Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake

Thomas Kerber
The University of Edinburgh and IOHK
t.kerber@ed.ac.uk

Aggelos Kiayias
The University of Edinburgh and IOHK
akiayias@ed.ac.uk

Markulf Kohlweiss
The University of Edinburgh and IOHK
mkohlwei@ed.ac.uk

Vassilis Zikas
The University of Edinburgh and IOHK
vzikas@ed.ac.uk

Abstract—**We present Ouroboros Crypsinous, the first formally analyzed privacy-preserving proof-of-stake blockchain protocol. To model its security we give a thorough treatment of private ledgers in the (G)UC setting that might be of independent interest. To prove our protocol secure against adaptive attacks, we introduce a new coin evolution technique relying 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 blockchain, such as the proof-of-work-based Zerocash are analyzed only against static corruptions.**

I. Introduction

A significant limitation of traditional blockchain protocols, such as Bitcoin, is the fact that the transaction ledger is a public resource and thus significant information about the way the transaction issuers operate may be leaked to an adversary. This consideration was acknowledged early on and Bitcoin itself [29] 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 are supposedly 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., [31], [25].

This state of affairs motivated the development of privacy enhancing and privacy preserving techniques for distributed ledgers. First, methods such as CoinJoin [24] and CoinShuffle [32] 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 Zerocoin [27], Zerocash [4], and Cryptonote [33]. We note that despite their enhanced privacy characteristics, deploying these protocols in practice, as e.g., in the Monero or Zcash cryptocurrencies, may introduce some leakage (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 [23], [28], [20]. Still, protocols like Zerocash have theoretically strong, provable privacy guarantees and can, in principle, provide the foundations for a highly private transaction ledger. Interestingly though, invariably all the 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 led to a sequence of works in proof-of-stake (PoS) blockchain protocols that include Algorand [26], Ouroboros [21], Ouroboros Praos [13], Ouroboros Genesis [1], Sleepy-Consensus [30], 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*). 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 [34] 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

Work partly supported by H2020 project #780477, PRIViLEDGE.

[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.

is inspired by Ouroboros Genesis, [1], a recent (non-private) PoS blockchain protocol formally analyzed in the UC setting. Our protocol provides the first PoS-based privacy-enhanced blockchain protocol. Moreover, for the first time our protocol achieves simulation-based security that is even universally composable, 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 withstands adaptive attacks. We note that previous work on provable privacy enhanced ledgers (in the proof-of-work setting), notably [4] is analysed in the static corruption setting using game-based definitions for security.

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 [26] 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.

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 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. [16]. 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 [16], 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.

II. 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 VI, 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.

A. 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 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

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

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.

B. 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.

C. 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.

D. 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 VIII 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, Δ_{max}. This allows us to utilize forward secure encryption instead of non-committing encryption, under the assumption that corruption is "delayed" by Δ_{max}. This delay is modeled by restricting adversarial access to the forward secure encryption secret key at time τ to the key for time $\tau + \Delta_{\mathsf{max}}$.

III. 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 V. 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 $\mathcal{F}^{\Delta}_{\text{N-MC}}$. Every honest party can send a message thought $\mathcal{F}^{\Delta}_{\text{N-MC}}$ 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. However, as in [17], [1] our protocol is implicitly aware of an overestimate Δ_{max} of the actual (unknown) network delay Δ. However, this Δ_{max} is not used in the message passing; instead the protocol proceeds in an optimistic manner once messages are received (after at most Δ rounds from sending), and Δ_{max} is only used in the staking procedure to determine the leader(s) of each slot.

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: a) *Offline* parties are honest parties that are deregistered from the network functionality. b) *(Fully) online* parties are registered with all their setups and ideal resources. c) *(Online but) stalled* parties 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 than `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 V.)

- 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.
- The bounded-delay authenticated channels network $\mathcal{F}^{\Delta}_{\text{N-MC}}$ 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 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 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 without loss of generality 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 public-key infrastructure (PKI) or a common reference string (CRS) and their security proofs are described in Section IV. 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 IV-B) an implementation of this primitive against fully adaptive adversaries might be impossible without additional setup assumptions. Instead, here we make an assumption about the (in)ability of the adversary to quickly read keys of newly corrupted parties and prove the security of our protocols under this assumption. Proving impossibility of the primitive against a fully adaptive adversary (or providing a protocol for it) is an interesting future direction.

Finally, our construction will make use of non-interactive equivocal commitments and pseudo-random functions (PRFs). Construction of both these primitives exists in the CRS model under standard hardness assumption, notably the hardness of the DDH (Decisional Diffie Hellman) problem.

Remark 1: (Assumptions on the environment/adversary as setup-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 compromising composability: Instead of explicitly restricting the adversary and environment, we introduce a functionality wrapper that wraps the (local setup) functionalities that the protocol accesses and forces the required assumptions on the adversary/environment. We refer to [2] for a more detailed discussion. The wrapper used in our security statements is left implicit here; a more explicit statement can be found in the full version of this paper.

