Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake

Thomas Kerber*, Markulf Kohlweiss†, Aggelos Kiayias‡, and Vassilis Zikas§

The University of Edinburgh and IOHK

May 15, 2019

Abstract

We present Ouroboros Crypsinous, the first formally analysed privacy-preserving proof-of-stake (PoS) blockchain protocol. To model its security we give a thorough treatment of private ledgers in the universal composition (UC) setting that might be of independent interest. To prove our protocol secure against adaptive attacks, which are particularly critical in the PoS setting, we introduce a new coin evolution technique that relies on SNARKs and key-private forward secure encryption. The latter primitive–and the associated construction–can be of independent interest. We stress that existing approaches to private blockchains, such as the proof-of-work-based Zerocash are analyzed only against static corruptions.

1 Introduction

A significant limitation of traditional blockchain protocols, such as the Bitcoin, is the fact that the transaction ledger is a public resource and thus information about the way the transaction issuers operate may be leaked to an adversary. This consideration was acknowledged early on and Bitcoin itself [27] includes a number of measures to mitigate transaction privacy loss. Namely users produce a new pseudonymous address for each payment received and addresses from the same wallet can be selected to be indistinguishable from addresses from different wallets. Still, the information available in the blockchain itself is susceptible to analysis and it has been demonstrated early on that significant information can be extracted by clustering the Bitcoin transaction "graph", see e.g., [30, 24].

This state of affairs motivated the development of privacy enhancing and privacy-preserving techniques for distributed ledgers. First, methods such as CoinJoin [23] and CoinShuffle [31] were proposed as mechanisms to reduce the effectiveness of de-anonymization techniques based on tracing and clustering. Subsequently redesigned cryptocurrencies were put forth that attempted to introduce stronger privacy-enhancing techniques by design in the distributed ledger protocol. These included Zerocash [4] and Monero which is based on Cryptonote [32]. We note that despite their enhanced privacy characteristics, some leakage still exists in these protocols (even if we exclude leakage on the network layer, which is an issue orthogonal to what these protocols study including the present work). This can be exploited as demonstrated in recent works [22, 26, 19]. The

*Email: t.kerber@ed.ac.uk

†Email: mkohlwei@ed.ac.uk

‡Email: akiayias@ed.ac.uk; Work partly supported by H2020 project #780477, PRIViLEDGE.

§Email: vzikas@ed.ac.uk

above privacy-enhancing techniques primarily focused on the transaction processing layer of the distributed ledger leaving the consensus back-end mechanism largely the same.

Concurrently with these developments however, another line of research works in blockchain design focused on resolving fundamental issues with the energy consumption requirements of the underlying proof-of-work (PoW) mechanism of Bitcoin. In particular, this lead to a sequence of works in proof-of-stake (PoS) blockchain protocols that include Algorand [25], Ouroboros [20], Ouroboros Praos [13], Ouroboros Genesis [1], Sleepy-Consensus [29], and Snow White [5]. PoS blockchain protocols alleviate the requirement to perform proof-of-work by solving computationally hard puzzles. Instead, they refer to the stake that each participant possesses as reported in the blockchain and, through cryptographic means, elect the next participant to extend the transaction ledger (who is commonly referred as the next *leader* or even *slot leader* when the execution time is divided in slots.) PoS protocols have been touted as the next important advance in real-world distributed ledger systems and a number of well-known cryptocurrencies are in the process of incorporating them into their deployed systems including Ethereum with the Casper protocol [33] and Cardano with Ouroboros [11].

The above state of affairs raises an important open question: is it possible to build a PoS-based privacy enhanced distributed ledger? This is the main motivation of this work where we tackle this problem and answer the question in the affirmative.

Our results. We propose a new formal model for a PoS-based privacy-preserving distributed ledger in the universal composition (UC) setting, [8], and a new protocol that realizes it, Ouroboros Crypsinous.[1] Our protocol analysis with respect to the basic properties of consistency and liveness is inspired by Ouroboros Genesis, [1], a recent (non-private) PoS blockchain protocol formally analyzed in the UC setting. Our protocol provides the first formally analysed PoS-based privacy-enhanced blockchain protocol. Moreover, for the first time our protocol achieves simulation-based privacy that is even universally composable, as well as forward-secure, i.e., it ensures that privacy (as well as consistency and liveness) are preserved independently of any other protocols running concurrently with our ledger implementation, and even under adaptive corruption.

It is worth noting that PoS 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 designing a new privacy-enhancing PoS operation that, roughly speaking, extends the SNARK machinery of "transaction pouring" in Zerocash to a setting where coins evolve without loosing their value, enabling on the way a proof of stake-eligibility that does not leak any additional information.

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 e.g., re-issuing past PoS blocks. In non-private PoS protocols such as Algorand [25] and Ouroboros Genesis [1] 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 realize–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 naïve 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.

[1]The word "Crypsinous" is Greek and refers to a person who is mindful of their privacy. We thank Konstantinos Mitropoulos for suggesting it to us.

Our private ledger formalization is also of independent interest since it captures for the first time the concept of a privacy enhanced transaction ledger in the UC-setting which is generally applicable to both the PoW and PoS settings. Interestingly, we observe that the latter case requires a slightly expanded adversarial interface that allows a sampling of the stakeholder distribution per unit of time (referred to as "slot"). (A similar sampling can be also observed in Bitcoin, but since *miner privacy* is not considered a prime requirement this was never formalized.) Adversarial sampling captures the fact that in the PoS setting traffic analysis is possible based merely on the frequency one entity issues a PoS block. Our formal model ensures that this is the only privacy leakage that will be incurred during the execution of the protocol. A secondary formalization contribution is the concept of UC key-private forward-secure encryption which, even though the two relevant properties were studied independently, a UC functionality capturing both has never appeared until our work.

We note that our work is concurrent, and independent, of another paper on privacy-preserving proof-of-stake by Ganesh et al. [15]. This work focuses on constructing a generic, privacy-preserving leadership election, given a list of commitments to each party's stake. Our work 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 [15], 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.

2 Protocol Intuition

To begin with, we give a high-level sketch of the Ouroboros Crypsinous protocol in this Section, to aid in understanding the more formal break-down of the protocol in Section 6, and to introduce core concepts. We will first sketch the design of two protocols we are building on – Ouroboros Genesis [1], and Zerocash [4]. 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.

2.1 The Foundations of Genesis and Zerocash

Ouroboros Genesis [1], divides time into discrete *slots*. At protocol start, parties are assigned initial *stake* in the system. Typically, only the relative amount of such stake is considered, i.e. how much each party holds out of the total stake. By protocol-external means, the distribution of this stake may shift over time, e.g. by users trading it amongst each other. Each slot, users have a probability proportional[2] 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 epoch, 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 [4] 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

[2]We note that although it is not technically linear, this is a close approximation.

closely mirrors the simplest form of Bitcoin transactions. Each party holds a secret key used to spend coins. This secret key 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 coins 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 coins *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.

2.2 The Core Protocol

The core principle of Ouroboros Crypsinous is combining the strengths of both the Ouroboros Genesis and Zerocash protocols. In particular, we note that 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 coins value, however, in Crypsinous parties produce a NIZK proof of this, as well as proving that the respective coin is unspent. 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.

2.3 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 Ouroboros 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 as sets of 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 coins 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 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.

2.4 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. Further, we note that – in the UC

framework – a non-committing encryption 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 acts 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 creating time, by committing to this in the corresponding public key. While this does mean keys can expire, we note parties can trivially refresh them, and further will sketch in Section 8 that this is a rare occurrence for practical parameters. We emphasize that parties *are* able to spend and refresh keys, even when expired.

While we could easily present Ouroboros Crypsinous using non-committing encryption, known realizations 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, $\Delta_{\max}$. This allows us to utilize forward secure encryption instead of non-committing encryption, under the assumption that corruption is "delayed" by $\Delta_{\max}$. This delay is modeled by restricting adversarial access to the forward secure encryption secret key at time τ to the key for time $\tau + \Delta_{\max}$.

3 The Model

Following the recent line of works proving composable security of blockchain ledgers [2, 1] we provide our protocol and security proof in Canetti's universal composition (UC) framework [8]. 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. We assume that the reader is familiar with simulation-based security and has basic knowledge of the UC framework. We provide all the aspects of the execution model from [2, 1] 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. We note that for obtaining a better abstraction of reality, some of our hybrids are described as global (GUC) setups [9]. The main difference of such setups from standard UC functionalities is that the former are accessible by arbitrary protocols and, therefore, allow the protocols to share their (the setups') state. The low-level details of the GUC framework–and the extra points which differentiate it from UC–are not necessary for understanding our protocols and proofs; we refer the interested reader to [9] for these details. We will use sid as a session identifier throughout the paper.

Protocol participants are represented as parties–formally Interactive Turing Machine instances (ITIs)—in a multi-party computation. We assume a central adversary $\mathcal{A}$ who corrupts stakeholders and uses them to attack the protocol. The adversary is *adaptive*, i.e., can corrupt additional stakeholders at any point and depending on his current view of the protocol execution. We cast our protocols in the partially synchronous communication version of UC proposed in [2]: parties have access to a global clock setup, denoted by $\mathcal{G}_{\text{CLOCK}}$, and can communicate over a network of authenticated multicast channels with a bounded delay Δ denoted by $\mathcal{F}_{\text{N-MC}}^{\Delta}$. Every honest party can send a message thought $\mathcal{F}_{\text{N-MC}}^{\Delta}$ to all other honest parties but the adversary can delay its delivery to any honest party by a number of rounds of his choice but no greater than Δ. Honest

receivers cannot tell when a message will arrive as they know neither when the message was sent nor the delay Δ. As in the case of Bitcoin, cf. [16, 28, 1], our protocol is implicitly aware of an overestimate $\Delta_{\max}$ of the actual (unknown) network delay Δ. However, this $\Delta_{\max}$ 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 $\Delta_{\max}$ is only used in the staking procedure to determine the leader(s) of each slot.

Similarly to [2, 1], for UC realization in such a globally synchronized setting, the target ideal functionality, i.e., the ledger, needs to keep track of the number of activations that an honest party gets–so that it can enforce in the ideal world the same pace of the clock as in the real world. This is achieved by describing the protocol so that it has an (implicit) predictable behavior of clock interactions for any given activation pattern–which the ideal functionality can (and will) mimic. We refer to [2] for details.

We adopt the *dynamic availability* model implicit in [2] which was fleshed out in [1]. We next sketch its main components: All functionalities, protocols, and global setups have a dynamic party set, i.e., they all include special instructions allowing parties to register, deregister, and allowing the adversary to learn the current set of registered parties. Additionally, global setups allow any other setup (or functionality) to register and deregister with them, and they also allow other setups to learn their set of registered parties.

Utilizing the full dynamic availability model results in separating the honest parties in the following categories: *offline* parties are honest parties that are deregistered from the network functionality. Parties which are not offline are separated into two (sub-)categories, called *(fully) online*–parties which are registered with all their setups and ideal resources–and *(online but) stalled*–parties that are registered with their local network functionality, but are unregistered with at least one of the global setups. Each of these (non-offline) subclasses is further split into two subcategories along the lines of the classification of [2]: those that have been in a non-offline state for more that `Delay` rounds–where `Delay` is a ledger parameter–are *synchronized*, whereas the remainder are *de-synchronized*. Our protocol makes use of the following hybrid functionalities from [1]. (The ideal world execution makes access to the global setups presented below and the private ledger functionality which is presented in Section 5.)

- The global clock functionality $\mathcal{G}_{\text{CLOCK}}$ which keeps track of the current (global) round and reports it to any party that requests it. The round advances whenever all honest (currently registered) parties and functionalities inform $\mathcal{G}_{\text{CLOCK}}$ that they are finished with their current round's actions (note that this is not a communication round).
- The bounded-delay authenticated channels network $\mathcal{F}_{\text{N-MC}}^{\Delta}$ described above.
- 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 [1] 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 [1] we assume wlog that the global time is $\tau = 0$ in the genesis round. We refer to Appendix A for a description of our new genesis block functionality.
- A global random oracle $\mathcal{G}_{\text{RO}}$ for abstracting hash function queries. As typically in cryptographic proofs the queries to hash function are modeled by assuming access to a random oracle: Upon receiving a query $(\text{EVAL}, \text{sid}, x)$ from a registered party, if x has not been queried before, a

value y is chosen uniformly at random from $\{0,1\}^\kappa$ (for security parameter κ) and returned to the party (and the mapping (x, y) is internally stored). If x has been queried before, the corresponding y is returned. As in [1] we capture this by a global random oracle (GRO), i.e., a global setup that behaves as above.

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 [13, 1] relies on verifiable random functions and key-evolving signatures to ensure security of the lottery which defines slot leaders and prevent 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. These functionalities along with their implementation from a common reference string (CRS) and their security proofs are described in Section 4. To our knowledge no definition of key-private forward secure encryption or an implementation thereof has been suggested. In fact, for reasons discussed below (see Section 4.2) 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 pseudorandom functions (PRFs). Construction of both these primitives exists assuming a CRS under standard hardness assumption, e.g., hardness of the DDH (Decision Diffie Hellman) problem.

Remark 1: (Assumptions on the environment/adversary as functionality wrappers.) The security statements about implementation of ledgers are typically conditional. E.g., 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 [2] to capture these assumptions without compromizing 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 [2] for a more detailed discussion. As a forward pointer, the wrapper used in our security statements is sketched in Appendix E. As this wrapper only becomes relevant for interpreting our main theorems (Theorem 1 and 2) it might be easier for the first-time reader to postpone parsing it until then.

4 Tools

In this section we describe the main tools used by Ouroboros Crypsinous: non-interactive zero-knowledge proofs (NIZKs), key-private forward secure encryption, maliciously-unpredictable PRFs (MUPRFs), and equivocal commitments. We describe ideal functionalities capturing NIZK and key-private forward-secure encryption, and refer to their UC implementations in the appendix. Ouroboros Crypsinous is described and proved secure assuming hybrid access to these ideal functionalities and its security when these functionalities are replaced by their implementations will follow directly from the universal composition theorem.

Further, we define the properties satisfied by MUPRFs and equivocal commitments.

4.1 Non-Interactive Zero Knowledge

We utilize the Non-Interactive Zero Knowledge functionality $\mathcal{F}_{\text{NIZK}}$ and protocol of [21] , (for completeness, $\mathcal{F}_{\text{NIZK}}$ is described in detail in Appendix A). This functionality allows generating proofs π that a statement $\mathbf{x}$ is in a given NP language $\mathcal{L}$, with a witness $\mathbf{w}$. We use the "weak" functionality suggested, which permits an adversary to generate new proofs for already proven statements.

We note that while [21] provides a construction, it is only shown to satisfy game-based properties. We formally prove its security in the UC setting assuming a CRS in Appendix F.

NIZKs can be used for signature-like behaviour by embedding the messages that are to be signed in the statements of simulation-extractable NIZKs, constructing in this way a *signature of knowledge* [17] (SoK). In particular, we note that witnesses used to generate proofs in Ouroboros 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 Ouroboros Crypsinous has similar unforgeability properties as standard signatures.

4.2 Key-private Forward-Secure Encryption

To guarantee the forward-privacy of transactions, a forward-secure encryption scheme [10] is necessary to hide information sent encrypted to a party's long-term encryption secret key. Note that 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 [3]. 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 introducing a 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 $\Delta_{\max}$ 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 realize 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 τ and sent over to the receiver so that it can be decrypted with key sk_{τ}^{ENC}. Upon receiving it, the receiver can decrypt it (using sk_{τ}^{ENC}), and immediately update the key to $sk_{\tau'}^{\text{ENC}}$ for the next round (and erase sk_{τ}^{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 out setting, as this gives the adversary a window of opportunity of size Δ, and bounded only by $\Delta_{\max}$, 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 sk_{τ}^{ENC} at time τ does not receive sk_{τ}^{ENC}, but rather the key $sk_{\tau+\Delta_{\max}}^{\text{ENC}}$, which this party would hold in time $\tau + \Delta_{\max}$, if it were allowed to properly update its key. We emphasize that this is a milder assumption than that of delayed party-corruption which underlines the security of [20, 5]. 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 [2, 14] (related to the wrappers used in Remark 1.): Concretely, we introduce an ideal functionality which captures this restriction/assumption. This functionality, denoted by $\mathcal{F}_{\text{KEYMEM}}$, 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) $\mathcal{F}_{\text{KEYMEM}}$ first applies Update $\Delta_{\max}$ times, and returns the updated key to the adversary.

Note that the direct way of enforcing the assumption would be to limit all our statements to only apply to a restricted class of adversaries. For reasons similar to the discussion in Remark 1 above, this would immediately imply that universal composition no longer holds.[3] As an added bonus from using the above functionality-based approach for restricting the adversary, our treatment ensures that the restriction is localized to the encryption functionality; thus, if someone comes up with an instantiation of the encryption functionality against a fully adaptive adversary, or protocol would immediately be secure against such an adversary. The $\mathcal{F}_{\text{KEYMEM}}$ functionality is specified below.

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

$\mathcal{F}_{\text{KEYMEM}}$ is parameterized by its corruption delay $\Delta_{\max}$, and a memory update function Update. It maintains a memory M_p for each party U_p, initialized to ϵ, as well as a flag isInit_p for each party U_p, initialized to $\perp$. We write $\mathsf{Update}^{\Delta}_{\max}$ to mean "apply Update $\Delta_{\max}$ times."

On receiving a message (Init, sid, M') *from* U_p*:* If $\mathsf{isInit}_p = \perp$, let $M_p \leftarrow M'$; $\mathsf{isInit}_p \leftarrow \top$.

On receiving a message (Get, sid) *from* U_p*:* If $\mathsf{isInit}_p = \top$, return M_p if U_p is honest, otherwise return $\mathsf{Update}^{\Delta}_{\max}(M_p)$.

On receiving a message (Update, sid) *from* U_p*:* If $\mathsf{isInit}_p = \top$, update $M_p \leftarrow \mathsf{Update}(M_p)$.

The UC functionality for key-private and forward-secure encryption, $\mathcal{F}_{\text{FWENC}}$, is described in detail in Appendix A, and the accompanying construction is described below.

We extend the notion of forward-secure encryption (FSE) with a notion of *key privacy*, described in detail in Definition 1 below. While this definition itself is novel, it is possible to combine existing schemes to satisfy it. In particular, [10] constructs FSE from hierarchical identity-based encryption (HIBE). Their scheme, paired with the anonymous HIBE construction of [7] satisfies our requirements of key-privacy as we will argue below.

