receive the reply $(\mathsf{tx}, \cdot)$,
    satisfying $T(\mathsf{tx}) = \bot \wedge \mathsf{tx} \neq \bot$, else
    sampling from $(\{0,1\}^\kappa, \bot)$
  let $T(\mathsf{tx}) \leftarrow \{\mathsf{cm}\}$
  return $(\mathsf{tx}, \varnothing)$
else if $x = \textsc{mint}$ **then**
  let $\mathsf{cm} \xleftarrow{*} \mathsf{comm}_{\{0,1\}^\kappa}(\mathsf{prf}^{\mathsf{pk}}_{\{0,1\}^\kappa}(1), \{0,1\}^\kappa)$
  let $\mathcal{T}_\sigma \leftarrow ((\textsc{mint}, \mathsf{cm}), \varnothing)$
  query $\mathcal{A}$ **with** $(\textsc{transaction}, \mathcal{T}_\sigma, D)$ and
    receive the reply $(\mathsf{tx}, \cdot)$,
    satisfying $T(\mathsf{tx}) = \bot \wedge \mathsf{tx} \neq \bot$, else
    sampling from $(\{0,1\}^\kappa, \bot)$
  let $T(\mathsf{tx}) \leftarrow \{\mathsf{cm}\}$
  return $(\mathsf{tx}, \varnothing)$
else if $x = \textsc{balance}$ **then**
  let $\mathcal{T}_\sigma \leftarrow \varepsilon$
  query $\mathcal{A}$ **with** $(\textsc{transaction}, \mathcal{T}_\sigma, D)$ and
    receive the reply $(\mathsf{tx}, \cdot)$,
    satisfying $T(\mathsf{tx}) = \bot \wedge \mathsf{tx} \neq \bot$, else
    sampling from $(\{0,1\}^\kappa, \bot)$
  return $(\mathsf{tx}, \varnothing)$
else abort
return (tx, a')

When receiving a message $(\textsc{input}, \mathsf{tx})$ *from* $\mathcal{F}_{\mathrm{SC}}^{\Delta_{\mathrm{pp}}, \Lambda_{\mathrm{pp}}}$*:*

send $(\textsc{input}, \mathsf{tx})$ **to** $\mathcal{A}$ and
  receive the reply $(\psi, w, (\mathcal{T}_\sigma, \mathcal{O}_\rho), \cdot, D)$
if $T(\mathsf{tx}) \neq \bot$ **then return** NONE

let $T(\mathsf{tx}) \leftarrow \varnothing$

if $w = \big(\textsc{send}, (\mathsf{pk}_z, \cdot)\big)$ **then**

if $\mathcal{T}_\sigma = ((\textsc{spend}, \mathsf{sn}, \mathsf{rt}), \varnothing) \parallel ((\textsc{msg}, M), \varnothing) \parallel ((\textsc{mint}, \mathsf{cm}'), \varnothing)$ **then**

send $(\textsc{send}, \mathsf{pk}_e)$ **to** $\mathcal{O}_\rho$ and

receive the reply $(p, r, K_z, p', r', \mathsf{rt}', \mathsf{path}, M')$

let $\mathsf{cm} \leftarrow \mathsf{comm}_r((\mathsf{prf}^{\mathsf{pk}}_{K_z}, p))$

let $b \leftarrow \top$

send READ **to** $\mathcal{G}_{\mathsf{Ledger}}$ and **receive the reply** Σ

if $\nexists t\colon 0 \le t \le |\Sigma| \wedge \mathsf{rt} = \mathsf{root}(t) \wedge \exists \mathsf{tx}\colon (T(\mathsf{tx}) = \mathsf{cm} \wedge \mathsf{tx} \in \Sigma[:t])$ **then**

let $b \leftarrow \bot$

if $\mathsf{sn} \neq \mathsf{prf}^{\mathsf{sn}}_{K_z}((p, r)) \vee \mathsf{rt} \neq \mathsf{rt}' \vee M \neq M'$ **then**

let $b \leftarrow \bot$

if $\mathsf{cm}' \neq \mathsf{comm}_{r'}(\mathsf{pk}_z, p')$ **then let** $b \leftarrow \bot$

if $\neg b$ **then return** NONE

// We now know the transaction is valid.

// We must determine if M can be

// honestly decrypted, and which

// adversarial coin is being spent.

if $\exists((\cdot, K_e), (\mathsf{pk}_z, \mathsf{pk}_e)) \in K$ **then**

let $d = \mathsf{dec}(M, K_e)$

if $d = (r', p')$ **then**

let $w \leftarrow \big(\textsc{send}, (\mathsf{pk}_z, \mathsf{pk}_e)\big)$

else

let $w \leftarrow (\textsc{send}, (\textsc{simkey}, \bot))$

else

let $B \leftarrow B \cup \{(\textsc{simkey}, \bot, \mathsf{cm}')\}$

let $w \leftarrow (\textsc{send}, (\textsc{simkey}, \bot))$

if $\exists(\mathsf{pk}'_z, \mathsf{pk}'_e, \mathsf{cm}) \in B\colon \mathsf{pk}'_z = \mathsf{prf}^{\mathsf{pk}}_{K_z}$ **then**

let $a \leftarrow (\mathsf{pk}'_z, \mathsf{pk}'_e)$

else abort

let $T(\mathsf{tx}) \leftarrow \{\mathsf{cm}'\}$

let $z \leftarrow \varnothing$

else return NONE

else if $w = \textsc{mint} \wedge \mathcal{T}_\sigma = ((\textsc{mint}, \mathsf{cm}), \varnothing)$ **then**

let $B \leftarrow B \cup \{(\textsc{simkey}, \bot, \mathsf{cm})\}$

let $T(\mathsf{tx}) \leftarrow \{\mathsf{cm}'\}$

let $z \leftarrow (\text{SIMKEY}, \perp); a \leftarrow \varnothing$

else return NONE

return (ψ, w, z, a, D)

Helper procedures:

procedure root(t)

let cms $\leftarrow \varnothing$

send READ **to** $\mathcal{G}_{\text{Ledger}}$ and **receive the reply** Σ

for tx $\in \Sigma[:t]$ **do**

let cms $\leftarrow$ cms $\cup\, T$(tx)

return merkleroot(cms)

Proof (sketch, of Theorem 6.2). To begin with, observe that from the collision resistance of PRFs, commitments, and sampling from $\{0,1\}^\kappa$, all coin commitments, serial numbers, and public keys will be unique with overwhelming probability.

The environment can perform the following primary actions: a) For any honest party, run (POST-QUERY, w). b) For any honest party, run (CHECK-QUERY, tx). c) For any party, run (SUBMIT, tx) against $\mathcal{G}_{\text{L}}$. d) For any party, run READ against $\mathcal{G}_{\text{L}}$. e) Run (ADVANCE, p, Σ') against $\mathcal{G}_{\text{L}}$, and f) for any party, run (EXTEND, Σ') against $\mathcal{G}_{\text{L}}$.

All but the first two of these are trivial. The simulator forwards all queries to $\mathcal{G}_{\text{L}}$ and the state of $\mathcal{G}_{\text{L}}$ depends on no other functionality (transactions "submitted" in the ideal functionality are only passed to the adversary). As a direct result, the state and return value of $\mathcal{G}_{\text{L}}$ follow the same distribution in both worlds, giving the environment no means to distinguish.

During the running of CHECK-QUERY the environment does have a significant additional means of input, in the form of being able to assign meaning to adversarial transactions *as they get executed for the first time*. It is sufficient to show the following: a) From the ideal-world leakage, the simulator can create indistinguishable real-world leakage. b) Ideal-world transactions have the same leakage descriptions sent to the environment (and are rejected under the same conditions). c) An invariant holds between the ideal and real-world contract state, such that it is preserved across both honest and adversarial transactions' transition function executions.

We omit the full detail of this invariant. To sketch the idea behind it, we must

prove that the following are preserved: The public keys recorded in the ideal contract state and the simulator must correspond directly to secret keys recorded in the real contract state and the same public keys are returned by the real contract. Furthermore, the coins held by honest parties in the real contract should be valid at any time and correspond directly to the balance of the same party in the ideal contract. Honest unconfirmed transactions in both the real and ideal contracts should still be valid when they are finally executed (also implying they do not conflict with each other).

These are preserved across honest INIT calls, as the simulator ensures the keys it stores and the public keys returned in the ideal contract are generated in the same way as in the real contract. They are preserved across honest SEND calls, as they remove one commitment from an honest party's coins and potentially add it to the respective recipient party. Furthermore, the leakage function of honest SENDs in the real contract ensures the same coin cannot be spent again. They are preserved across honest MINTs, as again the balance is incremented alongside a new coin being recorded. For adversarial transactions, as the simulator has all honest private keys, it can, and does, check if an honest party would register receiving a new coin. If a coin is sent, but no honest party receives it, the simulator records it as adversarial – even if it may not be spendable by the real-world adversary. Furthermore, the simulator manages which real-world coin commitments are associated with which adversarial public key in the ideal world. This ensures the simulator can always spend a corresponding ideal coin to whatever was spent in the real world (assuming the real world adversary does not spend a coin they do not own, violating the one-wayness of the PRF).

Transactions remaining valid in the ideal world is guaranteed by ensuring the balance of a party cannot fall below zero – by assuming the worst case of only balance removing transactions becoming confirmed. Likewise, in the real world, the coins eligible for spending are those received in confirmed transactions, but not spent in unconfirmed ones, ensuring they will not conflict. In both cases, key generation will be refused if one is currently unconfirmed. MINT and BALANCE queries both only require initialisation to have taken place in either world.

To observe that the simulator creates indistinguishable leakage, we first note that the leakage for real-world INIT transactions is an empty transcript, which the simulator indeed recreates. For SEND transactions, the simulator creates a public state transcript following the same structure of one in the real contract

execution – spending a coin, creating a new one, and sending a message. Here there are two cases: either the recipient is adversarial, or they are honest. In the case of an honest recipient, the simulator does not know the exact public key of the recipient. Fortunately, however, the environment does not know their secret keys for the same reason. As a result, it is sufficient to commit to an arbitrary coin and encrypt arbitrary secrets. Due to the hiding of the commitments and the key-privacy of the encryption scheme, the environment cannot distinguish this from a real transaction. The simulator creates a random serial number – revealing nothing due to collision resistance – and from the leakage of the length of the ledger, can reconstruct the corresponding Merkle tree root, revealing the same root as the corresponding real-world transaction.