IV. Tools

In this section we describe the main tools used by Ouroboros Crypsinous: non-interactive zero-knowledge (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. Further, we define the properties satisfied by MUPRFs and equivocal commitments. Ouroboros Crypsinous will then be described and proved secure assuming hybrid access to the corresponding ideal functionalities and its security when these functionalities are replaced by their implementations will follow directly from the universal composition theorem.

A. Non-Interactive Zero Knowledge

We utilize the Non-Interactive Zero Knowledge functionality $\mathcal{F}_{\text{NIZK}}$, and protocol of [22]. This functionality allows generating proofs π that a statement $\mathbf{x}$ is in a (fixed) 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 NIZK can be used for signature-like behavior by embedding the messages that are to be signed in the statements of simulation-extractable NIZKs, constructing a *signature of knowledge* [18] (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 have similar unforgeability properties as standard signatures.

B. Key-private Forward-Secure Encryption

In order to construct Zerocash-like transactions, an encryption mechanism is necessary, for parties to send information about newly created coins to their recipients. To preserve the anonymity of Crypsinous transactions, key-privacy [3] is a necessary property of this encryption. Furthermore, encryption in the UC setting is required to be non-committing in order to withstand adaptive corruptions, as the simulator must create simulated ciphertexts, which it may later need to reveal the message of.

While key-private, non-committing encryption would satisfy the needs of our protocol, practical constructions are inefficient, especially considering parties must attempt to decrypt each message, in case it is for them. Instead, we utilize a key-private encryption with forward-security, and a time-sensitive non-committing property. This weaker, time-sensitive, non-committing property is sufficient to realize Ouroboros Crypsinous. Informally, only messages addressed to a time window of size Δ_{max} into the future are protected. Nonetheless, even this notion seems too strong to be implementable against a fully adaptive adversary in a Δ-bounded-delay network. Intuitively, the reason is the following: A common way to realize non-interactive non-committing encryption via erasures is to have parties update their keys once the message is received. The ideal is that a message is encrypted at round τ so that it can be decrypted with key sk_{τ}^{ENC}, and sent over to the receiver. 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. However, this is not possible 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 Δ_{max}, to attack during which the message is already being transmitted but has not yet been received by the recipient. This makes erasures useless in this window.

To bypass this, we make an assumption on the adversary's adaptiveness which, roughly, implies that the adversary cannot immediately see the secret key of a newly corrupted party.

Specifically, we assume that the adversary corrupting a party with key sk^{ENC}_{τ} at time τ does not receive sk^{ENC}_{τ}, but rather the key $sk^{\text{ENC}}_{\tau+\Delta_{\max}}$, which this party would hold in time $\tau + \Delta_{\max}$, if it were allowed to properly update its key. We note that this is a milder assumption than that of delayed party-corruption which underlines the security of [21], [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.

The straightforward way of enforcing the assumption would be to make all our statements for a restricted class of adversaries. However, for reasons similar to the discussion in Remark 1 above, this would immediately imply that universal composition no longer holds. Instead, we use the approach from [2], [14] and 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. As an added bonus of using the above functionality-based approach for restricting the adversary, we ensure 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, Ouroboros Crypsinous immediately become secure against such an adversary. 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 in Appendix F.

C. 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. While such a property can easily be satisfied by a random oracle, Ouroboros Crypsinous invokes it within the NP language of a NIZK. Specifically, we would need to not just assume the existence of a random oracle, but that a specific polynomial function, such as well known symmetric primitives, constitute a random oracle. We instead choose a standard-model construction.

We propose the following construction. Let $H : \{0,1\}^* \to \langle g \rangle$ be a function mapping to the cyclic group generated by g that is selected according to 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. Then we 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 the following is a DDH triple $\langle g^k, H(m), H(m)^k \rangle$ 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, $\Pr[H(x)^k < T] \leq \sum_{y<T} \Pr[H(x)^k = y] = T \Pr[H(x) = y^{1/k}] \leq T/2^{\kappa}$ in the conditional space $x \notin Q_H$.

D. Equivocal Commitments

We make use of a standard non-interactive equivocal commitment scheme, $(\mathsf{Init}_{\mathsf{comm}}, \mathsf{Comm}, \mathsf{DeComm}, \widehat{\mathsf{Init}_{\mathsf{comm}}}, \widehat{\mathsf{Comm}}, \mathsf{Equiv})$, which is secure in the CRS model assuming hardness of discrete logarithms (cf. [12]). This is used in the simulation to open coin commitments to a specific party's public key, when this party is corrupted. For self-containment we have included a high-level description, including some notation used in our proofs in Appendix G.