For the argument of key privacy, the FSE from HIBE construction in [10] is straightforward, with the ciphertexts simply being the underlying HIBE scheme's ciphertexts. The core argument of the anonymity of [7], is the indistinguishability of ciphertexts from random group elements – and therefore their independence of the encrypting identity (cf. [7, Lemmas 8& 9]. We note that the ciphertexts' pseudo-randomness implies a stronger notion than just anonymity – the ciphertext

[3] One could attempt to prove a tailored, weaker, and non-universal composition theorem that would apply only to the restricted class of adversaries considered in our encryption-scheme's security proof. But this is not in the spirit of our treatment which explicitly aims at fully-composable (UC) protocols, and it is also rendered unnecessary using our trick above.

also does not reveal any information about the HIBE public key. In particular, as ciphertexts are indistinguishable, our enhanced security game given in Definition 1 is satisfied. The game, as well as the subsequent UC construction, can be found in Appendix G.

This construction's time and space complexity is logarithmic to 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.

Key-Private Forward-Security Against Chosen Ciphertext Attacks

Definition 1. *A key-evolving public-key encryption scheme is* key-privately forward-secure against chosen ciphertext attacks (kp-fs-CCA) *if any PPT adversary has only negligible advantage* $|2\cdot\Pr[b' = b] - 1|$ *in the following game:*

Setup: *For each party* $U_p \in \mathcal{P}$*, run* $(pk_p, sk_p^0) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, N)$*. The adversary receives all public keys* pk_p*. Further, a bit* $b \leftarrow \{0,1\}$ *is selected, but not revealed to the adversary.*

Attack: *The adversary issues multiple* $\mathsf{challenge}(j, (U_0, m_0), (U_1, m_1))$ *queries, multiple* $\mathsf{corrupt}(i, U_p)$ *queries, and multiple* $\mathsf{decrypt}(k, c, U_p)$ *queries, where* $U_p, U_0, U_1 \in \mathcal{P}$*, and* $0 \leq i \leq N; 0 \leq j \leq N; k \leq N$*. Further, if a* corrupt *query is made for some party 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.*

- $\mathsf{corrupt}(i, U_p)$ *is answered with* $sk_p^i \triangleq \mathsf{Upd}(\ldots \mathsf{Upd}(sk_p^0, 1), \ldots, i)$*.*
- $\mathsf{challenge}(j, (U_0, m_0), (U_1, m_1))$ *is answered by responding with* $c = \mathsf{Enc}_{pk_{U_b}}(j, m_b)$*, and* (j, c, U_0) *and* (j, c, U_1) *are recorded as challenges.*
- $\mathsf{decrypt}(k, c, U_p)$ *is answered with* $\perp$ *if* (k, c, U_p) *is recorded as a challenge. Otherwise, it is answered with* $\mathsf{Dec}_{sk_p^k}(k, c^*)$*.*

Guess: *The adversary outputs a guess* $b' \in \{0,1\}$*, and wins the game iff* $b' = b$*.*

4.3 PRFs with unpredictability under malicious keys

Consider a PRF family $\{f_k\}_{k\in K}$ such that $f_k : X \to Y$ 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 [13]. 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 $\Pr[f_k(x) < T | x \notin Q_H]$ 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 $H : \{0,1\}^* \to \langle g\rangle$ be a function mapping to a cyclic group generated by g with a compact representation. We use an elliptic curve group based on the "elligator" curves [6] that have the property that a uniform element over $\langle g\rangle$ is indistinguishable from a random κ-bit string. We then define $f_k(m) \mapsto H(m)^k$ 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 $\langle g^k, H(m), H(m)^k\rangle$ is a DDH triple over the group $\langle g\rangle$. 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 $x \notin Q_H$.

4.4 Equivocal Commitments

We make use of a standard non-interactive equivocal commitment scheme, which is secure in the CRS model assuming hardness of discrete logarithms (cf. [12]). 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, $\widehat{\mathsf{Init}}_{\mathsf{comm}}$, $\widehat{\mathsf{Comm}}$, and Equiv. $\mathsf{Init}_{\mathsf{comm}}$ generates a public key pk^{COMM} which is given as an argument to Comm and DeComm and will be part of the parameterization of the CRS functionality. In addition to satisfying the traditional commitment properties, of binding, hiding, and correctness, the scheme also satisfies equivocality. Specifically, $\widehat{\mathsf{Init}}_{\mathsf{comm}}$ provides an equivocation key in addition to pk^{COMM}. This equivocation key "breaks" the binding property – $\widehat{\mathsf{Comm}}$ 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 Ouroboros Crypsinous' design, in particular the NIZK functionality.

We write $(cm, r) \leftarrow \mathsf{Comm}(m)$ to create the commitment cm for message m, and $\mathsf{DeComm}(cm, m, r) = \top$ if the decommitment to m and r verifies. Likewise, we write $cm \leftarrow \widehat{\mathsf{Comm}}(ek)$ for simulating a commitment with equivocation key ek, and $r \leftarrow \mathsf{Equiv}(ek, cm, m)$ to equivocate, where $\mathsf{DeComm}(cm, m, r) = \top$. In all these, we leave the public key pk^{COMM} implicit, as it is assumed to be globally known via the CRS.

5 The Private Ledger

We next provide the complete description of the private ledger functionality that, as we prove, is implemented by Ouroboros Crypsinous. To describe how privacy is captured in the Crypsinous ledger, we first recall how submitted transactions are stored in the original–non-private—ledger from [2, 1]: When a transaction tx is submitted, the ledger creates–and stores in the buffer–an *annotated version* of the transaction tx, denoted as $\mathsf{BTX} := (\mathsf{tx}, \text{txid}, \tau_L, U_s)$, which includes several useful metadata: txid is a unique identifier for this transaction, τ_L is the clock value when the transaction is received, and U_s is the ID of the party that submitted the transaction. Note that this metadata is used for internal bookkeeping and is not necessarily included in the state of the ledger when (and if) the transaction makes it there. In fact, whether or not this data is included in the state is mandated by the $\mathsf{Blockify}$ function of [2, 1], a parameter to the private ledger. Nonetheless, in the non-private ledger case, the metadata is handed also to the simulator whenever a transaction is given to him.

Privacy of Crypsinous is captured by the following modifications: First, transactions returned from the functionality are *blinded* by a function $\mathsf{BlindTx}$ which is a parameter of a functionality. This function hides any information a party should not see from the ledger state, while state validation operates over the entire, non-blinded state. Further, we parameterize the private ledger with a general purpose leakage algorithm, Lkg, which may additionally leak any function of the ledger state to the adversary. To facilitate an easy comparison with previous work, in the below description, differences from the ledger functionality of [2, 1] are highlighted in blue.

As a technicality, as the $\mathsf{BlindTx}$ function must be applied to "blockified" states, however the structure of these is not known in general, an additional "state blinding" algorithm is ac-

cepted as a parameter, which we require to behave equivalently to first blinding all transactions, then passing them to $\mathsf{Blockify}$. Intuitively, for any given state state, $\mathsf{Blind}(\mathcal{P}, \mathsf{ids}, \mathsf{state})$ returns state with every transaction tx replaced by $\mathsf{BlindTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, \mathsf{tx})$. In particular, where $\beta \triangleq \mathsf{map}(\mathsf{BlindTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}))$, we require that:

$$\mathsf{Blind}\left(\mathcal{P}, \begin{array}{c}\mathsf{Blockify}(\vec{\mathsf{tx}}_1) \parallel \\ \dots \quad \parallel \\ \mathsf{Blockify}(\vec{\mathsf{tx}}_\ell) \parallel\end{array}\right) = \begin{array}{c}\mathsf{Blockify}(\beta(\vec{\mathsf{tx}}_1)) \parallel \\ \dots \quad \parallel \\ \mathsf{Blockify}(\beta(\vec{\mathsf{tx}}_\ell)) \parallel\end{array}$$

$\mathsf{Blind}_{\mathcal{A}}$ is defined the same as Blind, but with calls to $\mathsf{BlindTx}$ replaced with calls to $\mathsf{BlindTx}_{\mathcal{A}}$.

Functionality $\mathcal{G}_{\mathrm{PL}}$

$\mathcal{G}_{\mathrm{PL}}$ is parameterized by seven algorithms, $\mathsf{Validate}$, $\mathsf{ExtendPolicy}$, $\mathsf{Blockify}$, Lkg, $\mathsf{BlindTx}$, Blind, and $\mathsf{predict\text{-}time}$, along with three parameters: $\mathtt{windowSize}, \mathtt{Delay} \in \mathbb{N}$, and $\mathfrak{C}_1 := \{(U_1, s_1), \dots, (U_n, s_n)\}$. These parameters are all publicly known. The functionality manages variables state, NxtBC, buffer, τ_L, and $\vec{\tau}_{\mathsf{state}}$, as described in [2, 1], as well as a sequence of generated IDs, ids. The variables are initialized as follows: $\mathsf{state} := \vec{\tau}_{\mathsf{state}} := \mathsf{NxtBC} := \mathsf{ids} := \varepsilon$, $\mathsf{buffer} := \varnothing$, $\tau_L = 0$.

The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (following the definition of de-synchronized from above). The sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all initially set to $\varnothing$. When a new honest party is registered at the ledger, if it is registered with the clock and the global RO already, then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_L > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is de-registered, it is removed from $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \varnothing$. Finally, during registration, $(\textsc{generate}, \mathrm{sid}, \textsc{coin})$ is run once, and U_p is replaced with the resulting id in $\mathfrak{C}_1$. Further, the registration procedure returns id.
For each party $U_p \in \mathcal{P}$ the functionality maintains a pointer pt_p (initially set to 1) and a current-state view $\mathsf{state}_p := \varepsilon$ (initially set to empty). We refer to the vector $\mathsf{pt}_1, \dots, \mathsf{pt}_n$ as $\vec{\mathsf{pt}}$. The functionality also keeps track of the timed honest-input sequence (cf. [2]) in a vector $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T := \varepsilon$).

Handling initial stakeholders: If during round $\tau = 0$, the ledger did not received a registration from each initial stakeholder, i.e., $(U_p, s_p) \in \mathfrak{C}_1$, the functionality halts.

Upon receiving any input I from any party or from the adversary, send $(\textsc{clock-read}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathrm{CLOCK}}$; upon receiving response $(\textsc{clock-read}, \mathrm{sid}_C, \tau)$ set $\tau_L := \tau$ and do the following if $\tau > 0$ (otherwise, ignore input):

1. Let $\widehat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of de-synchronized honest parties that have been registered (continuously) since time $\tau' < \tau_L - \mathtt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.

2. If I was received from an honest party $U_p \in \mathcal{P}$:

   (a) If $I = (\textsc{submit}, \mathrm{sid}, \mathsf{tx})$, set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T \parallel ((\textsc{submit}, \mathrm{sid}, \mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{tx})), U_p, \tau_L)$; else set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T \parallel (I, U_p, \tau_L)$

   (b) Compute $\vec{N} = (\vec{N}_1, \dots, \vec{N}_\ell) := \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \mathsf{NxtBC}, \mathsf{buffer}, \vec{\tau}_{\mathsf{state}})$ and if $\vec{N} \neq \varepsilon$

set $\mathsf{state} := \mathsf{state} \,\|\, \mathsf{Blockify}(\vec{N}_1) \,\|\, \ldots \,\|\, \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\mathsf{state}} := \vec{\tau}_{\mathsf{state}} \,\|\, \tau_L^\ell$, where $\tau_L^\ell := \tau_L \,\|\, \ldots \,\|\, \tau_L$.

(c) For each $\mathsf{BTX} \in \mathsf{buffer}$: if $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}, \vec{\mathsf{pt}}, \mathcal{H}, \mathsf{ids}) = 0$ then delete BTX from buffer. Also, reset $\mathsf{NxtBC} := \varepsilon$.

(d) If there exists $U_j \in \mathcal{H} \setminus \mathcal{P}_{DS}$ such that $|\mathsf{state}| - \mathsf{pt}_j > \texttt{windowSize}$ or $\mathsf{pt}_j < |\mathsf{state}_j|$, then set $\mathsf{pt}_k := |\mathsf{state}|$ for all $U_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

3. If the calling party U_p is *stalled* (according to the definition above), then no further actions are taken. Otherwise, depending on the above input I and its sender's ID, $\mathcal{G}_{\mathrm{PL}}$ executes the corresponding code from the following list:

– *Submitting a transaction:*
If $I = (\textsc{submit}, \mathsf{sid}, \mathsf{tx})$ and is received from a party $U_p \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party U_p) do the following

(a) Choose a unique transaction ID txid and set $\mathsf{BTX} := (\mathsf{tx}, \mathrm{txid}, \tau_L, U_p)$.

(b) If $\mathsf{Validate}(\mathsf{BTX}, \mathsf{state}, \mathsf{buffer}, \vec{\mathsf{pt}}, \mathcal{H}, \mathsf{ids}) = 1$, then $\mathsf{buffer} := \mathsf{buffer} \cup \{\mathsf{BTX}\}$.

(c) Send $(\textsc{submit}, \mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{BTX}))$ to $\mathcal{A}$.

– *Generating IDs:*
If $I = (\textsc{generate}, \mathsf{sid}, \mathsf{tag})$ is received from a party $U_p \in \mathcal{P}$, query the adversary with $(\textsc{generate}, \mathsf{sid}, U_p, \mathsf{tag})$, denoting the response id. Ensure the response is unique for tag and not equal to $\perp$, and record $\mathsf{ids} \leftarrow \mathsf{ids} \,\|\, (U_p, \mathsf{tag}, \mathsf{id})$. Return id.

– *Reading the state:*
If $I = (\textsc{read}, \mathsf{sid})$ is received from a party $U_p \in \mathcal{P}$ then set $\mathsf{state}_p := \mathsf{state}|_{\min\{\mathsf{pt}_p, |\mathsf{state}|\}}$ and return $(\textsc{read}, \mathsf{sid}, \mathsf{Blind}(\{U_p\}, \mathsf{ids}, \mathsf{state}_p))$ to the requestor. If the requestor is $\mathcal{A}$ then send $(\mathsf{Blind}_{\mathcal{A}}(\mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{state}), \mathsf{map}(\mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P} \setminus \mathcal{H}), \mathsf{ids}, \mathsf{buffer}), \mathsf{Lkg}(\mathsf{state}, \mathsf{buffer}, \tau_L), \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.

– *Maintaining the ledger state:*
If $I = (\textsc{maintain-ledger}, \mathsf{sid})$ is received by an honest party $U_p \in \mathcal{P}$ and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \widehat{\tau} > \tau_L$ then send $(\textsc{clock-update}, \mathrm{sid}_C)$ to $\mathcal{G}_{\textsc{clock}}$. Else send I to $\mathcal{A}$.

– *The adversary proposing the next block:*
If $I = (\textsc{next-block}, \mathrm{hFlag}, (\mathrm{txid}_1, \ldots, \mathrm{txid}_\ell))$ is sent from the adversary, update NxtBC as follows:

(a) Set listOfTxid $\leftarrow \epsilon$

(b) For $i = 1, \ldots, \ell$ do: if there exists $\mathsf{BTX} := (x, \mathrm{txid}, \tau_L, U_j) \in \mathsf{buffer}$ with ID $\mathrm{txid} = \mathrm{txid}_i$ then set listOfTxid := listOfTxid $\,\|\, \mathrm{txid}_i$.

(c) Finally, set $\mathsf{NxtBC} := \mathsf{NxtBC} \,\|\, (\mathrm{hFlag}, \mathrm{listOfTxid})$ and output $(\textsc{next-block}, ok)$ to $\mathcal{A}$.

– *The adversary setting state-slackness:*
If $I = (\textsc{set-slack}, (U_{i_1}, \widehat{\mathsf{pt}}_{i_1}), \ldots, (U_{i_\ell}, \widehat{\mathsf{pt}}_{i_\ell}))$, with $\{U_{p_{i_1}}, \ldots, U_{p_{i_\ell}}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:

(a) If for all $j \in [\ell]$: $|\mathsf{state}| - \widehat{\mathsf{pt}}_{i_j} \leq \mathtt{windowSize}$ and $\widehat{\mathsf{pt}}_{i_j} \geq |\mathsf{state}_{i_j}|$, set $\mathsf{pt}_{i_1} := \widehat{\mathsf{pt}}_{i_1}$ for every $j \in [\ell]$ and return (SET-SLACK, ok) to $\mathcal{A}$.

(b) Otherwise set $\mathsf{pt}_{i_j} := |\mathsf{state}|$ for all $j \in [\ell]$.

– *The adversary setting the state for desychronized parties:*
If $I = (\text{DESYNC-STATE}, (U_{i_1}, \mathsf{state}'_{i_1}), \ldots, (U_{i_\ell}, \mathsf{state}'_{i_\ell}))$, with $\{U_{i_1}, \ldots, U_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\mathsf{state}_{i_j} := \mathsf{state}'_{i_j}$ for each $j \in [\ell]$ and return (DESYNC-STATE, ok) to $\mathcal{A}$.

5.1 Blinding for Forward-Secure Transactions

In order to define blinding on transactions, we first define transactions as consisting of a vector of sub-transactions, denoted $\mathsf{tx} \triangleq (\mathsf{stx}_1, \mathsf{stx}_2, \ldots, \mathsf{stx}_\ell)$. Each sub-transaction consists of a recipient public key pk_r, and an arbitrary message x, that is $\mathsf{stx} \triangleq (pk_r, x)$. In this context, pk_r is either a public key, generated by a party with an GENERATE query, or the special symbol PUBLIC, denoting the sub-transaction is publicly readable. We do not leak the entire annotated transaction to the adversary. Instead, the adversary is shown a modified vector tx, with sub-transactions addressed to honest parties replaced with $\perp$. While we do not go into the detail of transfer transactions here, we also replace components referring to already spent coins – for honest parties or adversarial – with $\perp$. This guarantees forward privacy of past transactions, as even on corruption, the adversary cannot retrieve this information. Concretely, we define *blinding* functions $\mathsf{BlindSTx}$ and $\mathsf{BlindTx}$, described below, which hide parts of the ledger from read requests.