If the adversary *is* the recipient, the simulator is given the actual public keys – and can use these directly as in the real protocol, creating a valid spendable commitment and a message the adversary can decrypt. Minting is similar to the case of sending to an honest party – except no message is encrypted. For the same reason, the leakage is indistinguishable. Finally, honest balance queries have no leakage in the real world.

For honest parties, the leakage descriptor the environment is asked to sign off on is identical – for INIT, MINT, and BALANCE consisting of just this string, and for SEND, it is (SEND, t, pk), where pk is the recipient, if it is adversarial, and otherwise is omitted. In each case, assertions made about the current and projected states are satisfied in either both worlds, or neither, ensuring the transaction is rejected or posted equally in both worlds. Specifically, all have tests for whether keys are initialised (asserting negatively in INIT and positively everywhere else). During spending, a positive spendable balance is also asserted in both worlds. These holding simultaneously is guaranteed by the invariant holding.

Finally, the transaction outputs the environment receives are the same in both worlds: For INIT, the simulator ensures it sees equally distributed public keys. For BALANCE, the equal distribution is guaranteed by the invariant. For all other messages, it will only see if they are in the ledger state – as the honest transactions cannot fail and return nothing. □

6.6 Expansions to KACHINA

KACHINA is intended as a basis to build more complex systems on. By itself it

has many limitations, however it presents a more flexible basis to interact with than the entirely public state machines of traditional smart contract systems. In this sections a few expansions are sketched, notably how the adversary's ability to arbitrarily set its private state can be constrained and how to model transactions which go beyond a single atomic interaction. This section also explores potential future work, discussing how the two-state model of KACHINA might be expanded.

6.6.1 Enforcing Private State Consistency

The protocol presented so far allows an adversary to arbitrarily set their own private state. Often it may be desirable to ensure that parties must follow the rules of the contract, even when it comes to the private state, however. This is possible, although it also introduces extra costs and has the caveat of not functioning with nondeterminism.

The core idea is to store commitments to private states within the public state of the contract. The contract itself can then verify that the private state is consistent with this commitment, update it, and then re-commit to the new state, proving the correctness of every step along the way. Clearly this adds more work to be verified about the contract, however a more worrying change is that again the contract needs to be able to process the entirety of the private contract state. Fortunately using slightly more complex updateable cryptographic datastructures, such as Merkle trees, can mitigate this problem – although it cannot be eliminated entirely, as computation which aggregates the entire private state will still be as costly.

6.6.2 Non-Atomic Executions

Smart contracts are typically closely linked to transactions made on the underlying ledger and indeed we explicitly make the same link in this chapter. That being said, there are numerous applications which do not rely on a single transaction per interaction with a contract, from Hawk [KMS+16], which requires at least two transactions per round of interaction, to state channels [DFH18], which have many of the same properties of smart contracts, but may (under optimal conditions) not require transactions at all.

While the model of smart contracts presented in Section 6.2 technically ex-

cludes both of these, and a full treatment of both would require further work, it is nonetheless worth considering how they can be – albeit imperfectly – embedded in this model. First, let us consider contract queries which require multiple on-chain interactions to "complete". As an example from Hawk, consider Alice posts a query to a Hawk-style contract. Naturally, this will not immediately return – even if Alice's transaction has made it on-chain. Instead, the transaction could return a "future object" – a concept often used in concurrent programming design, essentially just being a reference ID and a promise to associate some data with it later. Both Alice and the manager party would have to regularly poll the contract – for instance, send a contract query POLL every 10 minutes. On the manager's next POLL query, he would update the Hawk private state, and encrypt and post the result for Alice. Finally, when Alice next polls, she would retrieve the result and associate it with the previous "future object" as an output. This sketches a protocol running *on top of* KACHINA, which achieves this style of interaction. It is worth noting that this requirement for end-users to interact is also a limitation of the underlying model of universal composability: The environment must manually instruct parties to resume, or messages to be forwarded by the adversary.

In a similar vein, we can observe that some transactions need not be placed on a ledger. In particular, if the shared, public state is not used, the transaction is essentially "offchain" and there is no need to publicly post it. Furthermore, if the public state is used for message passing (such as in the construction of a Zerocash contract above), this part of the transaction need not be on-chain – sending an out of band message is cheaper. Using the same UC-based approach described above, it would therefore be possible, for example, to first define an ideal payment-channel contract and prove that this is UC-emulated by a contract implementing, for instance, Perun [DEFM19]. Finally, we can argue that most transactions in this contract can be omitted from the ledger, as they are just two-party channel interactions. This is a rather roundabout means of constructing off-chain communication, however it brings a crucial guarantee with it, namely that it behaves the same under ledger reorderings as a purely on-chain contract.

6.6.3 Meta-Parties and Alternative Trust Models

So far we have presented and argued for, a model of smart contracts with clear black-and-white privacy: Users have their own perfectly private local state and access to a perfectly public shared state. While we believe this to be the best starting point for approaching the issue of privacy in smart contracts, reality is not so simple: Often users have more complex relations with each other.

To consider this more carefully, we can consider that any piece of data in a smart contract must have a set of *owners* $\mathbb{O}$, who can interact with it. Furthermore, in any real system, there are parties which can, together, decipher the actual data itself and break the privacy of it. Let us refer to the set of all combinations of parties able to decipher the data as $\mathbb{T}$. While not strictly necessary, in general it is reasonable to assume that the owners are also the users able to break privacy, that is $\mathbb{T} \subseteq 2^{\mathbb{O}}$. While clearly there are many possible combinations here, a few stand out as interesting, and we observe that they all relate to some interpretation of privacy-preserving smart contracts:

- $\mathbb{O} = \mathcal{P}$, $\mathbb{T} = 2^{\mathcal{P}}$: This is the setting of Ethereum and of the σ used in this work. Data is public, but can be interacted with by all.
- $\mathbb{O} = \{\psi\}$, $\mathbb{T} = \{\psi\}$: This is the setting of ρ used in this work. Data is private, but cannot be interacted with by anyone else.
- $\mathbb{O} = \mathcal{P}$, $\mathbb{T}$ is all subsets of $\mathcal{P}$ with a resource majority (regardless of work, stake, or what other honest majority assumption is being made): This setting is feasible by running MPC across the honest majority of the underlying consensus protocol.
- $\mathbb{O}$ is a fixed-size set, $\mathbb{T} = \{m\}$: This is the setting of Hawk [KMS+16], in which a single party is trusted with privacy.
- $\mathbb{O}$ is a fixed-size set, $\mathbb{T} = \{\mathbb{O}\}$: This is the setting of privacy-preserving state-channels, in which parties run MPC out-of-band to agree on updates.
- $\mathbb{O}$ is a fixed-size set, $\mathbb{T} = 2^{\mathbb{O}}$: This is the setting of public state-channels, in which parties run Arbitrum-like protocols out-of-band to agree on updates.

In particular, this work only directly concerns itself with the first two of these. It is clear, however that different problems call for different solutions, and ideally a smart contract system would encode all of these trust systems, not just one, or a few. Part of the reason for the choice of the first two is that they are sufficient for constructing the rest, being the exteremes of the spectrum.

The case of Hawk, for instance, was already described in Subsection 6.6.2. We will sketch how state channels might be modelled on top of KACHINA, although we stress that a full formal treatment of this and other settings will be left for future work.

A state channel between two users can be interpreted as the two users constituting a "metaparty" – a single entity consisting of multiple parties. This is subject to some access control for when the constituent parties can act on behalf of both – commonly requiring agreement from all constituent parties. If Alice and Bob open a new state channel, this can be seen as creating a new combined party of (Alice, Bob).

In KACHINA, this party again has its own private state, and for state channels, this can track the most recent update of the channel. Updates are now operations that only affect the private state of this combined party and as argued in Subsection 6.6.2 can be left off the ledger entirely. Interestingly, the access structure for closing channels and reading the current state is more permissive in most state or payment channels – requiring only one user to initiate it.

Given a state channel system, most of it can be implemented in a KACHINA smart contract. It is not new for state or payment channels to use smart contracts, however this is typically only for the opening and closing of the channel. We observe that in KACHINA the update of the channel can also be modelled.

This approach of metaparties is useful, but not optimal. For instance, a contract cannot interact with both Alice's private state and the state channel between Alice and Bob at the same time, as presented here. Furthermore, how the constituents of a metaparty reach consensus on whether an action is permitted or not is unclear and varies from case to case. We leave as future work how to give first-class treatment to data owned by multiple parties.

6.7 Smart Contract Systems

To construct complex systems of multiple smart contracts, no additional machinery is required. In this section, we incrementally construct a complex system with similar functionality to Ethereum [Woo14]. We begin by multiplexing between a fixed set of transition functions, and expand this with the ability to allow new transition functions to be registered, transition functions to call each other, registered contracts to hold and transfer funds, and combine in a setting where computation has an associated cost, which must be paid by the caller. We finally show how access to the underlying ledger may be modelled.

It is worth noting that we concern ourselves only with the "real world" of KACHINA core contracts. A reasonable question is how to transfer a proof such as the one we presented in Section 6.5 into this setting. While we do not go into the details here, we observe that (with one exception for the specific token contract used), only the smart contract's own transition function affects its state. Running a multiplexed smart contract is equivalent to running many small smart contracts independently – only interpreting the ledger differently. This is no longer true once contracts may call each other – in which case it is sufficient to reason about the closure of contracts able to call each other instead.

In this section we will assume that the (sub-)contracts do not make use of COMMIT messages. While this mechanism can be accounted for, it is simpler to present without it, and the primary purpose of COMMITs in the first place is to enable gas payments – which this section does.