V. The Private Ledger

We next provide the description of the private ledger functionality that, as we prove, is implemented by Ouroboros Crypsinous. The private ledger is based on previous UC definitions of distributed ledgers [2], [1]. Due to the complexity of this functionality, and the fact that our modifications do not alter the main component of it, namely the consensus mechanism itself, we will only present how privacy affects the definition of an ordinary ledger. For a full functionality, please see the full version of this paper. 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.

Privacy of Crypsinous is captured by the following modifications: First, an ID generating mechanism is added, by submitting GENERATE queries. This allows parties to create new pseudonyms as desired. Second, transactions themselves are 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 a GENERATE query, or the special symbol PUBLIC, denoting the sub-transaction is publicly readable. Third, 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 $\perp$. Concretely, we introduce *blinding* functions $\mathsf{BlindTx}$ and Blind, described below, which hide parts of the ledger from read requests. Finally, we parameterize the private ledger with a general purpose leakage algorithm, Lkg, which the adversary is permitted to query. An overview of the functionality can be found in Figure 1.

- $\mathsf{BlindTx}$ takes as input an annotated transaction $\mathsf{BTX} = (\mathsf{tx}, \text{txid}, \tau_L, U_s)$, 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.[3] We make use of the commonly used higher-order function map, which applies a function to a list element-wise, and also implicitly use *currying*, i.e. a function applied to less arguments than it is defined for should be considered a *partial application* of this function.

$$\begin{aligned}
&\mathsf{BlindSTx}(\mathcal{P}, \mathsf{ids}, (pk, \mathsf{stx})) \triangleq \text{if } pk = \text{PUBLIC} \lor \\
&\qquad \exists U_p \in \mathcal{P} : (U_p, \text{ID}, pk) \in \mathsf{ids} \\
&\quad \text{then } (pk, \mathsf{stx}) \text{ else } (\perp, |\mathsf{stx}|) \\
&\mathsf{BlindTx}(\mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \text{txid}, \cdot, \cdot)) \triangleq \\
&\quad (\mathsf{map}(\mathsf{BlindSTx}(\mathcal{P}, \mathsf{ids}), \mathsf{tx}), \text{txid}) \\
&\mathsf{BlindTx}_{\mathcal{A}}(\mathcal{P}, \mathsf{ids}, (\mathsf{tx}, \text{txid}, \tau_L, U_s)) \triangleq \\
&\quad (\mathsf{map}(\mathsf{BlindSTx}(\mathcal{P}, \mathsf{ids}), \mathsf{tx}), \text{txid}, \tau_L, U_s)
\end{aligned}$$

- Blind is similar to $\mathsf{BlindTx}$ but operates on states. While this introduces subtleties regarding block representation, it is sufficient to think of as replacing each transaction in a state with its blinded version. For more detail, see the full version of this paper.
  $\mathsf{Blind}_{\mathcal{A}}$ is defined the same as Blind, but with calls to $\mathsf{BlindTx}$ replaced with calls to $\mathsf{BlindTx}_{\mathcal{A}}$.

In our system, we permit the leakage $\mathsf{Lkg}_{\mathsf{lead}}$ (Figure 2), 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].

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

$\mathcal{G}_{\text{PL}}$ is parameterized by two main algorithms, $\mathsf{Validate}$, and Lkg, along with one main parameter: the initial coin distribution $\mathfrak{C}_1 := \{(U_1, s_1), \ldots, (U_n, s_n)\}$. These parameters are all publicly known. The functionality manages a fixed ledger state, state, a buffer of unconfirmed transaction, buffer, the sequence of generated IDs, ids, the sequence of honest inputs, $\vec{\mathcal{I}}_H^T$, and a pointer pt_p for each party U_p, indicating its local state, i.e. the length of the prefix of state, which is visible to U_p. We write $\vec{\mathsf{pt}}$ to refer to a vector of all parties local state pointers We will refer to the set of honest parties as $\mathcal{H}$, and the set of all registered parties as $\mathcal{P}$.

Upon receiving any input I from any party U_p or from the adversary, retrieve the current time τ_L from $\mathcal{G}_{\text{CLOCK}}$, and record the interaction in $\vec{\mathcal{I}}_H^T$. Specifically, if $I \neq (\text{SUBMIT}, \text{sid}, \mathsf{tx})$, set $\vec{\mathcal{I}}_H^T \leftarrow \vec{\mathcal{I}}_H^T \,\|\, (I, U_p, \tau_L)$.

Submitting transactions. If $I = (\text{SUBMIT}, \text{sid}, \mathsf{tx})$: a) Choose a unique transaction ID txid and set $\mathsf{BTX} := (\mathsf{tx}, \text{txid}, \tau_L, U_p)$. 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) Set $\vec{\mathcal{I}}_H^T \leftarrow \vec{\mathcal{I}}_H^T \,\|\, ((\text{SUBMIT}, \text{sid}, \mathsf{BlindTx}_{\mathcal{A}}(\mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{tx})), U_p, \tau_L)$. d) Send $(\text{SUBMIT}, \mathsf{BlindTx}_{\mathcal{A}}(\mathcal{P} \setminus \mathcal{H}, \mathsf{ids}, \mathsf{BTX}))$ to $\mathcal{A}$.