$\mathsf{BlindTx}$ takes as input the full ledger state state, an annotated transaction $\mathsf{BTX} = (\mathsf{tx}, \mathrm{txid}, \tau_L, U_s)$, a set of parties $\mathcal{P}$, and the set of generated ids ids. It returns a vector consisting only of the components of the transaction that are readable by some party $U_p \in \mathcal{P}$. An adversarial version of $\mathsf{BlindTx}$, $\mathsf{BlindTx}_{\mathcal{A}}$, additionally returns the time of submission, τ_L, and the submitter U_s[4]. Below, we make use of the commonly used higher-order function map, which applies a function to a list element-wise.

function $\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (pk, \mathsf{stx}))$
  $b \leftarrow 0$
  if stx is not a change subtransaction **then**
    $b \leftarrow 1$
  end if
  if stx is a receipt subtransaction of an already spent coin **then**
    $b \leftarrow 1$
  end if
  if $pk \neq$ PUBLIC $\vee$ pk not owned by $U_p \in \mathcal{P}$ **then**
    $b \leftarrow 0$
  end if
  if $b \vee \mathsf{stx}$ is a receipt subtransaction for an adversarial coin **then**
    return (pk, stx)
  else
    return $(\perp, |\mathsf{stx}|)$
  end if
end function

[4]If we assumed an anonymous broadcast, the submitter would not be needed to be leaked, i.e., the requirement of leaking the submitter is strictly due to network leakage.

$$\mathsf{BlindTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \mathrm{txid}, \cdot, \cdot)) \triangleq \mathsf{map}(\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}), \mathsf{tx}), \mathrm{txid})$$

$$\mathsf{BlindTx}_{\mathcal{A}}(\mathsf{state}, \mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \mathrm{txid}, \tau_L, U_s)) \triangleq (\mathsf{map}(\mathsf{BlindSTx}(\mathsf{state}, \mathcal{P}, \mathsf{ids}), \mathsf{tx}), \mathrm{txid}, \tau_L, U_s)$$

5.2 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 τ, 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 [13]. 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 Ouroboros 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 [13, 1].

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 adversarially sent coins are spent.

Algorithm $\mathsf{Lkg}_{\mathsf{lead}}$ for $\mathcal{G}_{\mathrm{PL}}$

The $\mathsf{Lkg}_{\mathsf{lead}}$ algorithm maintains a record of past leaks, L_τ for each past time τ. This is to ensure the adversary is limited in accessing the leakage function for past slots.

procedure $\mathsf{Lkg}_{\mathsf{lead}}(\mathsf{state}, \mathsf{buffer}, \tau)$
  if L_τ is recorded **then return** L_τ
  Determine ep, the epoch for the time slot τ.
  Determine τ_{ep}, the time at which the stakeholder distribution for the epoch ep was frozen.
  Let $L \leftarrow \varnothing$
  for each party U_p **do**
    Determine the valid coins of U_p in $\mathsf{state}_{\tau_{ep}}$.
    Determine U_p's relative stake α_{U_p}.
    With probability of $\phi_f(\alpha_{U_p})$, add U_p to L.
  end for
  for each adversarially generated coin **c do**
    if c was spent in state or buffer **then**
      Let tx be the transaction it was spent in
      Let i be the index of the coin in the transaction
      Let $S \leftarrow S \cup \{(\mathsf{tx}, i)\}$
    end if
  end for
  Record $L_\tau \leftarrow L$, and **return** L, S
end procedure

In a preliminary step of our analysis we also utilize a leakage function leaking all information, $\mathsf{Lkg}_{\mathsf{id}}$. This is effectively the identity function, simply returning the parameters state, buffer, and τ passed to it. With this leakage the private ledger effectively becomes a standard ledger from [2, 1], with a stricter interface to the environment, as the simulator still receives all information it would with the standard non-private ledger.

6 The Ouroboros-Crypsinous Protocol

In this section we provide a detailed description of our protocol Ouroboros-Crypsinous as a (G)UC protocol. The protocol has a similar structure as Ouroboros-Genesis [1], but differs considerably in the leaderelection, 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.

6.1 Ideal-World Transactions

Before we delve into the protocol details, we note that unlike many other ledger protocols, we assign meaning to transactions, and this meaning, while more precisely defined later on, is helpful to understand the high-level design. Specifically, 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 realization of transfers using a Zerocash-like [4] 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: $((\text{PUBLIC}, \text{TRANSFER}), (pk_r, \mathbf{c}_4), (pk_s, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3))$, 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. The validation predicate ensures the total value is preserved across the transfer, and that an ID is only spent by its generating party. IDs must originate from the ledgers GENERATE interface, otherwise they are treated as invalid.

In the real world, the design looks slightly different, following the approach of Zerocash [4]. 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, $sn_{\mathbf{c}}$, as well as prove the existence of a valid commitment, $cm_{\mathbf{c}}$, 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. One key difference is the design of addresses, corresponding to the Ideal-world IDs. Parties will generate a new coin public/secret key pair when given a GENERATE query, and will update their secret key after spending a coin with it.

To become a leader at a time τ, parties must prove knowledge of a path in a local Merkle tree of secret keys sk^{COIN}, labeled with τ. 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, and recomputing parts of the bottom of the tree

as needed. It is parameterized by the number of leaves R, which we leave as a system parameter, although we note it could also be defined per-user.

A users public key is derived from the root of the Merkle tree, root, and the time it was created, τ. 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 $\perp$.

6.2 Protocol overview

The protocol Ouroboros-Crypsinous assumes as hybrids a network $\mathcal{F}_{\text{N-MC}}^{\Delta}$, a non-interactive-zero-knowledge scheme $\mathcal{F}_{\text{NIZK}}$, a forward-secure encryption scheme $\mathcal{F}_{\text{FwENC}}$, a global clock $\mathcal{G}_{\text{CLOCK}}$, a global random oracle $\mathcal{G}_{\text{RO}}$, a non-interactive equivocal commitment protocol, and a CRS used by the commitment protocol, to supply the commitment public key, $\mathcal{F}_{\text{CRS}}$.

The protocol execution proceeds in disjoint, consecutive time intervals called *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. Further, due to serial numbers being revealed, and being committing, an 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 if by a paranoid party, by it immediately transferring coins to itself on receipt. Further, it is only an issue for parties which *may be corrupted*. In a hypothetical setting where the adversary could commit 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 [4]. 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, $\mathcal{F}_{\text{NIZK}}$ 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 coins in leadership transactions have their nonce deterministically derived from the nonce of the old coin. The leadership test itself utilizes 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. Further, 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 U_p 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 Section 4.3, 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" $\mathsf{root}_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}}$. η_{ep} is generated similarly to Ouroboros Genesis – it is initially supplied through the CRS, then for subsequence 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 Section 4.3, for a given group G. If the MUPRF output y is below a certain threshold $T_{\mathbf{c}}$—which depends on $\mathbf{c}$'s stake–then U_p 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 $cm_{\mathbf{c}}$. 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: $\rho_{\mathbf{c}'} = \mathsf{PRF}^{\mathsf{evl}}_{\mathsf{root}_{\mathbf{c}}}(\rho_{\mathbf{c}})$. 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.

We note that, as in Ouroboros-Genesis, it is possible for multiple, or no party to be a leader of any given slot. Our protocol behaves identically to Genesis in this regard, and we utilize the same chain selection rule in our protocol.

We next turn to the formal specification of the protocol Ouroboros-Crypsinous. We note that our party management is identical to that of Ouroboros Genesis, and our protocol description follows the same modular design as Ouroboros Genesis. For brevity we will not re-state parts of the genesis protocol which remain unmodified, and we will leave precise UC specification of protocol components to Appendix C.

6.3 Real-world Transactions

Before giving the formal specification we introduce some necessary terminology and notation. Each party U stores a local blockchain $\mathcal{C}^{U_p}_{\mathsf{loc}}$—$U_p$'s local view of the blockchain.[5] Such a local blockchain is a sequence of blocks B_i ($i > 0$) where each $B \in \mathcal{C}_{\mathsf{loc}}$ has the following format: $B = (\mathsf{tx}_{\mathsf{lead}}, \mathsf{st})$; where

[5]For brevity, wherever clear from the context we omit the party ID from the local chain notation, i.e., write $\mathcal{C}_{\mathsf{loc}}$ instead of $\mathcal{C}^{U}_{\mathsf{loc}}$.

$\mathsf{tx}_{\mathsf{lead}} = (\text{LEAD}, \vec{\mathsf{stx}}_{\mathsf{ref}}, \mathsf{stx}_{\mathsf{proof}})$, and $\mathsf{stx}_{\mathsf{proof}} = (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, ep, sl, \rho, h, ptr, \pi)$. 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, $(cm_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{Comm}(pk^{\text{COIN}} \,\|\, \tau \,\|\, v_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}'})$ is the commitment of the newly-created coin, and $sn_{\mathbf{c}} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}^{\text{COIN}}_{sk}}(\rho_{\mathbf{c}})$ is the serial number of the coin $\mathbf{c}$, which is revealed to demonstrate the coin has not been spent. We define $\rho = \mu^{sk^{\text{COIN}}_{sl}}$, where μ is $\mathcal{G}_{\text{RO}}$ evaluated at $\text{NONCE} \,\|\, \eta_{ep} \,\|\, sl$; ρ 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 in Appendix D). The component $\vec{\mathsf{stx}}_{\mathsf{ref}}$ 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.

Ouroboros Crypsinous handles three kinds of transactions: *Leadership transactions*, such as the above $\mathsf{tx}_{\mathsf{lead}}$, *transfer transactions* $\mathsf{tx}_{\mathsf{xfer}}$, 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 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 $\mathsf{tx}^{\mathsf{ideal}}_{\mathsf{xfer}} = ((\text{PUBLIC}, \text{TRANSFER}), (pk_r, (\mathsf{id}_4, v_4)), (pk_s, (\mathsf{id}_1, v_1), (\mathsf{id}_2, v_2), (\mathsf{id}_3, v_3)))$ – with a cryptographic construction hiding the respective information. We define a real transfer transaction to be: $\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}} = (\text{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, c_r)$, where $\mathsf{stx}_{\mathsf{proof}} = (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \tau, \mathsf{root}, \pi)$, and c_r is a $\mathcal{F}_{\text{FWENC}}$-encryption for the slot the transaction was submitted of $\mathsf{stx}_{\mathsf{rcpt}} = (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_{\mathbf{c}_3})$ to pk_r. Similar to leadership transactions, $(cm_{\mathbf{c}_3}, r_{\mathbf{c}_3}) = \mathsf{Comm}(pk^{\text{COIN}}_{pk_s} \,\|\, \tau \,\|\, v_{\mathbf{c}_3} \,\|\, \rho_{\mathbf{c}_3})$, and $(cm_{\mathbf{c}_4}, r_{\mathbf{c}_4}) = \mathsf{Comm}(pk^{\text{COIN}}_{pk_r} \,\|\, \tau \,\|\, v_{\mathbf{c}_4} \,\|\, \rho_{\mathbf{c}_4})$; $sn_{\mathbf{c}_1}$ and $sn_{\mathbf{c}_2}$ are revealed to spend the coins $\mathbf{c}_1$ and $\mathbf{c}_2$ respectively, and π proves the statement XFER (defined in Appendix D), specifically proving the existence of $cm_{\mathbf{c}_1}$ and $cm_{\mathbf{c}_2}$, 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. Further, parties maintain locally the information needed to spend coins they own – specifically $(pk^{\text{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$.

6.4 Interacting with the Ledger

At the core of the Ouroboros 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 $(\text{SUBMIT}, \mathsf{sid}, \mathsf{tx})$ 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 parties' local transaction buffer, and multicast to the network.

- The command (GENERATE, sid) is used for creating a new address, which can be used by other parties to transfer funds to this current party.
- The command (READ, sid) 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, sid) 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 τ, 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, sid) message then before doing anything else, the party invokes an initialization 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 LedgerMaintenance, a detailed specification of the protocol ReadState for answering requests to read the ledger's state, and a detailed specification of the protocols SubmitXfer and SubmitGeneric.

Party Initialization A party that has been registered with all its resources and setups becomes operational by invoking the initialization protocol Initialization-Crypsinous upon processing its first command. As a first step the party receives its encryption key from $\mathcal{F}_{\text{FWENC}}$. It receives any initial stake it may have as a single coin from $\mathcal{F}_{\text{INIT}}$. Subsequently, protocol Initialization-Crypsinous proceeds as in Ouroboros-Genesis, although it does not register any keys. This is managed instead by the ledgers GENERATE interface. The precise description of the initialization can be found in Appendix C.

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 U_p is an eligible slot leader for a particular slot sl in an epoch ep if, one of U_p'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_{τ}^{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 the leadership transaction a coin originates from to not be present in the chain the party is currently attempting to extend. In this case, the coin is *still eligible*, as the originating leadership transaction will be added to $\vec{\mathsf{stx}}_{\mathsf{ref}}$.

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, $T_{\mathbf{c}} = \mathsf{ord}(G)\phi_f(\alpha_{\mathbf{c}})$. 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 it 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 technical description of the staking procedure can be found in Appendix C.2. It evaluates two district 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 $\mathcal{C}_{\mathsf{loc}}$ containing the new block B is finally multicast over the network.

From the staking procedure we construct the ledger maintenance protocol, which in addition to attempting to stake on each block, monitors incoming transactions and chains, decrypts ciphertexts where possible, updates the parties local state by adding received coins, records received messages, and performs the chain selection of [1]. The full description can be found in Appendix C.3

Submitting Transactions Transactions submitted to the Ouroboros Crypsinous protocols 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 assembled into blocks. Specifically, transfer transactions are mapped to Zerocash-like transactions, where only the first coin received to a given address it spent, and other transactions are mapped into encrypted components. The submitting procedure for transfer transactions is described in Appendix C.4, and that for generic transaction in Appendix C.5.

Reading the State The last command related to the interaction with the ledger is the read command (READ, sid) 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 map 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 full description of this reconstruction can be found in Appendix C.6.

6.5 Transaction Validity

Transaction validity again differs in the real and ideal world, as the transactions themselves differ.

Ideal World Validation The ideal world validation predicate validates only transfer transactions. It is parameterized by the initial distribution of coins $\mathfrak{C}_1$. It maintains, for each ID, an ordered sequence of received values, the ID's owner, and a flag marking whether the ID has already been used for spending. For each transfer transaction validated, first its format is enforced. Next, it asserts that $v_1 + v_2 = v_3 + v_4$. It checks that the IDs of $\mathbf{c}_1$ and $\mathbf{c}_2$ have indeed received transfer of value v_1 and v_2 respectively (and, if the IDs and value are equal, have received at least *two* transfers of that value). If there is ambiguity as to which coins to spend, those received first are spent. As a special case, if the ID of $\mathbf{c}_2$ is $\perp$, and $v_2 = 0$, it is always valid.[6] It is further checked that the coins the party is trying to spend are "old enough", specifically, they must be in the parties local view of the ledger state. (The validation predicate has access to the parties state pointer). If the sending party is honest, we further restrict it to only spending coins to which it owns the ID. Further, honest parties must address $\mathsf{stx}_{\mathsf{chng}}$ to their own public key – i.e. the first value generated

[6] This permits parties with only one coin to spend it.

by (GENERATE, sid, ID) by the party. If the sending party is corrupted, it may spend the coins of other corrupted parties, as well as arbitrary received values. If other transactions in the *buffer* attempt to spend the same coins, and the transaction is honest, it is also rejected – as in this case the party is attempting to double spend and de-anonymize themselves.

Finally, if the transaction is valid, a new receipt of a value of v_3 is recorded for $\mathbf{c}_3$, and respectively with v_4, and $\mathbf{c}_4$. The values spent are erased from the values lists of $\mathbf{c}_1$ and $\mathbf{c}_2$'s IDs, and their "spent" flags are set (with the exception of the id $\perp$).

Real-world Validation The real-world validation predicate maintains three sets, the sets of coin commitments $\mathbb{C}^{\mathsf{spend}}$, $\mathbb{C}^{\mathsf{lead}}$ for spending and leadership respectively, initialized to the initial set of coin commitments $\mathbb{C}_1$, and the set of spent serial numbers $\mathbb{S}$, initialized to $\varnothing$. A chain is validated transaction by transaction. Leadership transactions and transfer transactions are both validated, other transactions are ignored. A leadership transaction is valid iff all leadership transactions in $\vec{\mathsf{stx}}_{\mathsf{ref}}$ 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. Further, 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, $\mathbb{S} \leftarrow \mathbb{S} \cup \{sn\}$, $\mathbb{C}^{\mathsf{lead}} \leftarrow \mathbb{C}^{\mathsf{lead}} \cup \{cm\}$, $\mathbb{C}^{\mathsf{spend}} \leftarrow \mathbb{C}^{\mathsf{spend}} \cup \{cm\}$.

Transfer transactions are likewise validated by checking the NIZK proof with respect to the public transaction component. Further, it is checked that root was at some point the root of $\mathbb{C}^{\mathsf{spend}}$, and that $\{sn_1, sn_2\} \cap \mathbb{S} = \varnothing$. If so, the effect is updating $\mathbb{S} \leftarrow \mathbb{S} \cup \{sn_1, sn_2\}$, and $\mathbb{C}^{\mathsf{spend}} \leftarrow \mathbb{C}^{\mathsf{spend}} \cup \{cm, cm_3\}$. Finally, at the start of an epoch, old enough spending coins are allowed for leadership proofs: $\mathbb{C}^{\mathsf{lead}} \leftarrow \mathbb{C}^{\mathsf{lead}} \cup \mathbb{C}^{\mathsf{spend}}_{t-k}$, where $\mathbb{C}^{\mathsf{spend}}_{t-k}$ 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 $\vec{\mathsf{stx}}_{\mathsf{ref}}$ (i.e. 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.

Generic transactions are valid if and only if they do not start with the symbol (PUBLIC, TRANSFER).

7 Security Analysis

We split our security analysis of Ouroboros-Crypsinous into two parts: In a first, warm-up part, we show that Ouroboros-Crypsinous realizes a "non-private" version of $\mathcal{G}_{\mathrm{PL}}$ – specifically, we show that it realizes $\mathcal{G}_{\mathrm{PL}}$ with Lkg set to the identity function $\mathsf{Lkg}_{\mathsf{id}}$; i.e. the ledger leaks its entire content to the simulator, described in detail in Appendix B. We argue that the simulator $\mathcal{S}_1$ can simulate any real-world attacks on Ouroboros-Crypsinous against a non-private $\mathcal{G}_{\mathrm{PL}}$. 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 satisfied 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 $\mathcal{S}_2$ who is able, with access only to this restricted leakage to simulate the outputs of $\mathcal{S}_1$. generate a view which is indistinguishable from $\mathcal{S}_1$.