6.7.1 Multiplexing Contracts

The basic multiplexing contract takes n different sub-contracts as inputs. Each party supplies not only the input, but the index i of the contract they wish to call. The public and private states of the multiplexer consist of the product of the corresponding sub-contract states and oracle queries are re-written to address the correct part of the state. To do some, new oracles $\mathcal{O}'_\sigma$ and $\mathcal{O}'_\rho$ are constructed, which rewrite queries made to them. Then, the requested transition function is run with these oracles, instead of the original ones.

Transition Function Γ_{mux}

The multiplexing transition function Γ_{mux} is parameterised by n transition functions $\Gamma_1, \ldots, \Gamma_n$ and allows a user to address any one of them.

Public state variables and initialisation values:

Variable	Description
$\sigma_i \coloneqq \varnothing$	Public states for each sub-contract

---

Private state variables and initialisation values:

Variable	Description
$\rho_i \coloneqq \varnothing$	Private states for each sub-contract

When receiving an input (i, w):

assert $i \in \mathbb{Z}_n$
let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathsf{muxPubOracle}(i, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathsf{muxPrivOracle}(i, q))$
return $\Gamma_{i,\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(w)$

Helper procedures:

function muxPubOracle$(i, q, \sigma, \varnothing)$
  let $\sigma' \leftarrow \sigma.\sigma_i$
  let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
  if $\sigma' = \bot$ **then return** $(\bot, y)$
  else
    let $\sigma.\sigma_i \leftarrow \sigma'$
    return (σ, y)

function muxPrivOracle$(i, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  let $\rho' \leftarrow \rho.\rho_i$
  let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
  let $(\rho', y) \leftarrow q(\rho', z')$
  if $\rho' = \bot$ **then return** $(\bot, y)$
  else
    let $\rho.\rho_i \leftarrow \rho'$
    return (ρ, y)

We assume the existence of unmuxPubOracle and unmuxPrivOracle, which take an oracle transcript to an Oracle produced by a multiplexed oracle and

return the pair $(i, \mathcal{T}')$, where i is the address used in the original multiplexing and $\mathcal{T}'$ is the equivalent un-multiplexed transcript.

function $\mathsf{unmuxZ}_{\mathsf{mux}}((\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), i)$
  return $(\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$

function $\mathsf{unmuxX}_{\mathsf{mux}}(X, i)$
  let $X' \leftarrow \varepsilon$
  for $(u, \mathcal{T}, z, D)$ **in** X **do**
    if $\exists \mathcal{T}' : \mathsf{unmuxPrivOracle}(\mathcal{T}) = (i, \mathcal{T}')$ **then**
      let $X' \leftarrow X' \parallel (u, \mathcal{T}', \mathsf{unmuxZ}_{\mathsf{mux}}(z, i), D)$
  return X'

function $\mathsf{desc}_{\mathsf{mux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (i, w), z)$
  let $(\cdot, \mathcal{T}_\sigma') \leftarrow \mathsf{unmuxPubOracle}(\mathcal{T}_\sigma)$
  let $(\cdot, \mathcal{T}_\rho') \leftarrow \mathsf{unmuxPrivOracle}(\mathcal{T}_\rho)$
  let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{mux}}(X, i); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{mux}}(z, i)$
  return "Calling sub-contract i: "+$\mathsf{desc}_i(t, X', \mathcal{T}_\sigma', \mathcal{T}_\rho', w, z')$

function $\mathsf{dep}_{\mathsf{mux}}(X, \mathcal{T}_\rho, z)$
  if $\mathcal{T}_\rho = \varepsilon$ **then return** $\varnothing$
  else
    let $(i, \mathcal{T}_\rho') \leftarrow \mathsf{unmuxPrivOracle}(\mathcal{T}_\rho)$
    let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{mux}}(X, i); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{mux}}(z, i)$
    return $\mathsf{dep}_i(X', \mathcal{T}_\rho', z')$

6.7.2 Multiplexing with Registration

To allow registering new contracts in the multiplexer, it is possible to include the full contract's description as part of its *address* A. In practice it may make more sense to maintain a mapping from addresses to contract code, however this is not required. The only other large change is that, since contracts are created on the fly, we cannot rely on their states to have been initialised at any point. Therefore, this initialisation takes place at any point where the multiplexed state is accessed.

function $\mathsf{forceInitMaps}(((M_1, \ldots, M_n), k, v))$
  for $i \in \{1, \ldots, n\}$ **do**
    if $k \notin M_i$ **then let** $M_i(k) \leftarrow v$
  return $(M_1, \ldots, M_n)$

Transition Function Γ_{regmux}

The multiplexing with registration transition function Γ_{regmux} allows addressing any pair of address and sub-transition function (A, Γ). It uses the specified transition function on whatever state is associated with this pair, or a new, empty state for the first use.

Public state variables and initialisation values:

Variable	Description
$\Sigma \coloneqq \varnothing$	Mapping from address pairs to public states

---

Private state variables and initialisation values:

Variable	Description
$\mathrm{P} \coloneqq \varnothing$	Mapping from address pairs to private states

When receiving an input $(A = (i, \Gamma, \text{desc}, \text{dep}), w)$*:*

let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\text{muxPubOracle}(A, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\text{muxPrivOracle}(A, q))$
return $\Gamma_{\mathcal{O}'_\sigma, \mathcal{O}'_\rho}(w)$

Helper procedures:

function muxPubOracle$(A, q, \sigma, \varnothing)$
  if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
  let $\sigma' \leftarrow \sigma.\Sigma(A)$
  let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
  if $\sigma' = \bot$ **then return** $(\bot, y)$
  else
    let $\sigma.\Sigma(A) \leftarrow \sigma'$
    return (σ, y)

function muxPrivOracle$(A, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  let $(\rho.\mathrm{P}, \sigma^o.\Sigma.\rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}) \leftarrow$ forceInitMaps(
    $(\rho.\mathrm{P}, \sigma^o.\Sigma, \rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}), A, \varnothing)$
  let $\rho' \leftarrow \rho.\mathrm{P}(A)$
  let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
  let $(\rho', y) \leftarrow q(\rho', z')$
  if $\rho' = \bot$ **then return** $(\bot, y)$
  else
    let $\rho.\mathrm{P}(A) \leftarrow \rho'$

return (ρ, y)

We assume the existence of unmuxPubOracle and unmuxPrivOracle, which take an oracle transcript to an Oracle produced by a multiplexed oracle and return the pair $(A, \mathcal{T}')$, where $A = (i, \Gamma, \text{desc}, \text{dep})$ is the address used in the original multiplexing and $\mathcal{T}'$ is the equivalent un-multiplexed transcript.

function $\text{unmuxZ}_{\text{regmux}}((\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), A)$
  let $(\sigma^o.\Sigma, \rho^o.\text{P}, \sigma^\pi.\Sigma, \rho^\pi.\text{P}) \leftarrow \text{forceInitMaps}($
    $(\sigma^o.\Sigma, \rho^o.\text{P}, \sigma^\pi.\Sigma, \rho^\pi.\text{P}), A, \varnothing)$
  return $(\sigma^o.\Sigma(A), \rho^o.\text{P}(A), \sigma^\pi.\Sigma(A), \rho^\pi.\text{P}(A), \eta)$

function $\text{unmuxX}_{\text{regmux}}(X, A)$
  let $X' \leftarrow \varepsilon$
  for $(u, \mathcal{T}, z, D)$ **in** X **do**
    if $\exists \mathcal{T}'$: unmuxPrivOracle$(\mathcal{T}) = (A, \mathcal{T}')$ **then**
      let $X' \leftarrow X' \parallel (u, \mathcal{T}', \text{unmuxZ}_{\text{regmux}}(z, A), D)$
  return X'

function $\text{desc}_{\text{regmux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (A = (\cdot, \cdot, \text{desc}, \cdot), w), z)$
  let $(\cdot, \mathcal{T}'_\sigma) \leftarrow \text{unmuxPubOracle}(\mathcal{T}_\sigma)$
  let $(\cdot, \mathcal{T}'_\rho) \leftarrow \text{unmuxPrivOracle}(\mathcal{T}_\rho)$
  let $X' \leftarrow \text{unmuxX}_{\text{regmux}}(X, A); z' \leftarrow \text{unmuxZ}_{\text{regmux}}(z, A)$
  return "Calling sub-contract A: "+desc$(t, X', \mathcal{T}'_\sigma, \mathcal{T}'_\rho, w, z')$

function $\text{dep}_{\text{regmux}}(X, \mathcal{T}_\rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  if $\mathcal{T}_\rho = \varepsilon$ **then return** $\varnothing$
  else
    let $(A = (\ldots, \text{dep}), \mathcal{T}'_\rho) \leftarrow \text{unmuxPrivOracle}(\mathcal{T}_\rho)$
    let $(\sigma^o.\Sigma.\rho^o.\text{P}, \sigma^\pi.\Sigma, \rho^\pi.\text{P}) \leftarrow$
      forceInitMaps$((\sigma^o.\Sigma, \rho^o.\text{P}, \sigma^\pi.\Sigma, \rho^\pi.\text{P}), A, \varnothing)$
    let $z' \leftarrow (\sigma^o.\Sigma(A), \rho^o.\text{P}(A), \sigma^\pi.\Sigma(A), \rho^\pi.\text{P}(A), \eta)$
    let $X' \leftarrow \varepsilon$
    for $(u, \mathcal{T}_\rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), D)$ **in** X **do**
      if $\exists \mathcal{T}'_\rho$: unmuxPrivOracle$(\mathcal{T}_\rho) = (A, \mathcal{T}'_\rho)$ **then**
        let $(\sigma^o.\Sigma.\rho^o.\text{P}, \sigma^\pi.\Sigma, \rho^\pi.\text{P}) \leftarrow \text{forceInitMaps}($
          $(\sigma^o.\Sigma, \rho^o.\text{P}, \sigma^\pi.\Sigma, \rho^\pi.\text{P}), A, \varnothing)$
        let $X' \leftarrow X' \parallel$
          $(u, \mathcal{T}'_\rho, (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta), D)$
    return $\text{dep}(X', \mathcal{T}'_\rho, z')$

6.7.3 Loopback Multiplexing

Smart contract systems truly become interesting when contracts are allowed to *call each other*. This is not a technically difficult operation: Contracts simply need to have an additional exit and entry point to allow new queries to other contracts to be made, and these queries to be responded to. Specifically, we require contracts to *either* return (RETURN, y), *or* (CALL, A, M), with the latter invoking a separate contract. We associate a special return value structure with indicating a new contract address and input to call, and require contracts to process a specific RESUME message.