Generating IDs. If $I = (\text{GENERATE}, \text{sid}, \mathsf{tag})$: query the adversary with $(\text{GENERATE}, \text{sid}, U_p, \mathsf{tag})$, 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 $I = (\text{READ}, \text{sid})$: set $\mathsf{state}_p := \mathsf{state}|_{\min\{\mathsf{pt}_p, |\mathsf{state}|\}}$ and return $(\text{READ}, \text{sid}, \mathsf{Blind}(\{U_p\}, \mathsf{ids}, \mathsf{state}_p))$ to the requestor. If the requestor is $\mathcal{A}$ then send $(\mathsf{Blind}_{\mathcal{A}}(\mathcal{P} \setminus \mathcal{H}, ids, \mathsf{state}), \mathsf{map}(\mathsf{BlindTx}_{\mathcal{A}}(\mathcal{P} \setminus \mathcal{H}), \mathsf{buffer}), \mathsf{Lkg}(\mathsf{state}, \mathsf{buffer}, \tau_L), \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.

Ledger maintainance. If $I = (\text{MAINTAIN-LEDGER}, \text{sid})$, the ledger performs *maintainance*. The full consensus procedure is not the focus of this paper, and is described in detail in [2], [1]. As a rough overview, it ensures the following properties: a) Transactions in buffer are continuously re-validated as in SUBMIT. b) Valid transactions in buffer will, within a fixed amount of time get appended to state. c) The values of all state pointers pt_p will lag behind the most current state by at most a fixed amount of time, k. d) state is append-only. Beyond these constraints, the adversary is free to control the values state and pt_p.

Figure 1. The private ledger

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.

In a preliminary step of our analysis we also utilize a leak-

[3] We note that if we assumed an anonymous broadcast, the submitter would not be leaked.

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

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 sampling from the leakage.

```
procedure $\mathsf{Lkg}_{\mathsf{lead}}(\mathsf{state}, \mathsf{buffer}, \tau)$
    if $L_\tau$ is recorded then return $L_\tau$
    Determine $ep$, the epoch for the time slot $\tau$.
    Determine $\tau_{ep}$, the time at which the stakeholder distribution for the epoch $ep$ was frozen.
    Let $L \leftarrow \varnothing; S \leftarrow \varnothing$
    for each party $U_p$ do
        Determine the valid coins of $U_p$ in $\mathsf{state}_{\tau_{ep}}$, that were not adversarially generated.
        Determine $U_p$'s relative stake of these coins $\alpha_{U_p}$.
        With probability of $\phi_f(\alpha_{U_p})$, add $U_p$ to $L$.
    end for
    for each adversarially generated coin $\mathbf{c}$ do
        if $\mathbf{c}$ was spent in $\mathsf{state}$ or $\mathsf{buffer}$ then
            Let $\mathsf{tx}$ 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
```

Figure 2. The leakage function for Ouroboros Crypsinous

age 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. We note that 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 ledger.

VI. The Ouroboros-Crypsinous Protocol

In this section we provide a detailed description of our protocol Ouroboros-Crypsinous as a synchronous (G)UC protocol. The protocol has a similar – and in many parts identical – structure as Ouroboros-Genesis [1], but differs considerably in the leadership election, and processing of transactions. As already discussed, the protocol assumes access to numerous functionalities, including global setup, networks, encryption, and NIZK.

A. 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 $(\textsc{public}, \textsc{transfer})$ 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: $((\textsc{public}, \textsc{transfer}), (pk_r, \mathbf{c}_4), (pk_s, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3))$, 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 $\textsc{generate}$ 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 $\textsc{generate}$ 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 $\textsc{read}$ 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 $\perp$.

B. 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 IV-C, with entropy supplied by the epoch randomness η_{ep}, by being evaluated at the slot index sl and η_{ep}, seeded with the "winning coin's secret key" $\mathsf{root}_{\mathbf{c}} \parallel \rho_{\mathbf{c}}$. 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 IV-C, 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 $\rho_{\mathbf{c}}$, $sk_{\mathbf{c},sl}^{\text{COIN}}$ 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: $\rho_{\mathbf{c}'} = \mathsf{PRF}_{\mathsf{root}_{\mathbf{c}}}^{\mathsf{evl}}(\rho_{\mathbf{c}})$. 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.