Theorem 1. Ouroboros-Crypsinous*, in the* $(\mathcal{W}_{\mathrm{OC}}^{PoS}(\mathcal{F}_{\mathrm{NIZK}}^{\mathcal{L}_{\mathrm{LEAD}}}, \mathcal{F}_{\mathrm{NIZK}}^{\mathcal{L}_{\mathrm{XFER}}}, \mathcal{F}_{\mathrm{FWENC}}, \mathcal{F}_{\mathrm{N\text{-}MC}}^{\Delta}), \mathcal{G}_{\mathrm{RO}}, \mathcal{G}_{\mathrm{CLOCK}})$*-hybrid world, UC-emulates* $\mathcal{G}_{\mathrm{PL}}$ *with* $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$*, under the DDH assumption.*[7]

Proof (sketch). The backbone of the proof of Theorem 1 is similar to the security proof of Ouroboros Genesis [1] 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 argue 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 [1] in Step 4 for proving that Crypsinous implements, at the very 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- and post-processing has the same effect as blinding a transaction in the ideal world, and that the validation predicate of Ouroboros-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 $\mathcal{G}_{\mathrm{PL}}$ with $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$.

Step 1. The security properties guaranteed by $\mathcal{F}_{\mathrm{KES}}$, and used in [1], 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 $\mathcal{F}_{\mathrm{KES}}$, but sufficient for how it is used in [1]. 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 [17]. 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 Ouroboros 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. Further, as the statement LEAD has the same conditions as a leadership proof in [1], the desired authenticity property is also satisfied. This is not sufficient to emulate $\mathcal{F}_{\mathrm{KES}}$, however using $sk_s^{\mathrm{COIN}}l$ and $\rho_{\mathbf{c}}$ in the witness rectifies this. As honest parties update both $sk_s^{\mathrm{COIN}}l$ and $\rho_{\mathbf{c}}$ after the proof, and $sk_s^{\mathrm{COIN}}l$ 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. While this is effective only so long as $sk_s^{\mathrm{COIN}}l$ and $\rho_{\mathbf{c}}$ cannot be retrieved from elsewhere. $sk_s^{\mathrm{COIN}}l$ is generated locally by an honest party, is never communicated by it (except to $\mathcal{F}_{\mathrm{NIZK}}$, 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 generation of MUPRFs. Two VRF calls are embedded in the NIZK; the VRF used to generate the randomness contribution ρ, and the VRF used to check the target.

[7] 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.

While in Ouroboros 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 this is simply as the verification is stricter. The NIZK inputs in Ouroboros Crypsinous depend on the coin 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 Ouroboros-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 $\mathcal{G}_{\text{LEDGER}}$ proof of [1], the rest of the proof can be carried out the same for Ouroboros-Crypsinous. This establishes that, Ouroboros-Crypsinous effectively runs an internal ledger. While the transactions posted to this ledger are not directly those posted to Ouroboros-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 Ouroboros-Crypsinous. For brevity, we will refer to the former mapping as f, and the latter as $f_{U_p}^{-1}$. We define *consistency* of this mapping to be two the following two properties things: First, a validation predicate – specifically instantiated to that of Ouroboros-Crypsinous – over the mapped transactions must exists that holds if and only if the ideal-world validation predicate over the original transactions holds. Second, $f_{U_p}^{-1} \circ f = \mathsf{BlindTx}(\{U_p\})$ – i.e. READ requests return the same as $f_{U_p}^{-1}$ 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 U_p are encrypted with pk_p^{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 BlindTx exactly. Finally, the validation predicate is always true for generic transactions in the ideal world, and is only false for generic transactions that start with TRANSFER in the real world – which have no ideal world equivalent, and should cannot be created by honest parties.

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-, 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 public key $pk_{\mathbf{c}}^{\text{COIN}}$), 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, i.e. 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 $\mathcal{F}_{\text{INIT}}$, 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 maintains the induction hypothesis.

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_{τ}^{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.

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 and 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 validation predicate will detect and block the reuse of the coins serial number.

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 the validation predicate (which does not care about the order of adversarial transaction until they enter the state). The set of coins induces a stakeholder distribution, as required by the proof of [1].

Finally, the inverse mapping of parties views correspond to their ideal-world views. Specifically, if the party sees *anything* in the ideal world, it is the recipient of a coin, in which case it it need only 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 – iff 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. We conclude that honest parties response 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. Further, we are considering an overly permissive leakage predicate, $\mathsf{Lkg}_{\mathsf{id}}$, which provides the adversary with the same information it would receive from the standard ledger satisfying b). Finally, Ouroboros Crypsinous allows ID generation, which 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, $\mathcal{F}_{\text{FwEnc}}$ public keys, which are guaranteed uniqueness, or randomly samples values from $\{0,1\}^{\kappa}$, which have a negligible probability of collision. We conclude that $\mathsf{Ouroboros\text{-}Crypsinous}$ realizes $\mathcal{G}_{\text{PL}}$ with $\mathcal{S}_1$, under the leakage predicate $\mathsf{Lkg}_{\mathsf{id}}$. □

Theorem 2. $\mathsf{Ouroboros\text{-}Crypsinous}$*, in the* $(\mathcal{W}_{\text{OC}}^{PoS}(\mathcal{F}_{\text{NIZK}}^{\mathcal{L}_{\text{LEAD}}}, \mathcal{F}_{\text{NIZK}}^{\mathcal{L}_{\text{XFER}}}, \mathcal{F}_{\text{FwEnc}}, \mathcal{F}_{\text{N-MC}}^{\Delta}), \mathcal{G}_{\text{RO}}, \mathcal{G}_{\text{CLOCK}})$*-hybrid world, UC-emulates* $\mathcal{G}_{\text{PL}}$ *with* $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{lead}}$ *under the DDH assumption.*

Proof (sketch). The leakage $\mathsf{Lkg}_{\mathsf{lead}}$ leaks only the leader of any given slot. We utilize 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. The modified simulator, $\mathcal{S}_2$ is described in detail in Section B.2. 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 $\mathcal{G}_{\text{PL}}$ 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, $\mathcal{C}_{\mathsf{loc}}$, the local buffer buffer, the set of coins $\mathfrak{C}$ (as well as $\mathfrak{C}_{\mathsf{free}}$, and $\mathfrak{C}_{\mathsf{cnd}}$), and the log of transfer interactions, and ciphertext to plaintext mappings, log. Maintaining $\mathcal{C}_{\mathsf{loc}}$, and buffer is straightforward, as the network interactions directly dictate their content, 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 $\mathcal{F}_{\text{FWENC}}$. The corrupted party can only be the recipient U_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 $\mathcal{F}_{\text{FWENC}}$ 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 $pk^{\text{COIN}}_{\mathbf{c}}$, $v_{\mathbf{c}}$ from the corresponding ideal-world transaction. As the ideal-world transaction is valid, we know $pk^{\text{COIN}}_{\mathbf{c}}$ must be a valid ID for the corrupted party, in which case the simulator provided it, and knows the corresponding secret key $sk^{\text{COIN}}_{\mathbf{c}}$. It then opens the commitment $cm_{\mathbf{c}}$ to $pk^{\text{COIN}}_{\mathbf{c}} \,\|\, v_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}}$, 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 $\mathcal{F}_{\text{FWENC}}$ 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 $sk^{\text{COIN}}_{\mathbf{c}'} = \mathsf{PRF}^{\mathsf{evl}}_{sk^{\text{COIN}}_{\mathbf{c}}}(1)$, and $\rho_{\mathbf{c}'} = \mathsf{PRF}^{\mathsf{evl}}_{sk^{\text{COIN}}_{\mathbf{c}}}(\rho_{\mathbf{c}})$, opening $cm_{\mathbf{c}'}$, the commitment in the corresponding real-world leadership proof to $pk^{\text{COIN}}_{\mathbf{c}'} \,\|\, v_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}'}$, 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 $sk^{\text{COIN}}_{\mathbf{c}'}$, 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 are either purely random, or originates from the equivocal commitment scheme, the adversary cannot distinguish the corrupted parties state from the real parties state.

Step 3. We conclude from Theorem 1, and our observations in Steps 1 and 2, combined with the fact that $\mathcal{S}_1$ and $\mathcal{S}_2$ differ only in simulating transactions and corruption, that Theorem 2 holds. □

8 Performance Estimation

Coin transfers are modeled after Zerocash's [4] pour transactions. This enables us to reuse much of the existing implementation work invested on optimizing the performance critical SNARK operations by the Zcash project, cf. [18].

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 SHA256 compression function. The MUPRF requires variable base group exponentiations. As we require equivocal commitments, we replace the SHA256 coin commitments of Zerocash that require 83,712 constraints with the Pedersen commitments of Sapling [18] 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 2, the multiplication count of a leadership SNARK relation is less 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 have not focused on optimizing this process as Sapling has, by parallelizing the NIZK proofs, we emphasize that even unoptimized, Ouroboros Crypsinous would have a proving time only around double that of Sapling.

Primitive	Approx. constraints
SHA256	27,904
Exponentiation (variable base)	3,252 ([18], page 128)
Hidden range proof	256
Pedersen commitment	1,006 + 2.666 per bit[8]

Table 1: Number of multiplicative constraints in SNARK relations

Constraint count	$\mathcal{L}_{\text{XFER}}$	$\mathcal{L}_{\text{LEAD}}$
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 $cm_{\mathbf{c}_i}$	$2 \times 43,808$	$43,808$
(1 layer of 32)	$(1,369)$	$(1,369)$
Path for $\mathsf{root}_{sk_{\mathbf{c}_i}^{\text{COIN}}}$		$34,225$
(1 layer of 24)		$(1,369)$
(leaf preimage)		$(1,369)$
Check $sn_{\mathbf{c}_i}$	$2 \times 27,904$	$27,904$
Check $cm_{\mathbf{c}_i}$	$4 \times 2,542$	$2 \times 2,542$
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.) $y < \mathsf{ord}(G)\phi_f(v)$		256
Total	209,466	201,493

Table 2: Number of constraints per SNARK statement

We note in passing that the forward-secure encryption scheme is only needed 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.

[8] `https://github.com/zcash/zcash/issues/2634`

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 500kB storage requirement. The most expensive part of the process, key generation, still takes less than a minute on a modern CPU.

References

[1] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros Genesis: Composable proof-of-stake blockchains with dynamic availability. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 913–930, 2018.

[2] 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.

[3] 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.

[4] 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.

[5] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Provably secure proofs of stake. Cryptology ePrint Archive, Report 2016/919, 2016. `http://eprint.iacr.org/2016/919`.

[6] 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, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013.

[7] 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.

[8] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[9] 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.

[10] 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.

[11] Cardano Community. Cardano settlement layer documentation. https://cardanodocs.com/technical/, October 18 2018.

[12] Ivan Damgård and Jens Groth. Non-interactive and reusable non-malleable commitment schemes. In *35th ACM STOC*, pages 426–437. ACM Press, June 2003.

[13] 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.

[14] 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.

[15] Chaya Ganesh, Claudio Orlandi, and Daniel Tschudi. Proof-of-stake protocols for privacy-aware blockchains. Cryptology ePrint Archive, Report 2018/1105, 2018. `https://eprint.iacr.org/2018/1105`.

[16] 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.

[17] 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.

[18] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. 2018.

[19] George Kappos, Haaroon Yousaf, Mary Maller, and Sarah Meiklejohn. An empirical analysis of anonymity in zcash. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 463–477, 2018.

[20] 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.

[21] Ahmed E. Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, T.-H. Hubert Chan, Charalampos Papamanthou, Rafael Pass, Abhi Shelat, and Elaine Shi. How to use snarks in universally composable protocols. *IACR Cryptology ePrint Archive*, 2015:1093, 2015.

[22] Amrit Kumar, Clément Fischer, Shruti Tople, and Prateek Saxena. A traceability analysis of monero's blockchain. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II*, pages 153–173, 2017.

[23] Gregory Maxwell. CoinJoin: Bitcoin privacy for the real world. `https://bitcointalk.org/?topic=279249`, August 2013.

[24] 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. *Commun. ACM*, 59(4):86–93, 2016.

[25] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.

[26] Malte Möser, Kyle Soska, Ethan Heilman, Kevin Lee, Henry Heffan, Shashvat Srivastava, Kyle Hogan, Jason Hennessey, Andrew Miller, Arvind Narayanan, and Nicolas Christin. An empirical analysis of traceability in the monero blockchain. *PoPETs*, 2018(3):143–163, 2018.

[27] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[28] 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.

[29] Rafael Pass and Elaine Shi. The sleepy model of consensus. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 380–409. Springer, Heidelberg, December 2017.

[30] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, pages 6–24, 2013.

[31] 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.

[32] Nicolas van Saberhagen. Cryptonote v 2.0. https://cryptonote.org/whitepaper.pdf, October 17 2013.

[33] Vlad Zamfir. Casper the friendly ghost: A "correct-by-construction" blockchain consensus protocol. `https://github.com/ethereum/research/blob/master/papers/CasperTFG/CasperTFG.pdf`, December 17 2017.

A Hybrid World Functionalities

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

The functionality $\mathcal{F}_{\text{INIT}}$ is parameterized by the number of initial stakeholders n and their respective stakes $s_1, \ldots, s_n$. $\mathcal{F}_{\text{INIT}}$ interacts with stakeholders $U_1, \ldots, U_n$ as follows:

- In the first round, upon a request from some stakeholder U_i of the form $(\mathsf{claim}, \mathrm{sid}, U_i)$, then $\mathcal{F}_{\text{INIT}}$ samples sk^{COIN} as Ouroboros-Crypsinous does on GENERATE requests, $\rho_{\mathbf{c}_i}$ randomly, computes $pk^{\text{COIN}} \leftarrow \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}^{\text{COIN}}_{sk}}(0)$, and commits $(cm_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \mathsf{Comm}(pk^{\text{COIN}} \,\|\, s_i \,\|\, \rho_{\mathbf{c}_i})$, and returns the tuple $(pk^{\text{COIN}}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, s_i)$, along with sk^{COIN}. One all parties have registered, it

samples and stores a random value $\eta_1 \xleftarrow{\$} \{0,1\}^\lambda$. It then constructs a genesis block $(\mathbb{C}_1, \eta_1)$, where $\mathbb{C}_1 = \{cm_{\mathbf{c}_1}, \ldots, cm_{\mathbf{c}_n}\}$.

- If this is not the first round then do the following:
  - If any of the n initial stakeholders has not send a request of the above form, i.e., a $(\mathsf{keys}, \text{sid}, U_i, pk_i^{\text{ENC}})$-message, to $\mathcal{F}_{\text{INIT}}$ in the genesis round then $\mathcal{F}_{\text{INIT}}$ outputs an error and halts.
  - Otherwise, if the currently received input is a request of the form $(\mathsf{genblock\_req}, sid, U_i)$ from any (initial or not) stakeholder U, $\mathcal{F}_{\text{INIT}}$ sends $(\mathsf{genblock}, sid, (\mathbb{C}_1, \eta_1))$ to U.

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

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., parties $U_p = (\text{pid}, \text{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} := \emptyset$ and $F := \emptyset$.

For each session sid the clock maintains a variable τ_{sid}. For each identity $U_p := (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable d_{U_p}. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d_{(\mathcal{F},\text{sid})}$ (all integer variables are initially 0).

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $U_p \in \mathcal{P}$ set $d_{U_p} := 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, U_p)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some functionality $\mathcal{F}$ in a session sid such that $(\mathcal{F}, \text{sid}) \in F$ set $d_{(\mathcal{F},\text{sid})} := 1$, execute *Round-Update* and return $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathcal{F})$ to this instance of $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ from any participant (including the environment on behalf of a party, the adversary, or any ideal–shared or local–functionality) return $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ to the requestor.

Procedure Round-Update: For each session sid do: If $d_{(\mathcal{F},\text{sid})} := 1$ for all $\mathcal{F} \in F$ and $d_{U_p} = 1$ for all honest parties $U_p = (\cdot, \text{sid}) \in \mathcal{P}$, then set $\tau_{\text{sid}} := \tau_{\text{sid}} + 1$ and reset $d_{(\mathcal{F},\text{sid})} := 0$ and $d_{U_p} := 0$ for all parties $U_p = (\cdot, \text{sid}) \in \mathcal{P}$.

Functionality $\mathcal{F}_{\text{NIZK}}^{\mathcal{L}}$

The (proof-malleable) non-interactive zero-knowledge functionality $\mathcal{F}_{\text{NIZK}}^{\mathcal{L}}$ allows proving of statements in an NP language $\mathcal{L}$. It maintains a set of statement/proof pairs Π, initialized to $\varnothing$.

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:*

if $(x, w) \notin \mathcal{L}$ **then**
  return $(\mathsf{proof}, \text{sid}, x, \perp)$
end if
send $(\mathsf{prove}, \text{sid}, x)$ **to** $\mathcal{A}$ and **receive the reply** $(\mathsf{proof}, \text{sid}, x, \pi)$
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$
return $(\mathsf{proof}, \text{sid}, x, \pi)$

Proof Malleability *When receiving a message* $(\mathsf{maul}, sid, x, \pi)$ *from* $\mathcal{A}$*:*

if $\nexists \pi' : (x, \pi') \in \Pi$ **then**
  return $(\mathsf{maul}, \mathrm{sid}, x, \pi, \bot)$
end if
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$
return $(\mathsf{maul}, \mathrm{sid}, x, \pi, \top)$

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*

if $(x, \pi) \notin \Pi$ **then**
  send $(\mathsf{verify}, \mathrm{sid}, x, \pi)$ **to** $\mathcal{A}$ and **receive the reply** R
  if $R = (\mathsf{witness}, \mathrm{sid}, x, \pi, w) \wedge (x, w) \in \mathcal{L}$ **then**
    let $\Pi \leftarrow \Pi \cup (x, \pi)$
  end if
end if
return $(\mathsf{verify}, \mathrm{sid}, x, \pi, (x, \pi) \in \Pi)$

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

$\mathcal{F}_{\text{FwEnc}}$ is parameterized by, a security parameter κ, a set of parties $\mathcal{P}$, and a maximum delay $\Delta_{\max}$.