As for the first time, it is possible for multiple separate contracts to get called, we domain-separate the randomness source η.

function $\mathsf{unmuxZ}_{\mathsf{loopmux}}((\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta), A)$

  let $(\sigma^o.\Sigma, \rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}) \leftarrow \mathsf{forceInitMaps}((\sigma^o.\Sigma, \rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}), A, \varnothing)$

  Let η' be a randomness source determinstically and collision-resistantally derived from the pair (η, A).

  return $(\sigma^o.\Sigma(A), \rho^o.\mathrm{P}(A), \sigma^\pi.\Sigma(A), \rho^\pi.\mathrm{P}(A), \eta')$

Transition Function $\Gamma_{\mathsf{loopmux}}$

The multiplexing with registration and loopback transition function $\Gamma_{\mathsf{loopmux}}$ allows addressing any pair of address and sub-transition function (A, Γ). These sub-transition functions may return values of either (CALL, A, M), or (RETURN, y). In the former case, a different sub-transition function is invoked and the value it eventually returns is fed back into the original one, by re-invoking it with (RESUME, y).

Public state variables and initialisation values:

Variable	Description
$\Sigma \coloneqq \varnothing$	Mapping from address pairs to public states

Private state variables and initialisation values:

Variable	Description
$\mathrm{P} \coloneqq \varnothing$	Mapping from address pairs to private states

When receiving an input $(A = (i, \Gamma, \mathrm{desc}, \mathrm{dep}), w)$*:*

  let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathsf{muxPubOracle}(A, q))$

  let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathsf{muxPrivOracle}(A, q))$

  repeat

let $y \leftarrow \Gamma_{\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(w)$
if $\exists A', M : y = (\textsc{call}, A', M)$ **then**
let $w \leftarrow \left(\textsc{resume}, \Gamma_{\mathsf{loopmux},\mathcal{O}_\sigma,\mathcal{O}_\rho}((A', M))\right)$
until $\exists y' : y = (\textsc{return}, y')$
return y'

Helper procedures:

function muxPubOracle$(A, q, \sigma, \varnothing)$
if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
let $\sigma' \leftarrow \sigma.\Sigma(A)$
let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
if $\sigma' = \bot$ **then return** $(\bot, y)$
else
let $\sigma.\Sigma(A) \leftarrow \sigma'$
return (σ, y)

function muxPrivOracle(A, q, ρ, z)
let $z' \leftarrow \mathsf{unmuxZ}_{\mathsf{loopmux}}(z, A)$
if $A \notin \rho.\mathrm{P}$ **then let** $\rho.\mathrm{P}(A) \leftarrow \varnothing$
let $\rho' \leftarrow \rho.\mathrm{P}(A)$
let $(\rho', y) \leftarrow q(\rho', z')$
if $\rho' = \bot$ **then return** $(\bot, y)$
else
let $\rho.\mathrm{P}(A) \leftarrow \rho'$
return (ρ, y)

Unlike before, we cannot invert the multiplexing on an entire transcript, as the transcript may consist of multiple separate sub-contract calls. Instead, we can invert multiplexing each query/response pair in the transcript itself. We assume the existence of unmuxOracle, which takes a query-response pair (q, r), where the query is muxPubOracle(A, q') or muxPrivOracle(A, q'), and maps it to $(A, (q', r))$.

As far as descriptions go, it is crucial to note that the leakage description of a contract is no longer in isolation: what the contract may leak, depends on what this contract calls. We will assume instead that each sub-contract's leakage descriptor is aware that it is being run in a loopback system – and therefore we give it the full transcripts, even of sub-contracts being called. The assumption here

is that the contract directly called by the user is also trusted by this user – the descriptor it gives should be trusted, not necessarily that of any further contracts it invoked. It is worth noting that this change of setting for the descriptor function does not preclude using contracts designed without loopback systems in mind: As this cannot invoke other contracts, their old descriptor function can be easily lifted to this setting (a slight caveat is that either the old descriptor needs to be capable of tolerating unconfirmed transaction transcripts over multiple calls to the underlying function, or there should exist a function which splits transcripts into these individual calls).

function liftDesc(A, desc)($t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z$)
  let $\mathcal{T}'_\sigma \leftarrow$ map($\mathsf{proj}_2 \circ$ unmuxOracle, $\mathcal{T}_\sigma$)
  let $\mathcal{T}'_\rho \leftarrow$ map($\mathsf{proj}_2 \circ$ unmuxOracle, $\mathcal{T}_\rho$)
  let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{regmux}}(X, A); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{regmux}}(z, A)$
  return desc($t, X', \mathcal{T}'_\sigma, \mathcal{T}'_\rho, w, z'$)

function unmuxT($\mathcal{T}, A$)
  return map(proj_2, filter($\lambda(A', \cdot)$: $A = A'$,
    map(unmuxOracle, $\mathcal{T}$)))

function $\mathsf{unmuxX}_{\mathsf{loopmux}}(X, A)$
  let $X' \leftarrow \varepsilon$
  for $(u, \mathcal{T}, z, D)$ **in** X **do**
    let $\mathcal{T}' \leftarrow \mathsf{unmuxT}(\mathcal{T}, A); z' \leftarrow \mathsf{unmuxZ}_{\mathsf{loopmux}}(z, A)$
    let $X' \leftarrow X' \parallel (u, \mathcal{T}', z', D')$
  return X'

function $\mathsf{desc}_{\mathsf{loopmux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (A = (\cdot, \cdot, \mathsf{desc}, \cdot), w), z)$
  return "Calling sub-contract A: "+desc($t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z$)

function $\mathsf{dep}_{\mathsf{loopmux}}(X, \mathcal{T}_\rho, z)$
  let $S \leftarrow \varnothing$
  for (q, r) **in** $\mathcal{T}_\rho$ **do**
    let $(A, \cdot) \leftarrow$ unmuxOracle((q, r))
    let $S \leftarrow S \cup \{A\}$
  let $D \leftarrow \varnothing$
  for $A = (\cdot, \cdot, \cdot, \mathsf{dep})$ **in** S **do**
    let $\mathcal{T}'_\rho \leftarrow \mathsf{unmuxT}(\mathcal{T}_\rho, A)$
    let $z' \leftarrow \mathsf{unmuxZ}_{\mathsf{loopmux}}(z, A)$
    let $X' \leftarrow \mathsf{unmuxX}_{\mathsf{loopmux}}(X, A)$

let $D \leftarrow D \cup \mathsf{dep}(X', \mathcal{T}'_\rho, z')$

return $\mathsf{map}(\mathsf{proj}_1, X) \cap D$

6.7.4 Integrated Payments Systems

Smart contract systems typically have an associated, native "asset", which can be traded not only by users, but by contracts as well. This asset is typically tied to a public key, which can be used as an identity of end users, providing a means to authenticate to contracts. We demonstrate a simple means of achieving this: We construct a "simple payments" contract, which allows payments by end users through demonstrating knowledge of secret keys, and arbitrary payments which will be restricted to system usage. It is worth noting that this *could* be done in a privacy-preserving manner, as presented in Section 6.5, although significant changes would have to be made, as there would be situations where a contract should publicly own funds and be able to transfer them, and the simplified single-denomination design is not ideal.

Transition Function Γ_{sp}

The state transition function for a simple payments system. Parties have associated public/private keys and balances. The payments system allows for parties without a key pair to generate one, and for parties to transfer and mint coins, as well as query their own balance.

Public state variables and initialisation values:

Variable	Description
$B \coloneqq \lambda \mathsf{pk}.\, 0$	Mapping of public keys to their spendable coins

Private state variables and initialisation values:

Variable	Description
$\mathsf{sk} \coloneqq \varnothing$	The party's secret key

When receiving an input INIT*:*

send INIT **to** $\mathcal{O}_\rho$ and **receive the reply** sk

let $\mathsf{pk} \leftarrow \mathsf{prf}^{\mathsf{pk}}_{\mathsf{sk}}(1)$

return pk

When receiving an input (SEND, recv, v)*:*

send SECRETKEY **to** $\mathcal{O}_\rho$ and **receive the reply** sk

let pk $\leftarrow \mathrm{prf}^{\mathrm{pk}}_{\mathrm{sk}}(1)$

send (SEND, pk, recv, v) **to** $\mathcal{O}_\sigma$

return pk

When receiving an input (SYSTEM-SEND, snd, recv, v)*:*

send (SEND, snd, recv, v) **to** $\mathcal{O}_\sigma$

When receiving an input (MINT, v)*:*

send SECRETKEY **to** $\mathcal{O}_\rho$ and **receive the reply** sk

let pk $\leftarrow \mathrm{prf}^{\mathrm{pk}}_{\mathrm{sk}}(1)$

send (MINT, pk, v) **to** $\mathcal{O}_\sigma$

When receiving an input BALANCE*:*

send BALANCE **to** $\mathcal{O}_\rho$

When receiving a private oracle query INIT*:*

assert $\rho^\pi.\mathrm{sk} = \varnothing$

let $\rho.\mathrm{sk} \xleftarrow{*} \{0,1\}^\kappa$

return $\rho.\mathrm{sk}$

When receiving a private oracle query SECRETKEY*:*

return $\rho.\mathrm{sk}$

When receiving a private oracle query BALANCE*:*

return $\sigma^\pi.B(\mathrm{prf}^{\mathrm{pk}}_{\rho^\pi.\mathrm{sk}}(1))$

When receiving a public oracle query (SEND, pk, recv, v)*:*

assert $\sigma.B(\mathrm{pk}) \geq v$

let $\sigma.B(\mathrm{pk}) \leftarrow \sigma.B(\mathrm{pk}) - v$

let $\sigma.B(\mathrm{recv}) \leftarrow \sigma.B(\mathrm{recv}) + v$

When receiving a public oracle query (MINT, pk, v)*:*

let $\sigma.B(\mathrm{pk}) \leftarrow \sigma.B(\mathrm{pk}) + v$

function $\mathrm{desc}_{\mathrm{sp}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z)$

  if $\mathcal{T}_\sigma = (\mathrm{INIT}, \mathrm{pk})$ **then**

    return (INIT, pk)

  else if $\mathcal{T}_\sigma = ((\mathrm{SEND}, \mathrm{snd}, \mathrm{recv}, v), \cdot)$ **then**

    return (SEND, snd, recv, v)

  else if $\mathcal{T}_\sigma = ((\mathrm{MINT}, \mathrm{pk}, v), \cdot)$ **then**

    return (MINT, pk, v)

else return $\bot$

function $\mathrm{dep}_{\mathrm{sp}}(X, \mathcal{T}, z)$

return ε

Once given such a payments system, the multiplexing system can ensure that for each call, a transfer to the called contract is initiated *first*, with the value of the transfer and the source address being passed into the contract being called. Likewise, if this calls another contract, this call may transfer funds from one contract to another.