C. Real World Transactions

Before giving the formal specification we introduce some necessary terminology and notation. Each party U_p stores a local blockchain $\mathcal{C}_{\mathsf{loc}}^{U_p}$—$U_p$'s local view of the blockchain.[4] 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 $\mathsf{tx}_{\mathsf{lead}} = (\textsc{lead}, \vec{\mathsf{stx}}_{\mathsf{ref}}, \mathsf{stx}_{\mathsf{proof}})$, 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, $(cm_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{Comm}(pk_{\mathbf{c}}^{\mathrm{COIN}} \,\|\, v_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}'})$ is the commitment of the newly-created coin. $sn_{\mathbf{c}} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{\mathbf{c}}}(\rho_{\mathbf{c}})$ is the serial number of the coin $\mathbf{c}$, which is revealed to demonstrate the coin has not been spent. We define $\rho = \mu^{\mathsf{root}_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}}}$, where μ is $\mathcal{G}_{\mathrm{RO}}$ evaluated at $\textsc{nonce} \,\|\, \eta_{ep} \,\|\, sl$. 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 $\textsc{lead}$ (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. We note the transfer transactions and general-purpose transactions correspond directly to ideal-world transactions with the same behavior. 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 ($\textsc{public}$), 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 $\textsc{public}$, or an encryption of the content using $\mathcal{F}_{\mathrm{FwEnc}}$, to the party specified as the recipient. Upon reading the state, parties attempt to decrypt ciphertexts, and failing that, replace it with $\bot$. To disambiguate transactions, we prefix generic transactions with the label $\textsc{generic}$.

The implementation of transfer transaction 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 $\mathsf{tx}_{\mathsf{xfer}}^{\mathsf{ideal}} = ((\textsc{public}, \textsc{transfer}), (pk_r, (\mathsf{id}_4, v_4)), (pk_s, (\mathsf{id}_1, v_1), (\mathsf{id}_2, v_2), (\mathsf{id}_3, v_3)))$ – with a Zerocash-like construction. We define a real transfer transaction to be: $\mathsf{tx}_{\mathsf{xfer}}^{\mathsf{real}} = (\textsc{transfer}, \mathsf{stx}_{\mathsf{proof}}, c_r)$, where $\mathsf{stx}_{\mathsf{proof}} = (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \mathsf{root}, \pi)$, and c_r is a $\mathcal{F}_{\mathrm{FwEnc}}$-encryption for the slot the transaction was submitted in $\mathsf{stx}_{\mathsf{rcpt}} = (\rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_{\mathbf{c}_3})$ to pk_r. Similar to leadership transactions, $(cm_{\mathbf{c}_3}, r_{\mathbf{c}_3}) = \mathsf{Comm}(pk_{\mathbf{c}_3}^{\mathrm{COIN}} \,\|\, v_{\mathbf{c}_3} \,\|\, \rho_{\mathbf{c}_3})$, and $(cm_{\mathbf{c}_4}, r_{\mathbf{c}_4}) = \mathsf{Comm}(pk_{\mathbf{c}_4}^{\mathrm{COIN}} \,\|\, v_{\mathbf{c}_4} \,\|\, \rho_{\mathbf{c}_4})$. $sn_{\mathbf{c}_1}$ and $sn_{\mathbf{c}_2}$ are revealed to spend the coins $\mathbf{c}_1$ and $\mathbf{c}_2$ respectively, and π proves the statement $\textsc{xfer}$ (defined in Appendix D), specifically proving the existence of $cm_{\mathbf{c}_1}$ and $cm_{\mathbf{c}_2}$, in a Zerocash-like Merkle tree of all valid coin commitments with the root root, as well as various consistency properties, described in detail in Appendix D. We note that the use of $\mathcal{F}_{\mathrm{FwEnc}}$ 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 $\textsc{read}$ 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 $(\mathsf{root}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})$.

D. Interacting with the Ledger

At the core of the Ouroboros Crypsinous protocol is the process that allows parties to maintain the ledger. There are four types of processes that are triggered by four different commands, provided that the party is already registered to all its local and global functionalities.

- The command $(\textsc{submit}, \mathrm{sid}, \mathsf{tx})$ 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 $(\textsc{generate}, \mathrm{sid}, \mathsf{tag})$ is used for creating new addresses, which can be used by other parties to transfer funds to this current party (if $\mathsf{tag} = \textsc{coin}$), and for initially creating a parties public keys (if $\mathsf{tag} = \textsc{id}$).
- The command $(\textsc{read}, \mathrm{sid})$ 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 $(\textsc{maintain-ledger}, \mathrm{sid})$ 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; and finally it invokes the staking procedure unless it has already done so in the current round.