- **Key Generation.** Upon receiving a message $(\mathsf{KeyGen}, \mathrm{sid})$ from a party U_p, verify that $U_p \in \mathcal{P}$, and that this is the first key generation. If so, send $(\mathsf{KeyGen}, \mathrm{sid}, U_p)$ to $\mathcal{A}$, and receive a value pk_p in return. Return pk_p to U_p, and initialize $\tau_p := 0$ and add U_p to the set of honest parties $\mathcal{H}$.
- **Encryption.** Upon receiving a message $(\mathsf{Encrypt}, \mathrm{sid}, pk, \tau, m)$ from some party U_p:
  - Check that there exists a $U_q \in \mathcal{P}$, where $pk_q = pk$ and $U_q \in \mathcal{H}$, and $\tau < \tau_q + \Delta_{\max}$. If so, send $(\mathsf{Encrypt}, \mathrm{sid}, \tau, |m|, U_p)$ to $\mathcal{A}$. Otherwise, send $(\mathsf{DummyEncrypt}, \mathrm{sid}, pk, \tau, m, U_p)$ to $\mathcal{A}$.
  - Receive a reply c from $\mathcal{A}$, and send $(\mathsf{ciphertext}, c)$ to U_p. Further, if the conditions in the previous step were satisfied, record the tuple (U_q, m, τ, c).
- **Decryption.** Upon receiving a message $(\mathsf{Decrypt}, \mathrm{sid}, \tau', c)$ from party $U_p \in \mathcal{P}$:
  - If $\tau' < \tau_p$, return $\bot$.
  - Else, if a tuple (U_p, m, τ', c) was recorded, return m to U_p.
  - Otherwise, send $(\mathsf{Decrypt}, \mathrm{sid}, \tau_p, c, U_p)$ to $\mathcal{A}$, receive a reply m, and forward m to U_p.
- **Update.** Upon receiving a message $(\mathsf{Update}, \mathrm{sid})$ from party $U_p \in \mathcal{P}$:
  1. Send $(\mathsf{Update}, \mathrm{sid}, U_p)$ to $\mathcal{A}$.
  2. Update $\tau_p \leftarrow \tau_p + 1$
- **Corruptions.** Upon corruption of a party $U_p \in \mathcal{P}$, remove U_p from $\mathcal{H}$.

B The Simulator

B.1 The Stage 1 Simulator

Procedures ExtendLedgerState, and AdjustView as in Ouroboros Genesis, and SimulateStaking as in Ouroboros Genesis for $\mathcal{S}_1$.

Simulator $\mathcal{S}_1$ (Part 1 - Main Structure)

Overview:

- The simulator internally emulates all local UC functionalities by running the code (and keeping the state) of $\mathcal{F}_{\text{INIT}}$, $\mathcal{F}_{\text{NIZK}}$, $\mathcal{F}_{\text{ENC}}$, $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$, and $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$.
- The simulator mimics the execution of Ouroboros-Crypsinous for each honest party U_p (including their state and the interaction with the hybrids).
- The simulator emulates a view towards the adversary $\mathcal{A}$ in a black-box way, i.e., by internally running adversary $\mathcal{A}$ and simulating his interaction with the protocol (and hybrids) as detailed below for each hybrid. To simplify the description, we assume $\mathcal{A}$ does not violate the requirements by the wrapper $\mathcal{W}_{\text{OG}}^{PoS}(\cdot)$ as this would imply no interaction between $\mathcal{S}_1$ (i.e., the emulated hybrids) and $\mathcal{A}$.
- For global functionalities, the simulator simply relays the messages sent from $\mathcal{A}$ to the global functionalities (and returns the generated replies). Recall that the ideal world consists of the dummy parties, the ledger functionality, and the clock.

Party sets:

- As defined in Ouroboros Genesis [1], honest parties are categorized. $\mathcal{S}_{\text{alert}}$ denote synchronized parties that are not stalled, $\mathcal{S}_{\text{syncStalled}}$ are synchronized parties that are stalled, and $\mathcal{P}_{DS}$ are de-synchronized parties.
- For each registered honest party, the simulator maintains the local state containing in particular the local chain $\mathcal{C}_{\mathsf{loc}}^{(U_p)}$, the time t_{on} it remembers when last being online, spendable coins $\mathfrak{C}$, and the log of transactions, log. For each party U_p and clock time τ, the simulator stores a flag $\mathsf{update}_{U_p,\tau}$ (initially false) to remember whether this party has updated its state already in this round. Note that an registered party is registered with all its local hybrids.
- Upon any activation, the simulator will query the current party set from the ledger, and the clock, to evaluate in which category an honest party belongs to. If a new honest party is registered to the ledger, it internally runs the initialization procedure of Ouroboros-Crypsinous.
- We assume that the simulator queries upon any activation for the sequence $\vec{\mathcal{I}}_H^T$, and the current time τ from the clock. We note that the simulator is capable of determining $\mathsf{predict\text{-}time}(\cdot)$ of $\mathcal{G}_{\text{PL}}$.

Messages from the Clock: as in Ouroboros Genesis.

Messages from the Ledger:

- Upon receiving $(\text{SUBMIT}, \mathsf{BTX})$ from $\mathcal{G}_{\text{PL}}$ where $\mathsf{BTX} := (\mathsf{tx}, \text{txid}, \tau, U_p)$, simulate running $(\text{SUBMIT}, \mathsf{BTX})$ as U_p, interacting with the simulated network $\mathcal{F}_{\text{N-MC}}$.
- Upon receiving $(\text{GENERATE}, U_p, \mathsf{tag})$ from $\mathcal{G}_{\text{PL}}$, if tag is ID, and this is the first ID query for U_p, return pk_p^{ENC}, otherwise execute GENERATE as the simulated party U_p and tag.
- Upon receiving $(\text{MAINTAIN-LEDGER}, \text{sid})$ from $\mathcal{G}_{\text{PL}}$, extract from $\vec{\mathcal{I}}_H^T$ the party U_p that issued this query. If U_p has already completed its round-task, then ignore this request. Otherwise, execute $\text{SIMULATESTAKING}(U_p, \tau)$.

Simulator $\mathcal{S}_1$ (Part 2 - Black-Box Interaction)

Simulation of Functionality $\mathcal{F}_{\text{INIT}}$ *towards* $\mathcal{A}$*:*

- The simulator relays back and forth the communication between the (internally emulated) $\mathcal{F}_{\text{INIT}}$ functionality and the adversary $\mathcal{A}$ acting on behalf of a corrupted party.
- If at time $\tau = 0$, a corrupted party $U_p \in \mathcal{S}_{\text{initStake}}$ registers via $(\mathsf{claim}, sid, U_p)$ to $\mathcal{F}_{\text{INIT}}$, then input $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}_{\text{PL}}$ on behalf of U_p. Intercept the keys returned from $\mathcal{F}_{\text{INIT}}$, locally store them, and send the intercepted pk^{COIN} as the id for the coin in $\mathcal{G}_{\text{PL}}$.

Simulation of the Functionalities $\mathcal{F}_{\text{NIZK}}$ *and* $\mathcal{F}_{\text{FwEnc}}$ *towards* $\mathcal{A}$*:*

- The simulator relays back and forth the communication between the (internally emulated) hybrids and the adversary $\mathcal{A}$ (either direct communication, communication to $\mathcal{A}$ caused by emulating the actions of honest parties, or communication of $\mathcal{A}$ on behalf of a corrupted party). Whenever a witness is supplied for a NIZK proof, the given witness is recorded.

Simulation of the Networks $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$*, and* $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$ *as in Ouroboros Genesis, with the following modifications:*

- The simulator records transactions originating from $\mathcal{A}$ or a corrupted party.
- When an adversarial transaction first enters the confirmed state, the simulator attempts to extract the witness.
- If the witness does not extract, abort.
- If the witness is successfully extracted, compute the corresponding ideal-world transaction as follows:
  - From the extracted secret keys and nonces, determine the ideal-world coins being spent. If one does not exist, abort.
  - If the public key of the "change" coin is assigned to the adversary, use it directly in the ideal transaction. If it is assigned to an honest party, generate a new adversarial ID in the ideal world for it, and record the relationship between the coins. If was not previously seen, generate it directly as an adversarial ID.
  - If the "recipient" coin public key is adversarial, use it directly as the coin ID. If it is honest, *and* the transaction's ciphertext is a correct encryption of the coin to the same

honest party, use it directly as the coin ID as well. If it is otherwise honest, again generate a new adversarial ID in the ideal world, and record the relationship between the coins. If it was not previously seen, generate it directly as an adversarial ID.

– Form an ideal-world transaction with the above coin IDs and extracted values.

B.2 The Stage 2 Simulator

Simulator $\mathcal{S}_2$

The Simulator $\mathcal{S}_2$ behaves like $\mathcal{S}_1$, with key differences listed below. The simulator maintains a record of simulated NIZK proofs. When asked to verify a simulated NIZK proof by the adversary through $\mathcal{F}_{\text{NIZK}}$, return $\top$ if the statement provided is the same statement recorded, otherwise return $\bot$. We define ℓ_{Coin} to be the length of coin tuples.

- Upon receiving (SUBMIT, BTX) from $\mathcal{G}_{\text{PL}}$ for honest transactions, if $\mathsf{BTX} = (\text{PUBLIC}, \text{TRANSFER}) \,\|\, \mathsf{BTX}'$, run SIMULATETRANSFER($\mathsf{BTX}'$). Otherwise, run SIMULATEGENERIC(BTX).
- Upon receiving (MAINTAIN-LEDGER, sid) from $\mathcal{G}_{\text{PL}}$, extract from $\vec{\mathcal{I}}_H^T$ the party U_p that issued this query. If U_p has already completed its round-task, then ignore this request. Otherwise, execute SIMULATESTAKING(U_p, τ, L_τ), where L_τ is the leadership leakage for time τ. If this is not yet known, query $\mathcal{G}_{\text{PL}}$ with READ for it.
- Upon the adversary requesting corruption of a party U_p, corrupt the corresponding ideal-world party immediately, and run CORRUPT(U_p).

procedure SIMULATETRANSFER$((\mathsf{stx}^{\mathsf{ideal}}_{\mathsf{rcpt}}, \mathsf{stx}^{\mathsf{ideal}}_{\mathsf{chng}}))$

if $\mathsf{stx}^{\mathsf{ideal}}_{\mathsf{rcpt}} = \bot$ **then**

Let $cm \leftarrow \widehat{\mathsf{Comm}}(ek)$.

Send $(\mathsf{Encrypt}, \text{sid}, \tau, \ell_{\mathsf{Coin}}, U_p)$ to $\mathcal{A}$, and denote the response $\mathsf{stx}^{\mathsf{real}}_{\mathsf{rcpt}}$.

else

Let $(pk_q^{\text{ENC}}, (pk^{\text{COIN}}, v)) \leftarrow \mathsf{stx}_{\mathsf{rcpt}}$

Let $\rho \xleftarrow{\$} \{0,1\}^{\ell_{\text{PRF}}}$

Let $(cm, r) \leftarrow \mathsf{Comm}(pk^{\text{COIN}} \,\|\, \rho \,\|\, v)$

Use $\mathcal{F}_{\text{FwEnc}}$ to encrypt $(pk^{\text{COIN}}, \tau, \rho, r, v)$ to pk_q^{ENC}, and denote the ciphertext $\mathsf{stx}^{\mathsf{real}}_{\mathsf{rcpt}}$.

end if

Let $cm_2 \leftarrow \widehat{\mathsf{Comm}}(ek)$.

Let $sn_1, sn_2 \xleftarrow{\$} \{0,1\}^{\ell_{\text{PRF}}}$

If either ρ_1 or ρ_2 were adversarially generated, and can be read from the transaction, use them directly to compute sn_1 or sn_2 instead.

Let root be the Merkle tree root of the current state of U_p.

Let $\mathbf{x} \leftarrow (\{cm_3, cm_4\}, \{sn_1, sn_2\}, \mathsf{root})$

Send $(\mathsf{Prove}, \mathbf{x}, U_p)$ to $\mathcal{A}$, denoting the response π.

Record the pair $(\mathbf{x}, \pi)$.

Let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (\{cm, cm_2\}, \{sn_1, sn_2\}, \mathsf{root}, \pi)$.

Broadcast $(\text{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, \mathsf{stx}^{\mathsf{real}}_{\mathsf{rcpt}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ as U_p.

end procedure

procedure SIMULATEGENERIC($\mathsf{tx}^{\mathsf{ideal}}$)
Let $\mathsf{tx}^{\mathsf{real}} = \text{GENERIC}$
for $\mathsf{stx} \in \mathsf{tx}^{\mathsf{ideal}}$ in order **do**
if $\mathsf{stx} = (pk_i^{\text{ENC}}, M)$ **then**
Send $(\mathsf{Encrypt}, \text{sid}, pk_i^{\text{ENC}}, \tau, M)$ to $\mathcal{F}_{\text{FwENC}}$ on behalf of U_p, and denote the response c.
Let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\perp, c)$.
else if $\mathsf{stx} = (\text{PUBLIC}, M)$ **then**
Let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\text{PUBLIC}, M)$.
else
Send $(\mathsf{Encrypt}, \text{sid}, \tau, |M|, U_p)$ to $\mathcal{A}$, and denote the response c.
Let $\mathsf{tx}^{\mathsf{real}} = \mathsf{tx}^{\mathsf{real}} \parallel (\perp, c)$.
end if
end for
Broadcast $\mathsf{tx}^{\mathsf{real}}$ to $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$ as U_p.
end procedure

procedure SIMULATESTAKING(U_p, τ, L)
if $U_p \notin L$ **then return**
Let $cm \leftarrow \widehat{\mathsf{Comm}}(ek); \rho, sn \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$.
If ρ was adversarially generated, and can be read from the transaction, use it directly to compute sn instead.
Send $(\mathsf{Encrypt}, \text{sid}, \tau, \ell_{\mathsf{Coin}}, U_p)$ to $\mathcal{A}$, and denote the response as c.
Let B, h, ptr, ep, sl, root, η_{ep}, and $\mathsf{stx}_{\mathsf{ref}}$ be defined as in an honest staking protocol execution by U_p.
Let $\mathbf{x} \leftarrow (\eta_{ep}, cm, sn, sl, \rho, h, ptr, \mathsf{root})$
Send $(\mathsf{prove}, \mathbf{x}, U_p)$ to $\mathcal{A}$, denoting the response π.
Record $(\mathbf{x}, \pi)$.
Let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (cm, sn, ep, sl, \rho, \pi, h, ptr)$
Let $\mathsf{tx} \leftarrow (\text{LEAD}, \vec{\mathsf{stx}}_{\mathsf{ref}}, \mathsf{stx}_{\mathsf{proof}})$
Broadcast tx to $\mathcal{F}_{\text{N-MC}}^{\mathsf{tx}}$, and (tx, B) to $\mathcal{F}_{\text{N-MC}}^{\mathsf{bc}}$ as U_p.
end procedure

procedure CORRUPT(U_p)
Corrupt U_p in the ideal protocol.
Send $(\mathsf{Read}, \text{sid})$ to $\mathcal{G}_{\text{PL}}$ on behalf of U_p. From the result, compute log, depending on the receiving transactions recorded.
Register U_p with $\mathcal{F}_{\text{FwENC}}$, and update the party's key for time τ.
Determine which leadership and transfer transactions were simulated as originating from U_p.
Disambiguate which coins won which leadership transactions.
for each unspent coin $\mathbf{c}$ belonging to U_p **do**
if $\mathbf{c}$ was created by an honest party **then**
Let $\rho_{\mathbf{c}} \xleftarrow{\$} \{0,1\}^{\kappa}$

Let τ be the time the coin creating transaction was submitted.
Let $r_{\mathbf{c}} \leftarrow \mathsf{Equiv}(ek, cm_{\mathbf{c}}, pk^{\text{COIN}}_{\mathbf{c}} \,\|\, \tau \,\|\, \rho_{\mathbf{c}} \,\|\, v_{\mathbf{c}})$.
else
Extract $(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$ by decrypting the corresponding ciphertext.
end if
if c is currently visible to U_p **then**
Add $(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$ to U_p's $\mathfrak{C}_{\mathsf{cnd}}$.
end if
Ensure $\mathfrak{C}_{\mathsf{free}}$, $\mathfrak{C}_{\mathsf{cnd}}$, and $\mathfrak{C}$ are consistent with a real execution, by checking which coins are confirmed, moving them to $\mathfrak{C}$, and erasing them from $\mathfrak{C}_{\mathsf{free}}$ and $\mathfrak{C}_{\mathsf{cnd}}$.
end for
end procedure

C UC Specification of Ouroboros Crypsinous

Protocol Ouroboros-Crypsinous$_k(U_p, \text{sid})$

Registration/Deregistration: *Intially, as in Ouroboros-Genesis, then call* Initialization-Crypsinous(U_p, sid, R)*, returning the result.*

Interacting with the Ledger ***(cf. Section 6.4):***

Upon receiving a ledger-specific input $I \in \{(\text{SUBMIT}, \ldots), (\text{READ}, \ldots), (\text{MAINTAIN-LEDGER}, \ldots)\}$ verify first that all resources are available. **If** not all resources are available, **then** ignore the input; **else** execute one of the following steps depending on the input I:

- **If** $I = (\text{SUBMIT}, \text{sid}, (\text{PUBLIC}, \text{TRANSFER}) \,\|\, \mathsf{tx})$ **then** set invoke the protocol $\mathsf{SubmitXfer}(\mathsf{tx}, \mathcal{C}_{\mathsf{loc}}, \mathsf{log})$.
- **Else if** $I = (\text{SUBMIT}, \text{sid}, \mathsf{tx})$ **then** set invoke the protocol $\mathsf{SubmitGeneric}(\text{sid})$.
- **If** $I = (\text{MAINTAIN-LEDGER}, \text{sid})$ **then** invoke protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, \mathfrak{C}, U_p, \text{sid}, k, s, R, f, \mathsf{log})$; if $\mathsf{LedgerMaintenance}$ halts **then** halt the protocol execution (all future input is ignored).
- **If** $I = (\text{GENERATE}, \text{sid}, \mathsf{tag})$ **then**
  - If $\mathsf{tag} = \text{COIN}$, query $\mathcal{G}_{\text{CLOCK}}$ for the current time τ. Then, sample $sk^{\text{COIN}}_{\tau} \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$, and let $sk^{\text{COIN}}_{i+1} \leftarrow \mathsf{PRF}^{\mathsf{evl}}_{sk_i}(1)$, for $i \in \{\tau+1, \ldots, \tau+R\}$. Let $\mathsf{root}^{\text{COIN}}_{sk}$ be the root of the Merkle tree over $sk^{\text{COIN}}_{\tau}, \ldots, sk^{\text{COIN}}_{\tau+R}$, and $pk^{\text{COIN}} \leftarrow \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}^{\text{COIN}}_{sk}}(\tau)$. Insert the Merkle tree into $\mathfrak{C}_{\mathsf{free}}$, and return pk^{COIN}.
  - If $\mathsf{tag} = \text{ID}$, and this is the first query for ID, send $(\mathsf{KeyGen}, \text{sid})$ to $\mathcal{F}_{\text{FWENC}}$. Denote the response by pk^{ENC}. Record pk^{ENC}, then return it.
  - Otherwise, return a uniformly sampled value from $\{0,1\}^{\kappa}$.
- **If** $I = (\text{READ}, \text{sid})$ **then** invoke protocol $\mathsf{ReadState}(k, \mathcal{C}_{\mathsf{loc}}, U_p, \text{sid}, R, f, \mathsf{log})$.