Transition Function Γ_{paymux}

The multiplexing with registration, loopback, and payments transition function Γ_{paymux} allows addressing any pair of address and sub-transition function (a, Γ). These sub-transition functions may return values of either $(\textsc{call}, A, M)$, or $(\textsc{return}, y)$. In the former case, a different sub-transition function is invoked and the value it eventually returns is fed back into the original one, by re-invoking it with $(\textsc{resume}, y)$.

Public state variables and initialisation values:

Variable	Description
$\Sigma \coloneqq \varnothing$	Mapping from address pairs to public states

---

Private state variables and initialisation values:

Variable	Description
$\mathrm{P} \coloneqq \varnothing$	Mapping from address pairs to private states

When receiving an input $(\textsc{token}, w)$*:*

assert $w \neq (\textsc{system-send}, \ldots)$

let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathrm{muxPubOracle}(\textsc{token}, q))$

let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathrm{muxPrivOracle}(\textsc{token}, q))$

return $\Gamma_{\mathrm{sp},\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(w)$

When receiving an input $(\textsc{call}, v, A = (i, \Gamma, \mathrm{desc}, \mathrm{dep}), w)$*:*

let $\mathrm{pk} \leftarrow \Gamma_{\mathrm{paymux},\mathcal{O}_\sigma,\mathcal{O}_\rho}(\textsc{token}, (\textsc{send}, A, v))$

return $\mathrm{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, \mathrm{pk}, A, w)$

Helper procedures:

function $\mathrm{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (i, \Gamma, \mathrm{desc}, \mathrm{dep}), w)$

assert $A' \neq \textsc{token}$

let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathsf{muxPubOracle}(\textsc{token}, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathsf{muxPrivOracle}(\textsc{token}, q))$
run $\Gamma_{\mathsf{sp},\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(\textsc{system-send}, A, A', v)$
return $\mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A', w)$

function $\mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (\cdot, \Gamma, \cdot, \cdot), w)$
let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathsf{muxPubOracle}(A', q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathsf{muxPrivOracle}(A', q))$
repeat
let $y \leftarrow \Gamma_{\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(\textsc{call}, A, v, w)$
if $\exists v', A'', w'\colon y = (\textsc{call}, v', A'', w')$ **then**
let $w \leftarrow (\textsc{resume}, \mathsf{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v', A', A'', w'))$
until $\exists y'\colon y = (\textsc{return}, y')$
return y'

function $\mathsf{muxPubOracle}(A, q, \sigma, \varnothing)$
if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
let $\sigma' \leftarrow \sigma.\Sigma(A)$
let $(\sigma', y) \leftarrow q(\sigma', \varnothing)$
if $\sigma' = \bot$ **then return** $(\bot, y)$
else
let $\sigma.\Sigma(A) \leftarrow \sigma'$
return (σ, y)

function $\mathsf{muxPrivOracle}(A, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
let $(\rho.\mathrm{P}, \sigma^o.\Sigma.\rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}) \leftarrow \mathsf{forceInitMaps}((\rho.\mathrm{P}, \sigma^o.\Sigma, \rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}), A, \varnothing)$
let $\rho' \leftarrow \rho.\mathrm{P}(A)$
let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
let $(\rho', y) \leftarrow q(\rho', z')$
if $\rho' = \bot$ **then return** $(\bot, y)$
else
let $\rho.\mathrm{P}(A) \leftarrow \rho'$
return (ρ, y)

function $\mathsf{desc}_{\mathsf{paymux}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, M, z)$
if $\exists w\colon M = (\textsc{token}, w)$ **then**
return "Calling token contract:"+$\mathsf{desc}_{\mathsf{sp}}(t, X, \mathsf{map}(\mathsf{proj}_2 \circ \mathsf{unmuxOracle}, \mathcal{T}_\sigma),$
$\mathsf{map}(\mathsf{proj}_2 \circ \mathsf{unmuxOracle}, \mathcal{T}_\rho), w, z)$
else if $\exists v, A = (\cdot, \cdot, \mathsf{desc}, \cdot), w\colon M = (\textsc{call}, v, A, w)$ **then**

return "Calling sub-contract A with pay-in v: "+desc$(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, (\bot, v, w), z)$

else return $\bot$

$\mathsf{dep}_{\mathsf{paymux}} = \mathsf{dep}_{\mathsf{loopmux}}$

6.7.5 Fees and Cost Models

In order to prevent denial-of-service attacks, the computations performed by the network in verifying a transaction must be paid for in some way. In public currencies, there is typically a *cost model*, which maps each step of computation to a cost, often referred to as *gas*. Each transaction declares a limit on how much gas it is willing to pay and what each unit's value should be. It then pays the corresponding amount into a fee pool and, while executing the transaction, the gas usage is counted. If the limit is reached, the transaction is rejected, otherwise any spare gas is refunded.

We do not explicitly specify how miners are awarded these fees – a simple approach is to not enable withdrawals from the fee pot within the transition function, relying on miners to do so themselves, and not include in their block other transactions which take from the pot.

In Kachina the computation done in public state oracles occupies a similar space: A modelling of fees must include estimating their likely cost, pay this estimation in advance, and then use up the gas during the actual oracle execution. In addition to this, the NIZK proof verification must be paid for. We will assume that this has a flat cost, dependant on the size of its inputs, that is, the size of the transcript.

Specifically, we assume two cost models: $\$_{\mathsf{zk}}$ and $\$_{\mathsf{std}}$, as well as a cost estimator $\mathrm{E}_{\mathsf{std}}$. $\$_{\mathsf{zk}}$ is simply a function from a public state transcript to the gas cost of verifying a NIZK proof against it. A transaction will publicly declare what it believes the cost of its transcript is and will use $\mathrm{E}_{\mathsf{std}}$ (as well as a user input dictating the cost per unit of gas) to estimate the cost of the remaining transaction. The transaction declares this total fee, which part of the fee is for the NIZK verification, and what the cost per unit of gas is. Transactions which pay too little for NIZK verification, or set the cost per unit of gas too low, may not be picked up by miners, although modelling miner incentives is not within the scope of this thesis.

Formally, $g \leftarrow \$_{\mathsf{zk}}(\mathcal{T})$ is a function from a public state transcript to a gas cost,

$(\sigma', g', y) \vee \bot \leftarrow \$_{\text{std}}(q, \sigma, g)$ is a function taking an oracle query, initial state, and gas limit, either returning the result and remaining gas, or returning $\bot$ if the supplied gas ran out. Finally, $(\sigma', g', y) \leftarrow \mathrm{E}_{\text{std}}(q, \sigma)$ returns an estimate as to the gas cost of running a query q, with the state σ as a reference point. E_{std} and $\$_{\text{std}}$ should return $(\sigma', y) = q(\sigma)$ if they succeed.

In attaching fees to our contract system, we operate as follows:

1. In the private state oracle, simulate the transaction creation process, constructing the public state transcript $\mathcal{T}$ and, for each public state interaction, recording the estimated cost, totalling to the overall gas cost g.

2. For a given gas price gasPrice, make two separate, public transfers into the fee pot: first $\$_{\text{zk}}(\mathcal{T}) \times \text{gasPrice}$ and second $g \times \text{gasPrice}$

3. Commit this as a partial execution success.

4. Execute the transaction as normal, except making public state oracle queries through a modified gas cost oracle instead, retaining a temporary state of the remaining gas.

5. Finally, the public state oracle relinquishes the remaining gas and returns it to the transaction creator.

We now give an example transition function that combines this gas model with the integrated payment system of Subsection 6.7.4.

Transition Function Γ_{scs}

The multiplexing with registration, loopback, payments, and fees transition function Γ_{scs} allows addressing any pair of address and sub-transition function (a, Γ). These sub-transition functions may return values of either $(\textsc{call}, A, M)$, or $(\textsc{return}, y)$. In the former case, a different sub-transition function is invoked, and the value it eventually returns is fed back into the original one, by re-invoking it with $(\textsc{resume}, y)$. The transition function first estimates the cost of this call, and pays for it in advance. This payment is then deduced from for executions, until a remainder is refunded at the end of a successful call.