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,

[4] 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 $\mathcal{C}_{\mathsf{loc}}^{U}$.

and a detailed specification of the protocols SubmitXfer and SubmitGeneric.

a) 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, using the separate encryption, and coin keys instead of VRF (verifiable random function) and KES (key-evolving signature) keys. The precise description of the initialization procedure is omitted, and can be found in the full version of this paper.

b) 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 $\mathsf{root}_{\mathbf{c}}$, 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. We note that 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 is 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 $\phi_f(\alpha_{\mathbf{c}})$ as [1] to determine the probability of a coin winning the leadership election, with the corresponding threshold, $T_{\mathbf{c}} = \mathsf{ord}(G)\phi_f(\alpha_{\mathbf{c}})$, where G is the target group of our MUPRFs. We note that 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-A. It evaluates two distinct MUPRFs for each eligible coin. If the output of one of these is under the target for some coin, the party is a slot leader, and continues to create a new block B from their current transaction buffer. Aside of the main contents, the party assembles a leadership transaction and assigns it to the block. This transactions 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, and records received messages, and performs the chain selection of [1]. The full description can be found in Appendix C-B

c) 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 is specified in detail in the full version of this paper. For generic transactions, each subtransaction in the submitted transaction tx is mapped, if it is addressed to a party U_r, to a ciphertext encrypted with $\mathcal{F}_{\text{FwEnc}}$, party U_r's public key, and the current time τ. The ciphertext-mapped transaction is then broadcast. The full description of this is omitted here, and can be found in the full version of this paper.

d) Reading the State: The last command related to the interaction with the ledger is the read command $(\text{READ}, \mathrm{sid})$ 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 log, recording information necessary to reconstruct the ideal-world representation of the transaction. The full description of this reconstruction can be found in the full version of this paper.

E. Transaction Validity

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

a) 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 it's 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). As a special case, if the ID of $\mathbf{c}_2$ is $\perp$, and $v_2 = 0$, it is always valid.[5] 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. (We note 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 by $(\text{GENERATE}, \mathrm{sid}, \text{ID})$ by the party. If the sending party is corrupted, it may spend the coins of other corrupted parties. 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 (with the exception of the id $\perp$.

b) 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 transaction 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 validity, 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 blocks 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 $(\text{PUBLIC}, \text{TRANSFER})$.

VII. Security Analysis

We split our security analysis of Ouroboros-Crypsinous into two parts: First, we show that Ouroboros-Crypsinous realizes a "leaky" version of $\mathcal{G}_{\text{PL}}$ – specifically, we show that it realizes $\mathcal{G}_{\text{PL}}$ with Lkg set to the identity function $\mathsf{Lkg}_{\mathsf{id}}$; i.e. the ledger leaks its entire content to the simulator, described in Appendix B. We argue that the simulator $\mathcal{S}_1$ can simulate any real-world attacks on Ouroboros-Crypsinous against a leaky $\mathcal{G}_{\text{PL}}$. In the second stage, we instantiate Lkg to $\mathsf{Lkg}_{\mathsf{lead}}$, in which only the leaders of a given slot are leaked. We argue that $\mathcal{S}_2$ is secure against this, as it behaves indistinguishably from $\mathcal{S}_1$. The smaller steps of this proof are stated here, and argued in Appendix E.

Theorem 1. Ouroboros-Crypsinous *UC-emulates* $\mathcal{G}_{\text{PL}}$ *with* $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$, *under the DDH assumption, in the random oracle model*[6].

We prove Theorem 1 by first establishing the similarity of Ouroboros-Crypsinous from Ouroboros-Genesis [1], showing that the usage of NIZKs, key-private forward-secure encryption, and MUPRFs is equivalent to Genesis' usage of forward secure signatures, and VRFs. With this, we argue that creating leadership transactions in Crypsinous is equivalent to creating leadership proofs in Genesis, allowing us to leverage the security analysis from Ouroboros Genesis [1]. We conclude that Ouroboros-Crypsinous realizes a version of the standard, Genesis, ledger $\mathcal{G}_{\text{LEDGER}}$, with a different interface to the environment. Specifically, we note that incoming transactions are mapped to "real" transaction, and outgoing transactions are mapped back to "ideal" transactions. We establish that this mapping is reversed in such a way that the output directly corresponds to that of the ideal private ledger, and does not impact the validation predicate. We note that due to the similarity of the standard ledger and the private ledger with the total leakage, combined with the modified output

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

[6] We will be working under these assumptions throughout the rest of the security analysis, and will typically leave them 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.

interface, Ouroboros-Crypsinous realizes the private ledger with $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$.

Lemma 1. *The private ledger $\mathcal{G}_{\text{PL}}$ with $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$ is equivalent to the standard ledger $\mathcal{G}_{\text{LEDGER}}$ from [1] with a different interface.*

Lemma 2. *A party can make a leadership transaction in* Ouroboros-Crypsinous *if and only if the party can make a corresponding leadership proof in* Ouroboros-Genesis.

Lemma 3. *The mappings used in* Ouroboros-Crypsinous *are consistent with the interface difference implicit in the $\mathcal{G}_{\text{PL}}$ to $\mathcal{G}_{\text{LEDGER}}$ reduction.*

Corollary 1. Ouroboros-Crypsinous *UC-emulates $\mathcal{G}_{\text{LEDGER}}$, with a different interface.*

Theorem 2. Ouroboros-Crypsinous *UC-emulates $\mathcal{G}_{\text{PL}}$ with $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{lead}}$ under the DDH assumption, in the random oracle model.*

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-B. We prove that the simulated transactions are indistinguishable from real transactions, and the reconstructed party state is indistinguishable from a real party's state. Therefore the simulator 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.