Handling external (protocol-unrelated) calls: *as in Ouroboros-Genesis.*

C.1 Party Initialization

Protocol Initialization-Crypsinous(U_p, sid, R)

The following steps are executed in an (MAINTAIN-LEDGER, sid)-interruptible manner:

Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$.
if $\tau = 0$ **then** execute the following steps in an (MAINTAIN-LEDGER, sid)-interruptible manner:
    Send $(\mathsf{claim}, sid, U_p)$ to $\mathcal{F}_{\text{INIT}}$ to claim stake from the genesis block, receiving the response $(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$, and sk^{COIN}.
    Let $\mathfrak{C} \leftarrow \{(pk^{\text{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}$, and $\mathfrak{C}_{\mathsf{free}} \leftarrow \{sk^{\text{COIN}}\}$
    Send (CLOCK-UPDATE, sid_C) to $\mathcal{G}_{\text{CLOCK}}$.
    Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$, and give up the activation.
    while $\tau = 0$ **do**
        Use the clock to update τ, ep, and sl and give up the activation.
    end while
else
    Send $(\mathsf{genblock\_req}, sid, U_p)$ to $\mathcal{F}_{\text{INIT}}$. If $\mathcal{F}_{\text{INIT}}$ signals an error then halt. Otherwise, receive from $\mathcal{F}_{\text{INIT}}$ the response $(\mathsf{genblock}, sid, \mathbf{G} = (\mathbb{C}_1, \eta_1))$.
    Set $\mathcal{C}_{\mathsf{loc}} \leftarrow (\mathbf{G})$.
    Send (NEW-PARTY, sid, U_p) to $\mathcal{F}^{\mathsf{new}}_{\text{N-MC}}$.
    Return $pk^{\text{COIN}}_{\mathbf{c}}$.
end if
Set $t_{\text{on}} \leftarrow \tau$.
Return $\varnothing$.

GLOBAL VARIABLES: The protocol stores the list of variables $pk^{\text{ENC}}, \tau, ep, sl, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C}, \mathfrak{C}_{\mathsf{free}}, t_{\text{on}}$ to make each of them accessible by all protocol parts.

C.2 The Staking Procedure

Protocol StakingProcedure$(k, U_p, ep, sl, \mathsf{buffer}, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C})$

The following steps are executed in an (MAINTAIN-LEDGER, sid)-interruptible manner:

for $(pk^{\text{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}) \in \mathfrak{C}$ **do**
    if c is not eligible for leadership **then continue**
    Send $(\mathsf{eval}, \text{sid}_{RO}, \text{NONCE} \,\|\, \eta_{ep} \,\|\, sl))$ to $\mathcal{G}_{\text{RO}}$, and denote the response μ_ρ.
    Send $(\mathsf{eval}, \text{sid}_{RO}, \text{LEAD} \,\|\, \eta_{ep} \,\|\, sl))$ to $\mathcal{G}_{\text{RO}}$, and denote the response μ_y.
    Lookup $sk^{\text{COIN}}_{\mathbf{c},\tau}$, $\mathsf{root}_{\mathbf{c}}$, and $\tau_{\mathbf{c}}$ in $\mathfrak{C}_{\mathsf{free}}$ corresponding to $pk^{\text{COIN}}_{\mathbf{c}}$.
    Let $\rho \leftarrow \mu_\rho^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}}} \,\|\, \rho_{\mathbf{c}}}; y \leftarrow \mu_y^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}}} \,\|\, \rho_{\mathbf{c}}}$
    if $y < \mathsf{ord}(G)\phi_f(v_{\mathbf{c}})$ **then**
        repeat
            Parse buffer' as sequence $(\mathsf{tx}_1, \ldots, \mathsf{tx}_n)$
            for $i = 1$ to n **do**
                if $\mathsf{ValidTx}_{\mathsf{OP}}(\mathsf{tx}_i, \vec{\mathsf{st}} || \mathsf{st}) = 1$ **then**
                    $\vec{N} \leftarrow \vec{N} || \mathsf{tx}_i$
                    Remove tx from buffer'

14: Set $\mathsf{st} \leftarrow \mathsf{blockify}_{\mathsf{OP}}(\vec{N})$
end if
end for
until $\vec{N}$ does not increase anymore
15: Set $ptr \leftarrow H(\mathsf{head}(\mathcal{C}_{\mathsf{loc}}))$; $h \leftarrow H(\mathsf{st})$
16: Set $\rho_{\mathbf{c}'} \leftarrow \mathsf{PRF}^{\mathsf{evl}}_{\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}}}}(\rho_{\mathbf{c}})$; $sn_{\mathbf{c}} \leftarrow \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\mathrm{COIN}}_{\mathbf{c}}}}(\rho_{\mathbf{c}})$
17: Set $(cm_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{Comm}(pk^{\mathrm{COIN}} \,\|\, v_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}'})$.
18: Let $\vec{\mathsf{stx}}_{\mathsf{ref}}$ be, in order, the list of leadership transactions made by U_p not in $\mathcal{C}_{\mathsf{loc}}$.
19: Let root be the root of the Merkle tree $\mathbb{C}^{\mathsf{lead}}$ in $\mathcal{C}_{\mathsf{loc}}$, after applying all transactions in $\vec{\mathsf{stx}}_{\mathsf{ref}}$. Let path be the path to $cm_{\mathbf{c}}$ in the same Merkle tree.
20: Let $\mathsf{path}_{\mathbf{c}}$ be the Merkle path to $sk^{\mathrm{COIN}}_{\mathbf{c},\tau}$ in the secret-key Merkle tree.
21: Let $\mathbf{x} = (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, \eta_{ep}, sl, \rho, h, ptr, \mu_\rho, \mu_y, \mathsf{root})$.
22: Let $\mathbf{w} = (\mathsf{path}, \mathsf{root}_{sk^{\mathrm{COIN}}}, \mathsf{path}_{\mathbf{c}}, \tau_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}, r_{\mathbf{c}'})$.
23: Send $(\mathsf{prove}, \mathrm{sid}, \mathbf{x}, \mathbf{w})$ to $\mathcal{F}_{\mathrm{NIZK}}^{\mathcal{L}_{\mathrm{LEAD}}}$, and denote the response π.
24: Let $\mathsf{tx}_{\mathsf{lead}} = (\mathrm{LEAD}, \vec{\mathsf{stx}}_{\mathsf{ref}}, (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, ep, sl, \rho, h, ptr, \pi))$.
25: Set $B \leftarrow (\mathsf{tx}_{\mathsf{lead}}, \mathsf{st})$; $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \,\|\, B$.
26: Update $\mathbf{c}$: $\mathfrak{C} \leftarrow (\mathfrak{C} \setminus \{(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}) \cup \{(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}'}, r_{\mathbf{c}'}, v_{\mathbf{c}})\}$
27: Send $(\mathrm{MULTICAST}, \mathrm{sid}, \mathsf{tx}_{\mathsf{lead}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
28: Send $(\mathrm{MULTICAST}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
29: **break**
end if
end for
30: **while** A $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ has not been received during the current round **do**
Give up activation. Upon next activation of this procedure, proceed from here.

end while

C.3 The Ledger Maintenance Procedure

Protocol LedgerMaintenance(...)

The following steps are executed in an $(\text{MAINTAIN-LEDGER}, \mathrm{sid})$-interruptible manner:

1: Execute **FetchInformation** to receive the newest messages for this round; denote the output by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$, $(\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$, and read the flag WELCOME.
2: **if** WELCOME $= 1$ **then**
3: Send $(\mathrm{MULTICAST}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$.
4: **for** each $\mathsf{tx} \in \mathsf{buffer}$ **do**
Send $(\mathrm{MULTICAST}, \mathrm{sid}, \mathsf{tx})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.
end for
end if
5: **for** transaction $\mathsf{tx} \in (\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$ **do**
6: **if** tx is a transfer transaction **then**

Attempt to decrypt each new ciphertext c by sending $(\mathsf{Decrypt}, \mathrm{sid}, c)$ to $\mathcal{F}_{\text{FWENC}}$. Receive the response m.
if $m = (pk^{\text{COIN}}, \tau, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}) \wedge cm_{\mathbf{c}} \in \mathsf{tx}$ **then**
if $\nexists sk_{\tau}^{\text{COIN}} \in \mathfrak{C}_{\mathsf{free}}$ corresponding to pk^{COIN}
then continue
Let $\mathfrak{C}_{\mathsf{cnd}} \leftarrow \mathfrak{C}_{\mathsf{cnd}} \cup \{(pk^{\text{COIN}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}$.
Let $\mathsf{log} \leftarrow \mathsf{log} \,\|\, (\mathsf{tx}, \text{RECEIVE}, (pk^{\text{COIN}}, v_{\mathbf{c}}))$.
end if
else if tx is a generic transaction **then**
Attempt to decrypt each subtransaction ciphertext c by sending $(\mathsf{Decrypt}, \mathrm{sid}, c)$ to $\mathcal{F}_{\text{FWENC}}$. Receive the response m.
if $m \neq \perp$ **then** $\mathsf{log} \leftarrow \mathsf{log} \,\|\, (\text{PLAINTEXT}, c, m)$
end if
end for
for coin $(sk_{\mathbf{c}}^{\text{COIN}}, \tau_{\mathbf{c}}) \in \mathfrak{C}_{\mathsf{free}}$ **do**
if $\exists$ a coin for pk^{COIN} in $\mathfrak{C}_{\mathsf{cnd}}$ whose transaction $\in \mathcal{C}_{\mathsf{loc}}^{\lceil k}$ **then**
Move such candidates to $\mathfrak{C}$.
end if
Erase $sk_{\mathbf{c},\tau}^{\text{COIN}}$ (and for any time before τ) from $\mathfrak{C}_{\mathsf{free}}$.
end for
Use the clock to update $\tau, ep \leftarrow \lceil \tau / R \rceil$, and $sl \leftarrow \tau$.
Set $\mathsf{buffer} \leftarrow \mathsf{buffer} || (\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$, $t_{\text{on}} \leftarrow \tau$, $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \text{and } \mathcal{C}_M\}$
Invoke Protocol $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$.
Update $\mathcal{F}_{\text{FWENC}}$ as many times as necessary for its time to be a least $\tau - k$.
if $t_{\text{work}} < \tau$ **then**
Invoke protocol $\mathsf{StakingProcedure}(k, U_p, ep, sl, \mathsf{buffer}, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C})$ (in a (MAINTAIN-LEDGER, sid)-interruptible manner).
Set $t_{\text{work}} \leftarrow \tau$ and send $(\text{CLOCK-UPDATE}, \mathrm{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$.
end if

C.4 Submitting Transfer Transactions

Protocol $\mathsf{SubmitXfer}(\mathsf{tx}_{\mathsf{xfer}}, \mathcal{C}_{\mathsf{loc}}, \mathfrak{C}, \mathsf{log})$

Let $((pk_r^{\text{ENC}}, (pk_{\mathbf{c}_4}^{\text{COIN}}, v_4)), (pk_s^{\text{ENC}}, (pk_{\mathbf{c}_1}^{\text{COIN}}, v_1), (pk_{\mathbf{c}_2}^{\text{COIN}}, v_2), (pk_{\mathbf{c}_3}^{\text{COIN}}, v_3))) \leftarrow \mathsf{tx}_{\mathsf{xfer}}$.
if $pk_s^{\text{ENC}} \neq pk^{\text{ENC}}$ **or** $v_1 + v_2 \neq v_3 + v_4$ **or** $pk_{\mathbf{c}_3}^{\text{COIN}} \notin \mathfrak{C}_{\mathsf{free}}$ **then return**
Check $\mathfrak{C}$ for the first coin received at ID $pk_{\mathbf{c}_1}^{\text{COIN}}$, $pk_{\mathbf{c}_2}^{\text{COIN}}$. Ensure they have value v_1 and v_2 respectively, and denote their (potentially evolved) variant as $\mathbf{c}_1$ and $\mathbf{c}_2$. Ensure these are in $\mathcal{C}_{\mathsf{loc}}^{\lceil k}$.
As a special case, allow $pk_{\mathbf{c}_2}^{\text{COIN}} = \perp$, and $v_2 = 0$.
if these do not exist, or are not in $\mathfrak{C}$ **then return**
Retrieve the corresponding $(pk_{\mathbf{c}_i}^{\text{COIN}}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, v_{\mathbf{c}_i})$ from $\mathfrak{C}$.
Lookup $sk_{\mathbf{c}_i}^{\text{COIN}}$ in $\mathfrak{C}_{\mathsf{free}}$ for $i \in \{1, 2\}$, corresponding to $pk_{\mathbf{c}_i}^{\text{COIN}}$.
if $pk_{\mathbf{c}_2}^{\text{COIN}} = \perp$, and $v_2 = 0$ **then** $sn_{\mathbf{c}_2} \leftarrow \mathsf{PRF}^{\mathsf{zdrv}}_{\mathsf{root}sk_{\mathbf{c}_1}^{\text{COIN}}}(\rho_{\mathbf{c}_1})$ and all other values for $\mathbf{c}_2$ are zeroed.
Sample $\rho_{\mathbf{c}_3}, \rho_{\mathbf{c}_4} \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{PRF}}}$.

10: Commit $(cm_{\mathbf{c}_i}, r_{\mathbf{c}_i}) \leftarrow \mathsf{Comm}(pk^{\text{COIN}}_{\mathbf{c}_i} \,\|\, v_i \,\|\, \rho_{\mathbf{c}_i})$, for $i \in \{3, 4\}$.
11: Let $sn_{\mathbf{c}_1} \leftarrow \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$; $sn_{\mathbf{c}_2} \leftarrow \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_2}}}(\rho_{\mathbf{c}_2})$
12: Extract the state $\vec{\mathsf{st}}$ from $\mathcal{C}_{\mathsf{loc}}$.
13: Let root be the transfer Merkle tree root in $\mathcal{C}^{\lceil k}_{\mathsf{loc}}$.
14: Let path_1 and path_2 be paths to $cm_{\mathbf{c}_1}$, and $cm_{\mathbf{c}_2}$ in the same Merkle tree, respectively, or, if $pk^{\text{COIN}}_{\mathbf{c}_2} = \perp$, and $v_2 = 0$, let path_2 be empty.
15: **if** either are not found in the Merkle tree **then return**
16: Let $\mathbf{x} \leftarrow (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \tau, \mathsf{root})$.
17: Let $\mathbf{w} \leftarrow (\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{path}_{sk^{\text{COIN}}_{\tau,\mathbf{c}_1}}, \mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, \mathsf{path}_{sk^{\text{COIN}}_{\tau,\mathbf{c}_2}}, pk^{\text{COIN}}_{\mathbf{c}_3}, pk^{\text{COIN}}_{\mathbf{c}_4}, (\rho_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v_1, \mathsf{path}_1), (\rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, v_2, \mathsf{path}_2), (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_3), (\rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_4))$.
18: Send $(\mathsf{prove}, \text{sid}, \mathbf{x}, \mathbf{w})$ to $\mathcal{F}^{\mathcal{L}_{\text{XFER}}}_{\text{NIZK}}$, and receive π.
19: Send $(\mathsf{encrypt}, \text{sid}, \tau, pk^{\text{ENC}}_r, (pk^{\text{COIN}}_{\mathbf{c}_4}, \tau, \rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_{\mathbf{c}_4}))$ to $\mathcal{F}_{\text{FWENC}}$, and receive c_{rcpt}.
20: Let $\mathsf{stx}_{\mathsf{proof}} \leftarrow (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \mathsf{root}, \pi)$.
21: Let $\mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}} \leftarrow (\text{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, c_{\mathsf{rcpt}})$.
22: Let $\mathsf{log} \leftarrow \mathsf{log} \setminus \{(pk^{\text{COIN}}_{\mathbf{c}_1}, v_{\mathbf{c}_1}), (pk^{\text{COIN}}_{\mathbf{c}_2}, v_{\mathbf{c}_2})\}$.
23: Erase $\mathbf{c}_{1,2}$: $\mathfrak{C} \leftarrow \mathfrak{C} \setminus \{(pk^{\text{COIN}}_{\mathbf{c}_i}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, v_{\mathbf{c}_i}) \mid i \in \{1, 2\}\}$.
24: Record $\mathbf{c}_3$: $\mathfrak{C}_{\mathsf{cnd}} \leftarrow \mathfrak{C}_{\mathsf{cnd}} \cup \{(pk^{\text{COIN}}_{\mathbf{c}_3}, \rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_{\mathbf{c}_3})\}$
25: Send $(\text{MULTICAST}, \text{sid}, \mathsf{tx}^{\mathsf{real}}_{\mathsf{xfer}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.