Public state variables and initialisation values:

Variable	Description
$\Sigma \coloneqq \varnothing$	Mapping from address pairs to public states
spare $\coloneqq 0$	Temporary book-keeping of the value to return

---

Private state variables and initialisation values:

Variable	Description
$\mathrm{P} \coloneqq \varnothing$	Mapping from address pairs to private states

When receiving an input $(\textsc{token}, w)$*:*

assert $w \neq (\textsc{system-send}, \ldots)$
let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathrm{muxPubOracle}(\textsc{token}, q))$
let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathrm{muxPrivOracle}(\textsc{token}, q))$
return $\Gamma_{\mathrm{sp},\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(w)$

When receiving an input $(\textsc{call}, \mathrm{gasPrice}, v, A = (i, \Gamma, \mathrm{desc}, \mathrm{dep}), w)$*:*

let $\mathrm{pk} \leftarrow \Gamma_{\mathrm{scs},\mathcal{O}_\sigma,\mathcal{O}_\rho}(\textsc{token}, (\textsc{send}, A, v))$
send $(\textsc{estimate-cost}, v, \mathrm{pk}, A, w)$ **to** $\mathcal{O}_\rho$ and
  receive the reply $(g_\mathcal{T}, g_\mathcal{O})$
run $\Gamma_{\mathrm{scs},\mathcal{O}_\sigma,\mathcal{O}_\rho}(\textsc{token}, (\textsc{send}, \textsc{fee-pot}, g_\mathcal{T} \times \mathrm{gasPrice}))$
run $\Gamma_{\mathrm{scs},\mathcal{O}_\sigma,\mathcal{O}_\rho}(\textsc{token}, (\textsc{send}, \textsc{fee-pot}, g_\mathcal{O} \times \mathrm{gasPrice}))$
commit $\textsc{gas-paid}$
send $(\textsc{init-gas}, g_\mathcal{O})$ **to** $\mathcal{O}_\sigma$
let $y \leftarrow \mathrm{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, \mathrm{pk}, A, w)$
send $(\textsc{deinit}, \mathrm{pk}, \mathrm{gasPrice})$ **to** $\mathcal{O}_\sigma$
return y

When receiving a public oracle query $(\textsc{init-gas}, g_\mathcal{O})$*:*

let $\sigma.\mathrm{spare} \leftarrow g_\mathcal{O}$

When receiving a public oracle query $(\textsc{deinit}, \mathrm{pk}, \mathrm{gasPrice})$*:*

run $\Gamma_{\mathrm{sp}}(\textsc{fee-pot}, \mathrm{pk}, \sigma.\mathrm{spare} \times \mathrm{gasPrice})$
let $\sigma.\mathrm{spare} \leftarrow 0$

When receiving a private oracle query $(\textsc{estimate-cost}, v, \mathrm{pk}, A, w)$*:*

let $\mathcal{O}'_\sigma \leftarrow \mathcal{O}((\sigma^\pi, \varepsilon, 0); \mathcal{O}'_\rho \leftarrow \mathcal{O}(\rho^\pi)$
let $\mathcal{O}''_\sigma \leftarrow \lambda q\colon \mathcal{O}'_\sigma(\mathrm{muxEst}(q))$
run $\mathrm{call}_{\mathcal{O}''_\sigma,\mathcal{O}'_\rho}(v, \mathrm{pk}, A, w)$
let $(\cdot, \mathcal{T}, g) \leftarrow \mathrm{state}(\mathcal{O}_\sigma)$

return $(\$_{\mathsf{zk}}(\mathcal{T}), g)$

Helper procedures:

function $\mathsf{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (i, \Gamma, \mathsf{desc}, \mathsf{dep}), w)$
  assert $A' \neq \textsc{token}$
  let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathsf{muxPubOracle}(\textsc{token}, q))$
  let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathsf{muxPrivOracle}(\textsc{token}, q))$
  run $\Gamma_{\mathsf{sp},\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(\textsc{system-send}, A, A', v)$
  let $y \leftarrow \mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A', w)$
  return y

function $\mathsf{call}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v, A, A' = (\cdot, \Gamma, \cdot, \cdot), w)$
  let $\mathcal{O}'_\sigma \leftarrow \lambda q\colon \mathcal{O}_\sigma(\mathsf{muxPubOracle}(A', q))$
  let $\mathcal{O}'_\rho \leftarrow \lambda q\colon \mathcal{O}_\rho(\mathsf{muxPrivOracle}(A', q))$
  repeat
    let $y \leftarrow \Gamma_{\mathcal{O}'_\sigma,\mathcal{O}'_\rho}(\textsc{call}, A, y, w)$
    if $\exists v', A'', w'\colon y = (\textsc{call}, v', A'', w')$ **then**
      let $y \leftarrow \mathsf{subCall}_{\mathcal{O}_\sigma,\mathcal{O}_\rho}(v', A', A'', w')$
      let $w \leftarrow (\textsc{resume}, y)$
  until $\exists y'\colon y = (\textsc{return}, y')$
  return y

function $\mathsf{muxPubOracle}(A, q, \sigma, \varnothing)$
  if $A \notin \sigma.\Sigma$ **then let** $\sigma.\Sigma(A) \leftarrow \varnothing$
  let $\sigma' \leftarrow \sigma.\Sigma(A)$
  let $r \leftarrow \$_{\mathsf{std}}(q, \sigma', \sigma.\mathsf{spare})$
  if $r = \bot$ **then**
    $\sigma.\mathsf{spare} \leftarrow 0$
    return $(\bot, y)$
  let $(\sigma', g', y) \leftarrow r$;
  if $\sigma' = \bot$ **then**
    $\sigma.\mathsf{spare} \leftarrow 0$
    return $(\bot, y)$
  else
    $\sigma.\Sigma(A) \leftarrow \sigma'; \sigma.\mathsf{spare} \leftarrow g'$
    return (σ, y)

function $\mathsf{muxPrivOracle}(A, q, \rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$
  let $(\rho.\mathrm{P}, \sigma^o.\Sigma.\rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}) \leftarrow \mathsf{forceInitMaps}((\rho.\mathrm{P}, \sigma^o.\Sigma, \rho^o.\mathrm{P}, \sigma^\pi.\Sigma, \rho^\pi.\mathrm{P}), A, \varnothing)$

let $\rho' \leftarrow \rho.\mathrm{P}(A)$
let $z' \leftarrow (\sigma^o.\sigma_i, \rho^o.\rho_i, \sigma^\pi.\sigma_i, \rho^\pi.\rho_i, \eta)$
let $(\rho', y) \leftarrow q(\rho', z')$
if $\rho' = \bot$ **then return** $(\bot, y)$
else
let $\rho.\mathrm{P}(A) \leftarrow \rho'$
return (ρ, y)

function muxEst$(q, \sigma, \varnothing)$
let $(\sigma', \mathcal{T}, g) \leftarrow \sigma$
let $(\sigma', g', y) \leftarrow \mathrm{E}_{\mathrm{std}}(q, \sigma', \varnothing)$
if $\sigma' = \bot$ **then return** $(\bot, y)$
else
return $((\sigma', \mathcal{T} \parallel [(q, y)], g + g'), y)$

$\mathrm{desc}_{\mathrm{scs}} = \mathrm{desc}_{\mathrm{paymux}}$

$\mathrm{dep}_{\mathrm{scs}} = \mathrm{dep}_{\mathrm{paymux}}$

6.7.6 Exporting Ledger Data

Real-world smart contract systems often have some means to extract limited information about the underlying consensus protocol, such as the hash of the most recent block, the address of the block's miner, or the length of the current chain. These can be useful in applications – in particular the latter, as it provides an imprecise clock for use in contracts.

Clearly, these rely on tighter integration with the underlying consensus mechanism than Kachina provides. We can still capture the core idea, by having a sub-contract which manages such chain data and allows this to be read and set arbitrarily[6]. We can then assume that the correct usage of this sub-contract is enforced by the validation of the underlying consensus mechanism – transactions which attempt to "incorrectly" set the chain data – for any definition of "correct" will never reach the ledger.

Transition Function $\Gamma_{\mathrm{chaindata}}$

The chain data transition function $\Gamma_{\mathrm{chaindata}}$ allows arbitrary setting and reading of state. An external assumption is that the setting of state is both enforced and re-

[6]This could be expanded to allow only certain types of setting – such as advancing the time, but not rewinding it.

stricted by the underlying ledger protocol, to give it meaning – for instance each block may induce a phantom "chain-data" transaction which appears on the ledger and sets the most recent block hash in the chain-data contract's state.

When receiving an input $(\textsc{set}, \sigma')$*:*

run $\mathcal{O}_\sigma(\lambda(\cdot,\cdot)\colon(\sigma', \top))$

When receiving an input GET*:*

return $\mathcal{O}_\sigma(\lambda(\sigma,\cdot)\colon(\sigma,\sigma))$

The contract we present here does have a further issue: Since the loopback in our multiplexers occurs only in the main transition function, the transcripts it generates will commit to specific values for the ledger data upon transaction creation – something which is likely not reasonable. A more complex loopback design, which we do not present here, would solve this: If calling into public or private parts of other contracts were permitted from within the public and private state oracles, respectively.

For both leakage descriptors and dependencies, we make use of our assumption that users cannot directly call SET.

function $\mathrm{desc}_{\mathsf{chaindata}}(t, X, \mathcal{T}_\sigma, \mathcal{T}_\rho, w, z)$

return "Reading the chain data"

function $\mathrm{dep}_{\mathsf{chaindata}}(X, \mathcal{T}_\rho, (\sigma^o, \rho^o, \sigma^\pi, \rho^\pi, \eta))$

return ε

7 Conclusion

This thesis has attempted to solidify the basis of privacy in decentralised blockchain systems, both at the protocol level with privacy preserving proof of stake and smart contracts, and answering questions about the underlying primitive of zk-SNARKs, questions of how they may be securely instantiated and composed. The major results of this thesis are as follows:

- Chapter 3, *Composition with Knowledge Assumptions*, demonstrated how zk-SNARKs can be used in wider composable security proofs, despite relying on knowledge assumptions for extraction. This directly confirms the correctness of constructions which use zk-SNARKs as building blocks in larger protocols, including other work in this thesis.

- Chapter 4, *Secure Reference Strings from Consensus*, addresses the question of trusted setups for zk-SNARKs, which are the weakest link in their security, as they rely on an honest party erasing a secret exponent. To solve this issue, *updateable* reference strings are used, which allow any user to perform an update. This task is given to miners of a Nakamoto-style ledger and, by the honesty assumptions governing these, an honest update is guaranteed.

- Chapter 5, *Privacy in Proof-of-Stake*, combines results in provably-secure proof-of-stake protocols with results from privacy-preserving transaction systems, constructing the first proof-of-stake system which operates over a privacy-preserving distribution of stake. It replaces some of the key cryptography from non-privacy-preserving proof-of-stake with non-interactive zero-knowledge, and relies on existing zk-SNARK constructions for privacy-preserving payments. The result is adaptively secure, ensuring that stake cannot be misused after-the-fact once it has been spent.