Lemma 4. *Simulated transactions are indistinguishable from real transactions.*

Lemma 5. *Corrupting an ideal-world party is indistinguishable from corrupting a real-world party.*

VIII. 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. [19].

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 $sk^{\text{COIN}}_{\mathbf{c}_2}$ 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 [19] 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 II, 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 ([19], page 128)
Hidden range proof	256
Pedersen commitment	1,006 + 2.666 per bit[7]

Table I
NUMBER OF MULTIPLICATIVE CONSTRAINTS IN SNARK RELATIONS

Constraint count	$\mathcal{L}_{\text{XFER}}$	$\mathcal{L}_{\text{LEAD}}$
Check $pk^{\text{COIN}}_{\mathbf{c}_i}$	$2 \times 27,904$	$27,904$
Check $\rho_{\mathbf{c}_2}$, $sk^{\text{COIN}}_{\mathbf{c}_2}$		$2 \times 27,904$
Path for $cm_{\mathbf{c}_i}$	$2 \times 43,808$	$43,808$
(1 layer of 32)	$(1,369)$	$(1,369)$
Path for $\mathsf{root}_{sk^{\text{COIN}}_{\mathbf{c}_i}}$		$34,225$
(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 II
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.

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.

[7] https://github.com/zcash/zcash/issues/2634

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] Eiichiro Fujisaki and Tatsuaki Okamoto. How to enhance the security of public-key encryption at minimum cost. In Hideki Imai and Yuliang Zheng, editors, *PKC'99*, volume 1560 of *LNCS*, pages 53–68. Springer, Heidelberg, March 1999.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] 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.

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

[27] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

Appendix 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 $s_1, \dots, s_n$. $\mathcal{F}_{\text{INIT}}$ interacts with stakeholders $U_1, \dots, U_n$ as follows:

- In the first round, upon a request from some stakeholder U_i of the form $(\mathsf{claim}, \mathrm{sid}, U_i, pk_i^{\text{ENC}})$, then $\mathcal{F}_{\text{INIT}}$ stores the keys tuple (U_i, pk_i^{ENC}). It then samples $sk_{\mathbf{c}_i}^{\text{COIN}}$ the way Ouroboros-Crypsinous does on GENERATE requests, $\rho_{\mathbf{c}_i}$ randomly, computes $pk_{\mathbf{c}_i}^{\text{COIN}} \leftarrow \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}_{sk_{\mathbf{c}_i}^{\text{COIN}}}}(0)$, and commits $(cm_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \mathsf{Comm}(pk_{\mathbf{c}_i}^{\text{COIN}} \parallel s_i \parallel \rho_{\mathbf{c}_i})$, and returns the tuple $(sk_{\mathbf{c}_i}^{\text{COIN}}, \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}, s_i)$. Once all parties have registered, it samples and stores a random value $\eta_1 \overset{\$}{\leftarrow} \{0,1\}^\lambda$. 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 $(\mathsf{keys}, \mathrm{sid}, U_i, pk_i^{\mathrm{ENC}})$-message, to $\mathcal{F}_{\mathrm{INIT}}$ in the genesis round then $\mathcal{F}_{\mathrm{INIT}}$ 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, $\mathcal{F}_{\mathrm{INIT}}$ sends $(\mathsf{genblock}, sid, (\mathbb{C}_1, \eta_1))$ to U.

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

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

Proving *When receiving a message* $(\mathsf{prove}, sid, x, w)$*:* If $(x, w) \in \mathcal{L}$, query $\mathcal{A}$ for a proof string π. Record (x, π), and return π.

Proof Malleability *When receiving a message* $(\mathsf{maul}, sid, x, \pi)$ *from* $\mathcal{A}$*:* If for some π', (x, π') was recorded, record (x, π).

Proof Verification *When receiving a message* $(\mathsf{verify}, sid, x, \pi)$*:* If (x, π) was not recorded, query $\mathcal{A}$ for a witness w. If $(x, w) \in \mathcal{L}$, record (x, π). Finally if (x, π) is recorded.

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

$\mathcal{F}_{\mathrm{FwEnc}}$ is parameterized by, a security parameter κ, a set of parties $\mathcal{P}$, and a maximum delay Δ_{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 there exists a $U_q \in \mathcal{P}$, where $pk_q = pk$ and $U_q \in \mathcal{H}$, and $\tau < \tau_q + \Delta_{\mathrm{max}}$. 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 $\perp$.
  - 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 if $m \neq \perp$, 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 $\mathcal{H}$.

APPENDIX B
THE SIMULATOR

A. The Stage 1 Simulator

The first simulation stage of the Ouroboros Crypsinous simulator is based closely on the Ouroboros Genesis [1] simulator. The full simulator is omitted for brevity, and can be found in the full version of this paper.