C.5 Submitting Generic Transactions

Protocol SubmitGeneric(tx)

1: Let $\mathsf{tx}^{\mathsf{real}} = \text{GENERIC}$.
2: **for** each $\mathsf{stx} \in \mathsf{tx}$ in order **do**
3: **if** $\mathsf{stx} = (\top, M)$ **then**
4: Let $\mathsf{tx}^{\mathsf{real}} \leftarrow \mathsf{tx}^{\mathsf{real}} \,\|\, \mathsf{stx}$.
5: **else if** $\mathsf{stx} = (pk^{\text{ENC}}_r, M)$ **then**
6: Send $(\mathsf{Encrypt}, \text{sid}, \tau, pk^{\text{ENC}}_r, M)$ to $\mathcal{F}_{\text{FWENC}}$, and denote the response c.
7: Let $\mathsf{tx}^{\mathsf{real}} \leftarrow \mathsf{tx}^{\mathsf{real}} \,\|\, (\perp, c)$.
end if
end for
8: Send $(\text{MULTICAST}, \text{sid}, \mathsf{tx}^{\mathsf{real}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$.

C.6 Reading the Ledger State

Protocol ReadState$(k, \mathcal{C}_{\mathsf{loc}}, U_p, \text{sid}, R, f)$

1: Execute **FetchInformation** to receive the newest messages for this round; denote the output chains by $(\mathcal{C}_1, \ldots, \mathcal{C}_M)$ (the list of transactions $(\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$ and the flag WELCOME can be ignored).
2: Invoke protocol **UpdateTime**(k, U_p, R, f) and denote the output as $\tau, ep, sl, \mathbb{S}_{ep}, \alpha^{ep}_p, T^{ep}_p$, and η_{ep}.
3: Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$.
4: Set $t_{\text{on}} \leftarrow \tau$, $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$.

Invoke Protocol $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$.
Extract the state $\vec{\mathsf{st}}$ from the current local chain $\mathcal{C}_{\mathsf{loc}}$.
Let $\vec{\mathsf{st}}^{\mathsf{ideal}} = \epsilon$.
for each $\mathsf{tx} \in \vec{\mathsf{st}}^{\lceil k}$ in order **do**
    if $\mathsf{tx} = (\text{TRANSFER}, \mathsf{stx}_{\mathsf{proof}}, \mathsf{stx}_{\mathsf{rcpt}})$ **then**
        Let $\mathsf{stx}_{\mathsf{chng}} \leftarrow \mathsf{stx}_{\mathsf{rcpt}} \leftarrow \bot$.
        if $\exists v : (\mathsf{tx}, \text{RECEIVE}, v) \in \mathsf{log}$ **then**
            Let $\mathsf{stx}_{\mathsf{rcpt}} \leftarrow (U_p, v)$.
        end if
        Let $\vec{\mathsf{st}}^{\mathsf{ideal}} \leftarrow \vec{\mathsf{st}}^{\mathsf{ideal}} \parallel ((\top, \text{TRANSFER}), \mathsf{stx}_{\mathsf{rcpt}}, \mathsf{stx}_{\mathsf{chng}})$.
    else if $\mathsf{tx} = (\text{GENERIC}, \mathsf{stx}_1, \ldots, \mathsf{stx}_n)$ **then**
        Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \epsilon$.
        for each subtransactions $\mathsf{stx} \in \mathsf{tx}$ in order **do**
            if $\mathsf{stx} = (\top, m)$ **then**
                Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \mathsf{tx}^{\mathsf{ideal}} \parallel (\top, m)$
            else if $\mathsf{stx} = (\bot, c)$ **then**
                if $\exists m : (\text{PLAINTEXT}, c, m) \in \mathsf{log}$ **then**
                    Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \mathsf{tx}^{\mathsf{ideal}} \parallel (U_p, m)$
                else
                    Let $\mathsf{tx}^{\mathsf{ideal}} \leftarrow \mathsf{tx}^{\mathsf{ideal}} \parallel \bot$
                end if
            end if
        end for
    end if
    Let $\vec{\mathsf{st}}^{\mathsf{ideal}} \leftarrow \vec{\mathsf{st}}^{\mathsf{ideal}} \parallel (\mathsf{tx}^{\mathsf{ideal}})$.
end for
Output $(\text{READ}, \mathrm{sid}, \vec{\mathsf{st}}^{\mathsf{ideal}})$.

D NIZK Statements

Recall that we use two NIZK statements: LEAD, and XFER. XFER is very close to the statement used in Zerocash [4], while LEAD is a mixture between a Zerocash proof, and an Ouroboros Praos [13] leadership proof. We define the statements by their corresponding NP languages:
A tuple $(\mathbf{x}, \mathbf{w}) \in \mathcal{L}_{\text{LEAD}}$ iff all of the following hold:

- $\mathbf{x} = (cm_{\mathbf{c}_2}, sn_{\mathbf{c}_1}, \eta, sl, \rho, h, ptr, \mu_\rho, \mu_y, \mathsf{root})$
- $\mathbf{w} = (\mathsf{path}, \mathsf{root}_{sk^{\text{COIN}}}, \mathsf{path}_{sk^{\text{COIN}}}, \tau_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}_1}, v, r_{\mathbf{c}_2})$
- $pk^{\text{COIN}} = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}_{sk^{\text{COIN}}}}(\tau_{\mathbf{c}})$
- $\rho_{\mathbf{c}_2} = \mathsf{PRF}^{\mathsf{evl}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$
- $\forall i \in \{1, 2\} : \mathsf{DeComm}(cm_{\mathbf{c}_i}, pk^{\text{COIN}} \parallel v \parallel \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \top$
- path is a valid Merkle tree path to $cm_{\mathbf{c}_1}$ in a tree with root root.
- $\mathsf{path}_{sk^{\text{COIN}}}$ is a valid path to a leaf at position $sl - \tau_{\mathbf{c}}$ in a tree with root $\mathsf{root}_{sk^{\text{COIN}}}$.

- $sn_{\mathbf{c}_1} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}}}(\rho_{\mathbf{c}_1})$
- $y = \mu_y^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}} \,\|\, \rho_{\mathbf{c}}}; \rho = \mu_\rho^{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}} \,\|\, \rho_{\mathbf{c}}}$
- $y < \mathsf{ord}(G)\phi_f(v)$

Note that $\mathbf{x}$ of LEAD contains values sl, h, ptr that seemingly nothing is proven about. As a UC proof system is non-malleable, this makes them part of the signature of knowledge message.
A tuple $(\mathbf{x}, \mathbf{w}) \in \mathcal{L}_{\text{XFER}}$ iff all of the following hold:

- $\mathbf{x} = (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \tau, \mathsf{root})$
- $\mathbf{w} = (\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_1}}, \mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, \mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_2}}, pk^{\text{COIN}}_{\mathbf{c}_3}, pk^{\text{COIN}}_{\mathbf{c}_4}, (\rho_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v_1, \mathsf{path}_1), (\rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, v_2, \mathsf{path}_2), (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_3), (\rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_4))$
- $\forall i \in \{1, 2\} : pk^{\text{COIN}}_{\mathbf{c}_i} = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}}(1)$ (or, if $v_2 = 0$, this check may be skipped for $i = 2$)
- $\forall i \in \{1, ..., 4\} : \mathsf{DeComm}(cm_{\mathbf{c}_i}, pk^{\text{COIN}}_{\mathbf{c}_i} \,\|\, v_i \,\|\, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \top$ (or, 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 $cm_{\mathbf{c}_1}$ in a tree with root root.
- path_2 is a valid path to $cm_{\mathbf{c}_2}$ in a tree with root root, **or** $v_2 = 0$ and $sn_{\mathbf{c}_2} = \mathsf{PRF}^{\mathsf{zdrv}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_1}}}(\rho_{\mathbf{c}_1})$.
- $\mathsf{path}_{sk^{\text{COIN}}_{\mathbf{c}_i}}$ is a valid path to a leaf at position τ in $\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}$, for $i \in \{1, 2\}$.
- $\forall i \in \{1, 2\} : sn_{\mathbf{c}_i} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}}(\rho_{\mathbf{c}_i})$ (or, if $v_2 = 0$, this check may be skipped for $i = 2$)

E Protocol Assumptions Encoded as a Wrapper

This section includes complementary material for the main body. We sketch below the wrapper functionality that is applied to the hybrid functionalities used by Ouroboros-Crypsinous. It is a slight adaptation of the same wrapper used in Ouroboros Genesis [1], with the modification that calls to $\mathcal{F}_{\text{NIZK}}$ are restricted, not $\mathcal{F}_{\text{VRF}}$. In a nutshell, the wrapper observes the advancement of the entire system and checks whether the proportional stake of alert parties, of corrupted or de-synchronized parties, and of stalled parties are within the allowed range specified as required by our main theorems.

Functionality $\mathcal{W}^{PoS}_{\text{OC}}(\cdot)$

The wrapper functionality is parameterized by the bounds α,β on the alert and participating stake ratio, as defined in Ouroboros Genesis [1], respectively, the network delay and a value $\varepsilon > 0$ (the parameter that describes the gap between the honest and adversarial stake). The wrapper is assumed to be registered with the global clock $\mathcal{G}_{\text{CLOCK}}$ and is aware of sets of registered parties, and the set of corrupted parties.
We note that 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 does, however, observe all network traffic, as well as all NIZK witnesses. From this, it can reconstruct exactly which party transfers stake when, and to

whom. We will not describe this extraction in full detail, but note that effectively, as the wrapper is around all privacy-preserving functionalities, it has a clear view of the state. We can therefore make assertions about the stake distribution despite the addition of privacy.

General:

- Upon receiving any request I from any party U_p or from $\mathcal{A}$ (possibly on behalf of a party U_p which is corrupted) to a wrapped hybrid functionality, record the request I together with its source and the current time.
- The wrapper keeps track of the active parties and their relative share to the stake distribution.

Restrictions on obtaining NIZK proofs:

- Upon receiving $(\mathsf{Prove}, \text{sid}, \cdot, \cdot)$ to $\mathcal{F}_{\text{NIZK}}$ from $\mathcal{A}$ on behalf of a party U_p which is corrupted or registered but de-synchronized do the following:
  1. If the fraction of alert stake relative to all active stake in this round τ so far does not satisfy the honest majority, as in Ouroboros Genesis [1] then ignore the request.
  2. Otherwise, forward the request to $\mathcal{F}_{\text{NIZK}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{NIZK}}$ returns.
- Upon receiving $(\mathsf{Prove}, \text{sid}, \cdot, \cdot)$ to $\mathcal{F}_{\text{NIZK}}$ from an alert party U_p do the following:
  1. Forward the request to $\mathcal{F}_{\text{NIZK}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{NIZK}}$ returns.
  2. If the minimal fraction (in stake) of participation (of alert parties and in total) as required by Ouroboros Genesis [1] is reached in round τ, send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ to release the clock for this round.
- Any other request is relayed to the underlying functionality (and recorded by the wrapper) and the corresponding output is given to the destination specified by the underlying functionality.

F Construction NIZKs via SNARKs

We will utilise, and prove the UC-security of the lifted SNARK system presented in [21]. Specifically, we will focus on the version presented allowing for proof-malleability, i.e. allowing the adversary to re-prove statements with a different proof object. For our purposes, this weak version is sufficient. We note that asimulation secure NIZK, as presented in [21] is a tuple $(\mathcal{K}, \mathcal{P}, \mathcal{V}, \widehat{\mathcal{K}}, \widehat{\mathcal{P}})$. This fairly directly corresponds to a UC protocol for $\mathcal{F}_{\text{NIZK}}$, in the $\mathcal{F}_{\text{CRS}}^{D}$-hybrid world, where D is the output distribution of $\mathcal{K}(1^\lambda, \mathcal{L})$, and proving and verification are implemented as expected with the provided algorithms. We will refer to this protocol as NIZK-SNARK. We are also guaranteed the existance of an algorithm $\mathcal{E}$ which can extract proofs, and although that may not be well-known, we note that both the environment and simulator may be assumed to have access to $\mathcal{E}$. For a security parameter κ, we are guaranteed the properties in Figures 1-4 hold. Where we use $\approx$, we mean that the statistical distance between the distributions is less than some negligible function μ of λ.

Determinism We will assume that $\mathcal{V}$ and $\mathcal{E}$ are *deterministic* algorithms. If we are given a non-deterministic verification algorithm $\mathcal{V}'$, $\mathcal{E}'$, we note that we can construct deterministic algorithms

$$\forall \mathcal{L}, (x,w) \in \mathcal{L}, \mathsf{crs} \in \mathcal{K}(1^\kappa, \mathcal{L}), \pi \in \mathcal{P}(\mathsf{crs}, x, w) : \\ \Pr\left[\mathcal{V}(\mathsf{crs}, x, \pi) = \top\right] = 1$$

Figure 1: Perfect completeness.

$$\forall \mathcal{L}, \mathcal{A} : \Pr\left[\mathsf{crs} \xleftarrow{\$} \mathcal{K}(1^\kappa, \mathcal{L}); \mathcal{A}^{\mathcal{P}(\mathsf{crs},\cdot,\cdot)}(\mathsf{crs}) = \top\right] \\ \approx \Pr\left[(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \xleftarrow{\$} \widehat{\mathcal{K}}(1^\kappa, \mathcal{L}); \mathcal{A}^{\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}},\tau,\cdot,\cdot)}(\widehat{\mathsf{crs}}) = \top\right]$$

Where $\widehat{\mathcal{P}}_1$ acts as $\widehat{\mathcal{P}}$, but aborts if it is asked to simulate a proof for $(x,w) \notin \mathcal{L}$.

Figure 2: Computational zero-knowledge.

$$\forall \mathcal{L}, \mathcal{A} : \Pr\left[\begin{array}{c} \mathsf{crs} \xleftarrow{\$} \mathcal{K}(1^\kappa, \mathcal{L}); \\ (x, \pi) \xleftarrow{\$} \mathcal{A}(\mathsf{crs}); \\ (\mathcal{V}(\mathsf{crs}, x, \pi) = \top) \wedge (\nexists w : (x,w) \in \mathcal{L}) \end{array}\right] \approx 0$$

Figure 3: Computational soundness.

$$\forall \mathcal{L}, \mathcal{A} : \Pr\left[\begin{array}{c} (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \xleftarrow{\$} \widehat{\mathcal{K}}(1^\kappa, \mathcal{L}); \\ (x, \pi) \xleftarrow{\$} \mathcal{A}^{\widehat{\mathcal{P}}(\widehat{\mathsf{crs}},\tau,\cdot,\cdot)}(\widehat{\mathsf{crs}}, \mathsf{ek}); \\ w \xleftarrow{\$} \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi); \\ x \notin Q \wedge (x,w) \notin \mathcal{L} \wedge \mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top \end{array}\right] \approx 0$$

Where Q is set set of statements x that $\mathcal{A}$ queried using oracle access to $\widehat{\mathcal{P}}$.

Figure 4: Simulation sound extractability.

$\mathcal{V}$, and $\mathcal{E}$ by fixing the random tape. $\mathcal{V}$ necessarily satisfies completeness and zero-knowledge, and with overwhelming probability will still satisfy soundness and simulation sound extractability. Likewise, $\mathcal{E}$ necessarily satisfies completeness, zero-knowledge, and soundness, and with overwhelming probability satisfies simulation sound extractability. If α is the fraction of random tapes for which $\mathcal{V}$ or $\mathcal{E}$ can break some property of the NIZK with a non-negligible probability of at least β, then since the sampling of the random tape and the other inputs in the security games is independant, $\mathcal{V}'$ or $\mathcal{E}'$ respectively has a probability of at least $\alpha\beta$ of breaking the same property. As β is non-negligible, and $\alpha\beta$ negligible, by assumption, α must be negligible. Therefore, with overwhelming probability, all properties hold for $\mathcal{V}$.

F.1 Proof of UC-Emulation

Functionality $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$

The protocol $\mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}}$ slightly idealises $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$, by ensuring that previously proved statements always verify. It is built in the $\mathcal{F}^{D}_{\mathrm{CRS}}$ hybrid model, and keeps crs, and Π as

variables. Π is initialized to $\varnothing$.

Initialisation *On first activation:*
send $(\mathsf{query}, \mathrm{sid})$ **to** $\mathcal{F}_{\mathrm{CRS}}^{D}$ and **receive the reply** $(\mathsf{query}, \mathrm{sid}, \mathsf{crs})$

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:*
if $(x, w) \in \mathcal{L}$ **then**
let $\pi \leftarrow \mathcal{P}(\mathsf{crs}, x, w); \Pi \leftarrow \Pi \cup \{(x, \pi)\}$
return $(\mathsf{proof}, \mathrm{sid}, x, \pi)$
else
return $(\mathsf{proof}, \mathrm{sid}, x, \bot)$
end if

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*
if $(x, \pi) \in \Pi$ **then**
return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$
else if $\mathcal{V}(\mathsf{crs}, x, \pi) = \top$ **then**
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$
return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$
else
return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \bot)$
end if

Lemma 1. $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ *perfectly UC-emulates* $\mathcal{F}_{\mathrm{NIZK}}^{\prime\mathcal{L}}$ *in the* $\mathcal{F}_{\mathrm{CRS}}^{D}$*-hybrid model.*

Proof. We note that (prove) queries, and CRS have identical output in $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ and $\mathcal{F}_{\mathrm{NIZK}}^{\prime\mathcal{L}}$. For previously unseen statement/proof pairs, verification queries are also identical. For previously seen statement/proof pairs (x, π), $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ would output $\mathcal{V}(\mathsf{crs}, x, \pi)$, while $\mathcal{F}_{\mathrm{NIZK}}^{\prime\mathcal{L}}$ outputs $\top$. There are two types of "previously seen" statement/proof pairs. π may be generated by $\mathcal{P}(\mathsf{crs}, x, w)$ for some w where $(x, w) \in \mathcal{L}$. In this case, by Figure 1, we know that $\mathcal{V}(\mathsf{crs}, x, \pi) = \top$, therefore the outputs are identical. Alternatively, the statement/proof pair was previously seen in verification, and $\mathcal{V}(\mathsf{crs}, x, \pi) = \top$. Since $\mathcal{V}$ is deterministic, it will return $\top$, the same as $\mathcal{F}_{\mathrm{NIZK}}^{\prime\mathcal{L}}$. $\square$

Functionality $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\mathcal{L}}$

The protocol $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\mathcal{L}}$ is a further idealisation of $\mathcal{F}_{\mathrm{NIZK}}^{\prime\mathcal{L}}$, which utilises simulated proofs instead of real proofs. It is built in the $\mathcal{F}_{\mathrm{CRS}}^{D'}$ hybrid model, where D' is the output distribution of $\widehat{\mathcal{K}}(1^{\kappa}, \mathcal{L})$. It keeps $\widehat{\mathsf{crs}}$, τ, and Π as variables, where Π is initialized to $\varnothing$.