- Chapter 6, *Privacy in Smart Contracts*, provides a platform for smart contract authors to use arbitrary computation in zero-knowledge proofs, sepa-

rating computation into three parts: the untrusted private, the trusted private, and the trusted public, where only the latter operates on shared state. This foundation permits smart contracts to be more expressive with respect to the privacy they achieve and enables additional trust assumptions and models to be layered on top of it. Smart contract systems are constructed modularly, including privacy-preserving currency and adjustable fees.

These results can be combined in a single distributed ledger – a proof-of-stake system running privacy-preserving smart contracts, drawing its stake distribution from a Zerocash-like private currency contract. Fully combining these results is not quite immediate, as the setup would need to be with an initially *public* proof-of-stake and the models used for privacy-preserving proof-of-stake and smart contracts differ sufficiently that using a smart contract for the stake of the former is non-trivial. These issues appear minor however – for the former the public Ouroboros Genesis can be run during the setup-phase and switched to Crypsinous once the reference string has finalised. For the latter, it is clear that a smart contract semantically identical to the transfer system of Crypsinous can be written in Kachina – it is simply the matter of correctly expressing the leakage Crypsinous makes in the Kachina model which is non-trivial.

In ending this thesis, I would like to pause to note how grand the problem of privacy in decentralised systems is. Any solution to it must be efficient – however, even the non-private solutions in use today are not efficient enough to be sustainable. Furthermore, the setting is more strict than usual in cryptography: No-one knows who *is participating*, let alone who is trustworthy. It has become increasingly apparent to me during my study that it is unlikely that there is a perfect solution for privacy, and that it is more important to broadcast that privacy is hard and that we should not expect it to come for free, or without a fight. Too often people – developers, users, and even cryptographers alike – assume privacy as automatic. It is not. In the course of this thesis, I tried to fight back a little more of it.

Bibliography

[ABL+19] Behzad Abdolmaleki, Karim Baghery, Helger Lipmaa, Janno Siim, and Michal Zajac. UC-secure CRS generation for SNARKs. In Johannes Buchmann, Abderrahmane Nitaj, and Tajje eddine Rachidi, editors, *AFRICACRYPT 19*, volume 11627 of *LNCS*, pages 99–117. Springer, Heidelberg, July 2019. doi:10.1007/978-3-030-23696-0_6.

[BAZB19] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. Cryptology ePrint Archive, Report 2019/191, 2019. https://eprint.iacr.org/2019/191.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96884-1_25.

[BBDP01] Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 566–582. Springer, Heidelberg, December 2001. doi:10.1007/3-540-45682-1_33.

[BCCT12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In Shafi Goldwasser, editor, *ITCS 2012*, pages 326–349. ACM, January 2012. doi:10.1145/2090236.2090263.

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. doi:10.1109/SP.2014.36.

[BCG+20] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy*, 2020. doi:10.1109/SP40000.2020.00050.

[BCH+20] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 1–30. Springer, Heidelberg, November 2020. doi:10.1007/978-3-030-64381-2_1.

[BGG19] Sean Bowe, Ariel Gabizon, and Matthew D. Green. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, *FC 2018 Workshops*, volume 10958 of *LNCS*, pages 64–77. Springer, Heidelberg, March 2019. doi:10.1007/978-3-662-58820-8_5.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 913–930. ACM Press, October 2018. doi:10.1145/3243734.3243848.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050.

[BHKL13] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 967–980. ACM Press, November 2013. doi:10.1145/2508859.2516734.

[BKP14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in Bitcoin P2P network. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 15–29. ACM Press, November 2014. doi:10.1145/2660267.2660379.

[BKSV21] Karim Baghery, Markulf Kohlweiss, Janno Siim, and Mikhail Volkhov. Another look at extraction and randomization of Groth's zk-SNARK. In *FC 2021*, March 2021.

[BLL00] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In Dimitris Gritzalis, Sushil Jajodia, and Pierangela Samarati, editors, *ACM CCS 2000*, pages 9–17. ACM Press, November 2000. doi:10.1145/352600.352604.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In

Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, August 2017. doi:10.1007/978-3-319-63688-7_11.

[BMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 991–1008. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bulck.

[BP15] Elette Boyle and Rafael Pass. Limits of extractability assumptions with distributional auxiliary input. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 236–261. Springer, Heidelberg, November / December 2015. doi:10.1007/978-3-662-48800-3_10.

[BPS16] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. https://eprint.iacr.org/2016/919.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. doi:10.1145/168588.168596.

[BW06] Xavier Boyen and Brent Waters. Anonymous hierarchical identity-based encryption (without random oracles). In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 290–307. Springer, Heidelberg, August 2006. doi:10.1007/11818175_17.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.

[CD08] Ran Canetti and Ronny Ramzi Dakdouk. Extractable perfectly one-way functions. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *ICALP 2008, Part II*, volume 5126 of *LNCS*, pages 449–460. Springer, Heidelberg, July 2008. doi:10.1007/978-3-540-70583-3_37.

[CD09] Ran Canetti and Ronny Ramzi Dakdouk. Towards a theory of extractable functions. In Omer Reingold, editor, *TCC 2009*, volume

5444 of *LNCS*, pages 595–613. Springer, Heidelberg, March 2009. doi:10.1007/978-3-642-00457-5_35.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 280–312. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78381-9_11.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007. doi:10.1007/978-3-540-70936-7_4.

[CDT19] Jan Camenisch, Manu Drijvers, and Björn Tackmann. Multi-protocol UC and its use for building modular and efficient protocols. Cryptology ePrint Archive, Report 2019/065, 2019. https://eprint.iacr.org/2019/065.

[CEK+16] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 807–840. Springer, Heidelberg, December 2016. doi:10.1007/978-3-662-53890-6_27.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, August 2001. doi:10.1007/3-540-44647-8_2.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In *30th ACM STOC*, pages 209–218. ACM Press, May 1998. doi:10.1145/276698.276741.

[CGH18] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million Facebook profiles harvested for Cambridge Analytica in major data breach. *The Guardian*, March 2018. Date accessed: December 2020. URL: https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election.

[CGJ+17] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 719–728. ACM Press, October / November 2017. doi:10.1145/3133956.3134092.

[CHK03] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 255–271. Springer, Heidelberg, May 2003. doi:10.1007/3-540-39200-9_16.

[CHM+19] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Report 2019/1047, 2019. https://eprint.iacr.org/2019/1047.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020. doi:10.1007/978-3-030-45721-1_26.

[CJS14] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 597–608. ACM Press, November 2014. doi:10.1145/2660267.2660374.

[CKM+19] Manuel Chakravarty, Roman Kireev, Kenneth MacKenzie, Vanessa McHale, Jann Müller, Alexander Nemish, Chad Nester, Michael Peyton Jones, Simon Thompsona, Rebecca Valentine, and Philip Wadler. Functional blockchain contracts, 2019. URL: https://iohk.io/research/papers/#functional-blockchain-contracts.

[CL99] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In Margo I. Seltzer and Paul J. Leach, editors, *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

[CL05] Jan Camenisch and Anna Lysyanskaya. A formal treatment of onion routing. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 169–187. Springer, Heidelberg, August 2005. doi:10.1007/11535218_11.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002. doi:10.1145/509907.509980.

[CPNX20] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020. doi:10.1145/3391195.

[Dam92] Ivan Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 445–456. Springer, Heidelberg, August 1992. doi:10.1007/3-540-46766-1_36.

[DEFM19] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. Perun: Virtual payment hubs over cryptocurrencies. In *2019 IEEE Symposium on Security and Privacy*, pages 106–123. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00020.

[DFH18] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 949–966. ACM Press, October 2018. doi:10.1145/3243734.3243856.

[DG03] Ivan Damgård and Jens Groth. Non-interactive and reusable non-malleable commitment schemes. In *35th ACM STOC*, pages 426–437. ACM Press, June 2003. doi:10.1145/780542.780605.

[DGHM13] Gregory Demay, Peter Gaži, Martin Hirt, and Ueli Maurer. Resource-restricted indifferentiability. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 664–683. Springer, Heidelberg, May 2013. doi:10.1007/978-3-642-38348-9_39.

[DGKR18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 66–98. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78375-8_3.

[DGN03] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 426–444. Springer, Heidelberg, August 2003. doi:10.1007/978-3-540-45146-4_25.

[Dig20] Digiconomist. Bitcoin energy consumption index. https://digiconomist.net/bitcoin-energy-consumption/, 2020. Date accessed: October 2020.

[DPS19] Phil Daian, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 23–41. Springer, Heidelberg, February 2019. doi:10.1007/978-3-030-32101-7_2.

[EGSvR16] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-NG: A scalable blockchain protocol. In Katerina J. Argyraki and Rebecca Isaacs, editors, *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pages 45–59. USENIX Association, 2016. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eyal.

[EPT19] The Enigma Project Team. What is Enigma? https://enigma.co/discovery-documentation/, 2019.

[ET18] Jacob Eberhardt and Stefan Tai. ZoKrates - scalable privacy-preserving off-chain computations. In *IEEE International Conference on Internet of Things*, pages 1084–1091. IEEE, 2018. doi:10.1109/Cybermatics_2018.2018.00199.

[Eth19] Etherscan. Ethereum sync (default) chart, 2019. URL: https://etherscan.io/chartsync/chaindefault.

[FGKR20] Matthias Fitzi, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Proof-of-stake blockchain protocols with near-optimal throughput. *IACR Cryptol. ePrint Arch.*, 2020:37, 2020. URL: https://eprint.iacr.org/2020/037.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 152–168. Springer, Heidelberg, August 2005. doi:10.1007/11535218_10.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96881-0_2.

[Fol17] David Folkenflik. Facebook scrutinized over its role in 2016's presidential election. *NPR*, September 2017. Date accessed: December 2020. URL: https://www.npr.org/2017/09/26/553661942/facebook-scrutinized-over-its-role-in-2016s-presidential-election.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987. doi:10.1007/3-540-47721-7_12.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without

PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013. doi:10.1007/978-3-642-38348-9_37.

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. Cryptology ePrint Archive, Report 2017/454, 2017. http://eprint.iacr.org/2017/454.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015. doi:10.1007/978-3-662-46803-6_10.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The Bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 291–323. Springer, Heidelberg, August 2017. doi:10.1007/978-3-319-63688-7_10.