Initialisation *On first activation:*
send $(\mathsf{query}, \mathrm{sid})$ **to** $\mathcal{F}_{\mathrm{CRS}}^{D'}$ and **receive the reply** $(\mathsf{query}, \mathrm{sid}, (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}))$

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:*
if $(x, w) \in \mathcal{L}$ **then**
let $\pi \leftarrow \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x); \Pi \leftarrow \Pi \cup \{(x, \pi)\}$
return $(\mathsf{proof}, \mathrm{sid}, x, \pi)$
else

return $(\mathsf{proof}, \mathrm{sid}, x, \bot)$
end if

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*
if $(x, \pi) \in \Pi$ **then**
return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$
else if $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$ **then**
let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$
return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$
else
return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \bot)$
end if

Simulator $\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}$

let $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \xleftarrow{\$} \widehat{\mathcal{K}}(1^\lambda, \mathcal{L})$
program $\mathcal{F}^D_{\mathrm{CRS}}$ to return $\widehat{\mathsf{crs}}$
simulate $\mathcal{A}$

Lemma 2. *$\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ in the $\mathcal{F}^D_{\mathrm{CRS}}$-hybrid model UC-emulates $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ in the $\mathcal{F}^{D'}_{\mathrm{CRS}}$-hybrid model.*

Proof. We note that it is sufficient to prove that there exists a simulator for the dummy adversary. We will use $\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}$ for this purpose. We note that the difference between $\mathcal{F}^D_{\mathrm{CRS}}$ and $\mathcal{F}^{D'}_{\mathrm{CRS}}$ is precicely the difference in the sampled key generation parameters of Figure 2. Further, we note that the environment can query $\mathcal{F}^D_{\mathrm{CRS}}$ through the dummy adversary, which corresponds precicely to the input parameter of crs or $\widehat{\mathsf{crs}}$ in Figure 2.

Aside from extracting the CRS from the adversary, the environment can only make honest interactions with $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$. We note that for verification queries, these are identical, given $\mathsf{crs}/\widehat{\mathsf{crs}}$. We can therefore assume without loss of generality that the environment computes them entirely locally, issuing only $(\mathsf{prove}, \mathrm{sid}, x, w)$ queries. We note that if $(x, w) \notin \mathcal{L}$ the queries are also identical, so we assume that the adversary only makes a sequence of (prove) queries where $(x, w) \in \mathcal{L}$. Then we note that if $\mathcal{Z}$ can distinguish between $(\mathcal{D}, \mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}})$ and $(\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}, \mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}})$ with a non-negligible advantage, then $\mathcal{Z}$ is a distinguisher for Figure 2 with a non-negligible advantage. More precisely, we can reframe the access to $\mathcal{F}'^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{D}$ as access to a proving oracle and crs, and reframe access to $\mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{S}''^{\mathcal{L}}_{\mathsf{SNARK}}$ as access to a (simulated) proving oracle and $\widehat{\mathsf{crs}}$. □

Functionality $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$

The protocol $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ further idealises $\mathcal{F}''^{\mathcal{L}}_{\mathrm{NIZK}}$, by ensuring that not-previously seen, verifying proofs are extractable. It is built in the $\mathcal{F}^{D'}_{\mathrm{CRS}}$ hybrid model, and keeps $\widehat{\mathsf{crs}}$, τ, ek, and Π as variables. Π is initialized to $\varnothing$.

Initialisation *On first activation:*
send $(\mathsf{query}, \mathrm{sid})$ **to** $\mathcal{F}^{D'}_{\mathrm{CRS}}$ and **receive the reply** $(\mathsf{query}, \mathrm{sid}, (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}))$

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:*
  if $(x, w) \in \mathcal{L}$ **then**
    let $\pi \leftarrow \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x); \Pi \leftarrow \Pi \cup \{(x, \pi)\}$
    return $(\mathsf{proof}, \mathrm{sid}, x, \pi)$
  else
    return $(\mathsf{proof}, \mathrm{sid}, x, \perp)$
  end if

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:*
  if $(x, \pi) \in \Pi$ **then**
    return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$
  else if $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$ **then**
    if $\exists \pi' : (x, \pi') \in \Pi$ **then**
      let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$
      return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$
    else
      let $w \leftarrow \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi)$
      if $(x, w) \in \mathcal{L}$ **then**
        let $\Pi \leftarrow \Pi \cup \{(x, \pi)\}$
      end if
      return $(\mathsf{verify}, \mathrm{sid}, x, \pi, (x, w) \in \mathcal{L})$
    end if
  else
    return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \perp)$
  end if

Lemma 3. $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\mathcal{L}}$ *UC-emulates* $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\prime\mathcal{L}}$ *in the* $\mathcal{F}_{\mathrm{CRS}}^{D'}$*-hybrid model.*

Proof. We note that $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\mathcal{L}}$ and $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\prime\mathcal{L}}$ differ only in the verification of statement/proof pairs (x, π), where $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$, and x was not previously proved. We note that any distinguishing environment will make some number k of such queries, which must be polynomial in the security parameter.

We construct an adversary, that using $\mathcal{Z}$ breaks simulation sound extractability (Figure 4). $\mathcal{A}$ simulates $\mathcal{Z}$ interacting with $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\prime\mathcal{L}}$ (and $\mathcal{D}$), with $\mathcal{F}_{\mathrm{CRS}}^{D'}$ programmed to return $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek})$. It records all returns w of $\mathcal{E}$, as well as the corresponding x, π inputs in a vector $\vec{e}$. If $\mathcal{Z}$ has advantage α, then with at least probability α, there must exist at least one query $\mathcal{Z}$ made where $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\mathcal{L}}$ and $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\prime\mathcal{L}}$ differed in their output. Specifically, this means that there exists a query to $\mathcal{E}$ such that the extraction failed, and $(x, w) \notin \mathcal{L}$. By the conditions that must be met before $\mathcal{E}$ is called, we also know that $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi)$, and $\nexists \pi' : (x, \pi') \in \Pi$.

$\mathcal{A}$ can test all queries made, and determine which one(s) fail extraction. It then returns (x, π) for which the extraction fails. We note that this directly breaks the extractability property given in Figure 4, with probability at least α. Therefore, if there is no adversary that can break simulation sound extractability except with negligible probability, there exists no environment that has a greater advantage and can distinguish $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\mathcal{L}}$ and $\mathcal{F}_{\mathrm{NIZK}}^{\prime\prime\prime\mathcal{L}}$. □

Simulator $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$

Initialisation *On first activation:*
  send $(\mathsf{query}, \mathrm{sid})$ **to** $\mathcal{F}^{D'}_{\mathrm{CRS}}$ and **receive the reply** $(\mathsf{query}, \mathrm{sid}, (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}))$
  simulate $\mathcal{A}$

Simulating Proofs *On receiving a messsage* (prove, sid, x) *from* $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$*:*
  let $Q \leftarrow Q \cup \{x\}$
  return $(\mathsf{proof}, \mathrm{sid}, \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x))$

Proof Verification *On receiving a message* $(\mathsf{verify}, sid, x, \pi)$ *from* $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$*:*
  if $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$ **then**
    if $x \in Q$ **then**
      send $(\mathsf{maul}, \mathrm{sid}, x, \pi)$ **to** $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$
      return $(\mathsf{ok}, \mathrm{sid}, x, \pi)$
    else
      return $(\mathsf{witness}, \mathrm{sid}, \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi))$
    end if
  else
    return $(\mathsf{reject}, \mathrm{sid}, x, \pi)$
  end if

Lemma 4. *$\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ in the $\mathcal{F}^{D'}_{\mathrm{CRS}}$-hybrid model perfectly UC-emulates $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$.*

Proof. We will show that $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ interacting with $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ perfectly simulates $\mathcal{D}$ interacting with $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$. The environment is presented with three types of actions it can do, as before. It can verify proofs, prove statements, and query $\mathcal{F}^{D'}_{\mathrm{CRS}}$ through $\mathcal{D}$. We note that as $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ perfectly simulates the interactions between $\mathcal{D}$ and $\mathcal{Z}$, the interaction with $\mathcal{F}^{D'}_{\mathrm{CRS}}$ through $\mathcal{D}$ and through $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ is identical.

We note that when receiving a $(\mathsf{prove}, \mathrm{sid}, x, w)$ query, $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ both return $(\mathsf{proof}, \mathrm{sid}, x, \bot)$ if $(x, w) \notin \mathcal{L}$. Otherwise, $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ selects $\pi \xleftarrow{\$} \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, x)$, while $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ queries $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$, which returns a value from the same distribution (note that $\widehat{\mathsf{crs}}$, and τ come from the same distribution D'. Further, both $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ add (x, π) to the set Π. In the query, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ adds x to the set Q. At each update of Π, we note that the relation $Q = \{\ x \mid (x, \pi) \in \Pi\ \}$ is preserved, and Π is equal in $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$. We will revisit this invariant at each point Π is modified. Both $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ return $(\mathsf{proof}, \mathrm{sid}, x, \pi)$.

For $(\mathsf{verify}, \mathrm{sid}, x, \pi)$ queries, we note that Π is identical in $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$ and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$, and that if $(x, \pi) \in \Pi$, both return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$. Otherwise, $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ queries $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$. If $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \bot$, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ will return $(\mathsf{reject}, \mathrm{sid}, x, \pi)$, and $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$ will return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \bot)$, as will $\mathcal{F}'''^{\mathcal{L}}_{\mathrm{NIZK}}$. We note that $\exists \pi' : (x, \pi) \in \Pi \Leftrightarrow x \in Q$. Therefore, if we consider the remaining cases, where $(x, \pi) \notin \Pi$, and $\mathcal{V}(\widehat{\mathsf{crs}}, x, \pi) = \top$, we have the cases that $x \in Q$ and $x \notin Q$ in both functionalities. If $x \notin Q$, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ returns the witness $\mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, x, \pi)$, and if $(x, w) \in \mathcal{L}$, adds (x, π) to Π. This still preserves the relation between Q and Π. Both functionalities return $(\mathsf{verify}, \mathrm{sid}, x, \pi, (x, w) \in \mathcal{L})$, and add (x, π) to Π iff $(x, w) \in \mathcal{L}$. If $x \in Q$, $\mathcal{S}^{\mathcal{L}}_{\mathsf{SNARK}}$ sends $(\mathsf{maul}, \mathrm{sid}, x, \pi)$ to $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$, which (since $x \in Q$), permits the malleability and adds (x, π) to Π. Both will return $(\mathsf{verify}, \mathrm{sid}, x, \pi, \top)$. □

Theorem 3. $\mathsf{NIZK\text{-}SNARK}(\mathcal{L})$ *in the $\mathcal{F}^{D}_{\mathrm{CRS}}$-hybrid model UC-emulates $\mathcal{F}^{\mathcal{L}}_{\mathrm{NIZK}}$.*

Proof. By the transitivity of UC-emulation. □

G Key-Private Forward-Secure Encryption

Lifting to a UC-Protocol A kp-fs-CCA-secure key-evolving encryption scheme induces the following protocol for realizing $\mathcal{F}_{\text{FWENC}}$ in the $\mathcal{F}_{\text{KEYMEM}}$-hybrid model:

Protocol kp-fs-Enc

kp-fs-Enc is parameterized by $\Delta_{\max}$, κ, and N, and operates in the $\mathcal{F}_{\text{KEYMEM}}$-hybrid world, where $\mathcal{F}_{\text{KEYMEM}}$ is parameterized by $\Delta_{\max}$, and the following Update function:

function $\mathsf{Update}((sk, \tau))$
  return $(\mathsf{Upd}(sk, \tau+1), \tau+1)$
end function

On receiving a message (KeyGen, sid) *for the first time:*

Let $(pk, sk^0) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, N)$.
Send $(\mathsf{Init}, \text{sid}, (sk^0, 0))$ to $\mathcal{F}_{\text{KEYMEM}}$.
Erase sk^0.
Record $\tau \leftarrow 0$
return pk

On receiving a message $(\mathsf{Encrypt}, sid, pk, \tau', m)$*:*

return $\mathsf{Enc}_{pk}(\tau', m)$

On receiving a message $(\mathsf{Decrypt}, sid, \tau', c)$*:*

if $\tau' < \tau$ **then**
  return $\perp$
else
  Send $(\mathsf{Get}, \text{sid})$ to $\mathcal{F}_{\text{KEYMEM}}$, denoting the response as $(sk^\tau, \cdot)$.
  Compute $sk^{\tau'}$ from sk^τ.
  Let $m \leftarrow \mathsf{Dec}_{sk^{\tau'}}(\tau', c)$.
  Erase $sk^{\tau'}$ and sk^τ.
  return m
end if

On receiving a message (Update, sid)*:*

Record $\tau \leftarrow \tau + 1$.
Send $(\mathsf{Update}, \text{sid})$ to $\mathcal{F}_{\text{KEYMEM}}$.

The Simulator We now give 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.

On initialization:

Let $(pk_{\mathsf{dummy}}, \cdot) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, N)$
Record pk_{dummy}

On receiving a message $(\mathsf{KeyGen}, sid, U_p)$*:*

Let $(pk_p, sk_p^0) \xleftarrow{\$} \mathsf{Gen}(1^\kappa, N)$
Record pk_p, sk_p^0, and $\tau_p \leftarrow 0$
Simulate sending $(\mathsf{Init}, \text{sid}, (sk_p^0, 0))$ to $\mathcal{F}_{\text{KEYMEM}}$ as U_p.
return pk_p

On receiving a message $(\mathsf{Encrypt}, sid, \tau, U_p, l)$*:*

Let $m \xleftarrow{\$} 0^l$
Let $c \xleftarrow{\$} \mathsf{Enc}_{pk_{\mathsf{dummy}}}(\tau, m)$
return c

On receiving a message $(\mathsf{DummyEncrypt}, sid, pk, \tau, m, U_p)$*:*

Let $c \xleftarrow{\$} \mathsf{Enc}_{pk}(\tau, m)$
return c

On receiving a message $(\mathsf{Decrypt}, sid, \tau, c, U_p)$*:*

if U_p's secret key sk_p^0 is recorded **then**
  Use Upd to derive sk_p^τ.
  Let $m \leftarrow \mathsf{Dec}_{sk_p^\tau}(\tau, c)$
  Return m
else
  return $\perp$
end if

On receiving a message $(\mathsf{Update}, sid, U_p)$*:*

Record $\tau_p \leftarrow \tau_p + 1$
Simulate sending $(\mathsf{Update}, \text{sid})$ to $\mathcal{F}_{\text{KEYMEM}}$ as U_p.

On receiving messages to $\mathcal{F}_{\text{KEYMEM}}$ *from* $\mathcal{A}$*:* Forward these messages to the simulated $\mathcal{F}_{\text{KEYMEM}}$.

UC Emulation

Theorem 4. *If the underlying key-evolving PKE scheme is kp-fs-CCA secure then* kp-fs-Enc *UC-emulates* $\mathcal{F}_{\text{FWENC}}$ *in the* $\mathcal{F}_{\text{KEYMEM}}$*-hybrid world.*

Proof. The points in which the simulator $\mathcal{S}_{\text{FWENC}}$, combined with $\mathcal{F}_{\text{FWENC}}$ can behave differently from kp-fs-Enc are in how they respond to various queries, and the internal state they maintain. kp-fs-Enc 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. Further, both initialize τ_p 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, τ_p is updating the same way, and matches the ideal functionality's τ_p 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 $U_1, \ldots, U_n$ 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 pk_{dummy} exists only in the ideal world, and it's 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 $(pk_1, sk_1), \ldots, (pk_n, sk_n), (pk_{\mathsf{dummy}}, sk_{\mathsf{dummy}})$ 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 environment $\mathcal{Z}$ 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 $\mathcal{A}$ in 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 parties secret keys, on a Decrypt query to the simulator, it posts a $\mathsf{decrypt}(\tau, c, U_p)$ query, and return 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 $\mathsf{corrupt}(\tau_p + \Delta_{\max}, U_p)$ query to obtain $sk_p^{\tau_p + \Delta_{\max}}$. While $\mathcal{F}_{\text{KeyMem}}$ at the time of corruption stores $sk_p^{\tau_p}$, by assumption it will first apply $\Delta_{\max}$ updates.
- When the ideal functionality receives an $(\mathsf{Encrypt}, \mathrm{sid}, pk_p, \tau, m)$ query, iff it does not reveal m to the simulator, $\mathcal{A}$ queries $\mathsf{challenge}(\tau, (U_{\mathsf{dummy}}, 0^{|m|}), (U_p, m))$, 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 $\mathcal{F}_{\text{FwEnc}}$, the challenges will be for, at latest, the time $\tau_p + \Delta_{\max} - 1$. On corruption, the $\mathsf{corrupt}(k, U_p)$ query is make with $k = \tau_p + \Delta_{\max}$. As τ_p is monotonically increasing, and Encrypt is not called after corruption – and therefore no further challenge queries are issued – the corruption is after all challenges. For b), we note that on corruption, $\mathcal{F}_{\text{FwEnc}}$ will no longer query the simulator with Encrypt queries for this party, but only with DummyEncrypt queries. As challenge queries are only issued on Encrypt queries, this party will not longer receive challenge queries.

Next, if $b = 0$, the execution perfectly matches a random ideal world execution with $\mathcal{S}_{\text{FwEnc}}$. Specifically, if $b = 0$ the result of $\mathsf{challenge}(\tau, (U_{\mathsf{dummy}}, 0^{|m|}), (U_p, m)) = \mathsf{Enc}_{pk_{\mathsf{dummy}}}(\tau, 0^{|m|})$. Further, $\mathsf{decrypt}(\tau, c, U_p) = \mathsf{Dec}_{sk_p^\tau}(\tau, c)$, i.e. 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 kp-fs-Enc is negligible. Honest parties perform four operations in kp-fs-Enc: 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. τ is initialized to 0 for U_p 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 $\mathsf{challenge}(\tau, (U_{\mathsf{dummy}}, 0^{|m|}), (U_p, m)) = \mathsf{enc}_{pk_p}(\tau, m)$, the ciphertext will be sampled from $\mathsf{enc}_{pk_p}(\tau, m)$, the same as 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 $\mathsf{Dec}_{sk_p^\tau}(\tau, c)$. 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 no by the correctness of the underlying key-evolving encryption scheme, that with overwhelming probability, the decryption must return the same plaintext. For update, τ_p 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. □