[GKM+18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96878-0_24.

[GKR18] Peter Gaži, Aggelos Kiayias, and Alexander Russell. Stake-bleeding attacks on proof-of-stake blockchains. Cryptology ePrint Archive, Report 2018/248, 2018. https://eprint.iacr.org/2018/248.

[GKZ19] Peter Gazi, Aggelos Kiayias, and Dionysis Zindros. Proof-of-stake sidechains. In *2019 IEEE Symposium on Security and Privacy*, pages 139–156. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00040.

[GM17] Jens Groth and Mary Maller. Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 581–612. Springer, Heidelberg, August 2017. doi:10.1007/978-3-319-63715-0_20.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987. doi:10.1145/28395.28420.

[GO94] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, December 1994. doi:10.1007/BF00195207.

[GOT18] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. Proof-of-stake protocols for privacy-aware blockchains. Cryptology ePrint Archive, Report 2018/1105, 2018. URL: https://eprint.iacr.org/2018/1105.

[Gro10] Jens Groth. Short pairing-based non-interactive zero-knowledge arguments. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 321–340. Springer, Heidelberg, December 2010. doi:10.1007/978-3-642-17373-8_19.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016. doi:10.1007/978-3-662-49896-5_11.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[HBHW18] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. 2018. URL: https://github.com/zcash/zips/blob/master/protocol/protocol.pdf.

[HT98] Satoshi Hada and Toshiaki Tanaka. On the existence of 3-round zero-knowledge protocols. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 408–423. Springer, Heidelberg, August 1998. doi:10.1007/BFb0055744.

[JJR02] Markus Jakobsson, Ari Juels, and Ronald L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In Dan Boneh, editor, *USENIX Security 2002*, pages 339–353. USENIX Association, August 2002.

[Jut12] Jutarul. ppcoin – stake burn-through vulnerability. https://bitcointalk.org/index.php?topic=131901.0, December 2012. Date accessed: December 2020.

[KCE+15] Harry A. Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of Namecoin and lessons for decentralized namespace design. In *14th Annual Workshop on the Economics of Information Security, WEIS 2015, Delft, The Netherlands, 22-23 June, 2015*, 2015. URL: http://www.econinfosec.org/archive/weis2015/papers/WEIS_2015_kalodner.pdf.

[Ker20] Thomas Kerber. Implementations to accompany "mining for privacy". GitHub, 2020. URL: https://github.com/tkerber/pistis-impl.

[KGC+18] Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1353–1370. USENIX Association, August 2018.

[KKK21a] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Composition with knowledge assumptions. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 364–393, Virtual Event, August 2021. Springer, Heidelberg. doi:10.1007/978-3-030-84259-8_13.

[KKK21b] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Kachina – foundations of private smart contracts. In *IEEE CSF 2021*, June 2021. URL: https://doi.org/10.1109/CSF51468.2021.00002.

[KKK21c] Thomas Kerber, Aggelos Kiayias, and Markulf Kohlweiss. Mining for privacy: How to bootstrap a snarky blockchain. In *FC 2021*, March 2021.

[KKKZ19] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. Ouroboros Crypsinous: Privacy-preserving proof-of-stake. In *2019 IEEE Symposium on Security and Privacy*, pages 157–174. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00063.

[KL20] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. A composable security treatment of the lightning network. In Limin Jia and Ralf Küsters, editors, *CSF 2020 Computer Security Foundations Symposium*, pages 334–349. IEEE Computer Society Press, 2020. doi:10.1109/CSF49147.2020.00031.

[KLT16] Aggelos Kiayias, Feng-Hao Liu, and Yiannis Tselekounis. Practical non-malleable codes from l-more extractable hash functions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1317–1328. ACM Press, October 2016. doi:10.1145/2976749.2978352.

[KM20] Cecilia Kang and David McCabe. House lawmakers condemn big tech's 'monopoly power' and urge their breakups. *The New York Times*, October 2020. Date accessed: December 2020. URL: https://www.nytimes.com/2020/10/06/technology/congress-big-tech-monopoly-power.html.

[KMS+16] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016. doi:10.1109/SP.2016.55.

[KN12] Sunny King and Scott Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake. 2012. URL: https://decred.org/research/king2012.pdf.

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, Heidelberg, August 2017. doi:10.1007/978-3-319-63688-7_12.

[KYMM18] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An empirical analysis of anonymity in Zcash. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 463–477. USENIX Association, August 2018.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010. doi:10.1007/978-3-642-17373-8_11.

[KZM+15] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. CøCø: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. URL: https://eprint.iacr.org/2015/1093.

[LH20] Kif Leswing and Todd Haselton. Epic Games and many other app makers are not impressed with Apple's olive branch. *CNBC*, November 2020. Date accessed: December 2020. URL: https://www.cnbc.com/2020/11/18/epic-games-spotify-not-impressed-with-apples-app-store-changes.html.

[Lip12] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 169–189. Springer, Heidelberg, March 2012. doi:10.1007/978-3-642-28914-9_10.

[LM20] David Lanzenberger and Ueli Maurer. Coupling of random systems. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 207–240. Springer, Heidelberg, November 2020. doi:10.1007/978-3-030-64381-2_8.

[LNS20] Jonathan Lee, Kirill Nikitin, and Srinath T. V. Setty. Replicated state machines without replicated execution. In *2020 IEEE Symposium on Security and Privacy*, pages 119–134. IEEE Computer Society Press, May 2020. doi:10.1109/SP40000.2020.00068.

[Mau02] Ueli M. Maurer. Indistinguishability of random systems. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 110–132. Springer, Heidelberg, April / May 2002. doi:10.1007/3-540-46035-7_8.

[Mau11] Ueli Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *TOSCA 2011*, volume 6993 of *LNCS*, pages 33–56. Springer, 2011. doi:10.1007/978-3-642-27375-9_3.

[Max13] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. https://bitcointalk.org/?topic=279249, August 2013.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, November 2019. doi:10.1145/3319535.3339817.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013. doi:10.1109/SP.2013.34.

[MOG+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy*, pages 1466–1482. IEEE Computer Society Press, May 2020. doi:10.1109/SP40000.2020.00057.

[MPJ+13] 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. *login Usenix Mag.*, 38(6), 2013. URL: https://www.usenix.org/publications/login/december-2013-volume-38-number-6/fistful-bitcoins-characterizing-payments-among.

[MR11] Ueli Maurer and Renato Renner. Abstract cryptography. In Bernard Chazelle, editor, *ICS 2011*, pages 1–21. Tsinghua University Press, January 2011.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126. Springer, Heidelberg, August 2002. doi:10.1007/3-540-45708-9_8.

[Nov21] Jordan Novet. Parler's de-platforming shows the exceptional power of cloud providers like Amazon. *CNBC*, January 2021. Date accessed: January 2021. URL: https://www.cnbc.com/2021/01/16/how-parler-deplatforming-shows-power-of-cloud-providers.html.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013. doi:10.1109/SP.2013.47.

[Pri17] Matthew Prince. Why we terminated Daily Stormer. *Cloudflare Blog*, August 2017. Date accessed: December 2020. URL: https://blog.cloudflare.com/why-we-terminated-daily-stormer/.

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 643–673. Springer, Heidelberg, April / May 2017. doi:10.1007/978-3-319-56614-6_22.

[PW20] Mario Parker and Josh Wingrove. Trump says he's being unfairly censored by Facebook, Twitter. *Bloomberg*, August 2020. Date accessed: December 2020. URL: https://www.bloomberg.com/news/articles/2020-08-06/trump-says-he-s-being-unfairly-censored-by-facebook-twitter.

[RMK14] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. CoinShuffle: Practical decentralized coin mixing for Bitcoin. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II*, pages 345–364, 2014. doi:10.1007/978-3-319-11212-1_20.

[SBG+19] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin T. Vechev. zkay: Specifying and enforcing data privacy in smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1759–1776. ACM Press, November 2019. doi:10.1145/3319535.3363222.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990. doi:10.1007/0-387-34805-0_22.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997. doi:10.1007/3-540-69053-0_18.

[SWB19] Josh Swihart, Benjamin Winston, and Sean Bowe. Zcash counterfeiting vulnerability successfully remediated. ECC Blog, February 2019. https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/.

[Sza97] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. URL: https://firstmonday.org/ojs/index.php/fm/article/view/548.

[VB15] Fabian Vogelsteller and Vitalik Buterin. ERC-20 token standard, 2015. URL: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[VMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 991–1008. USENIX Association, August 2018.

[vS13] Nicolas van Saberhagen. Cryptonote v 2.0. https://cryptonote.org/whitepaper.pdf, October 2013.

[Vuk15] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International workshop on open problems in network security*, pages 112–125. Springer, 2015. doi:10.1007/978-3-319-39028-4_9.

[War12] Jonathan Warren. Bitmessage: A peer-to-peer message authentication and delivery system, November 2012. URL: https://bitmessage.org/bitmessage.pdf.

[Wil16] Jeffrey Wilcke. The Ethereum network is currently undergoing a DoS attack. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/, September 2016.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[WP21] Julia Carrie Wong and Karl Paul. Twitter permanently suspends Trump's account to prevent 'further incitement of violence'. *The Guardian*, January 2021. Date accessed: January 2021. URL: https://www.theguardian.com/us-news/2021/jan/08/donald-trump-twitter-ban-suspended.

[WSNH19] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. SoK: Sharding on blockchain. Cryptology ePrint Archive, Report 2019/1178, 2019. https://eprint.iacr.org/2019/1178.

[Zah18] Joachim Zahnentferner. Chimeric ledgers: Translating and unifying UTXO-based and account-based cryptocurrencies. Cryptology ePrint Archive, Report 2018/262, 2018. https://eprint.iacr.org/2018/262.

[Zca18] Zcash. Parameter generation. https://z.cash/technology/paramgen/, 2018.

[Zca19] Zcash. Address and value pools in Zcash. https://zcash.readthedocs.io/en/latest/rtd_pages/addresses.html#turnstiles, 2019.

[ZNP15] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized Computation Platform with Guaranteed Privacy. *arXiv e-prints*, June 2015. arXiv:1506.03471.