Our stage 1 simulator differs only to deal with the difference between ideal and real transactions, in particular the following two differences exist: First, the simulator, instead of forwarding submitted honest transactions directly to the network, simulates executing the respective parties Submit operation. Second, to ensure adversarial transaction are correctly included in the ideal world, the simulator a) waits until the adversarial transaction is confirmed in the real world, and then b) extracts the transactions NIZK witness, uses this to construct the corresponding ideal transaction, and submits and immediately confirms it in the ideal world.

B. The Stage 2 Simulator

The second simulation stage restricts the leakage available to the simulator to that described in Figure 2. Again, details are left for the full version of the paper. The stage 2 simulator has challenges in the following two domains: a) when simulating the creation of transactions, it does not know their content, unless addressed to the adversary. b) when a party is corrupted, it must create the parties state after corruption. For both, the simulator makes use of the fact that all primitives used in transactions are themselves simulation secure. The simulator can create entirely simulated transactions, and on corruption, "open" them to a plausible value.

APPENDIX C
UC SPECIFICATION OF OUROBOROS CRYPSINOUS

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

Registration/Deregistration: *Initially, as in Ouroboros-Genesis, then call* $\mathsf{Initialization\text{-}Crypsinous}(U_p, sid, R)$.

Interacting with the Ledger ***(cf. Section VI-D):***
Upon receiving a ledger-specific input $I \in \{(\textsc{submit}, \ldots), (\textsc{read}, \ldots), (\textsc{maintain-ledger}, \ldots)\}$ 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** $I = (\textsc{submit}, \mathrm{sid}, (\textsc{public}, \textsc{transfer}) \,\|\, \mathsf{tx})$ **then** set invoke the protocol $\mathsf{SubmitXfer}(\mathsf{tx}, \mathcal{C}_{\mathsf{loc}}, \mathsf{log})$.
- **Else if** $I = (\textsc{submit}, \mathrm{sid}, \mathsf{tx})$ **then** set invoke the protocol $\mathsf{SubmitGeneric}(\mathrm{sid})$.
- **Else if** $I = (\textsc{generate}, \mathrm{sid}, \mathsf{tag})$ **then**
  - If $\mathsf{tag} = \textsc{coin}$, sample $sk_\tau^{\textsc{coin}} \xleftarrow{\$} \{0,1\}^{\ell}_{\mathsf{PRF}}$, and let $sk_{i+1}^{\textsc{coin}} = \mathsf{PRF}^{\mathsf{evl}}_{sk_i^{\textsc{coin}}}(1)$, for $i \in \tau, \ldots, \tau + R$. Let root be the root of the Merkle tree over $sk_\tau^{\textsc{coin}}, \ldots, sk_{\tau+R}^{\textsc{coin}}$, and $pk^{\textsc{coin}} \leftarrow \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}}(\tau)$. Insert the Merkle tree, and τ into $\mathfrak{C}_{\mathsf{free}}$, and return $pk^{\textsc{coin}}$.
  - If $\mathsf{tag} = \textsc{id}$, and this is the first query for $\textsc{id}$, send $(\mathsf{KeyGen}, \mathrm{sid})$ to $\mathcal{F}_{\mathrm{FwEnc}}$. Denote the response by

pk^{ENC}, record it, and then return it.

- Otherwise, return a uniformly random value in $\{0,1\}^{\kappa}$.

– **If** $I = (\textsc{maintain-ledger}, \mathrm{sid})$ **then** invoke protocol $\mathsf{LedgerMaintenance}(\mathcal{C}_{\mathsf{loc}}, \mathfrak{C}, U_p, \mathrm{sid}, k, s, R, f, \mathsf{log})$; if $\mathsf{LedgerMaintenance}$ halts **then** halt the protocol execution (all future input is ignored).

– **If** $I = (\textsc{read}, \mathrm{sid})$ **then** invoke protocol $\mathsf{ReadState}(k, \mathcal{C}_{\mathsf{loc}}, U_p, \mathrm{sid}, R, f, \mathsf{log})$.

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

A. The Staking Procedure

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

The following steps are executed in an $(\textsc{maintain-ledger}, \mathrm{sid})$-interruptible manner:

```
for $(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}) \in \mathfrak{C}$ do
    if $\mathbf{c}$ is not eligible for leadership then continue
    Send $(\mathsf{eval}, \mathrm{sid}_{RO}, \textsc{nonce} \,\|\, \eta_{ep} \,\|\, sl))$ to $\mathcal{G}_{\mathrm{RO}}$, and denote the response $\mu_\rho$.
    Send $(\mathsf{eval}, \mathrm{sid}_{RO}, \textsc{lead} \,\|\, \eta_{ep} \,\|\, sl))$ to $\mathcal{G}_{\mathrm{RO}}$, and denote the response $\mu_y$.
    Lookup $sk^{\mathrm{COIN}}_{\mathbf{c},\tau}$, $\mathsf{root}_{\mathbf{c}}$, and $\tau_{\mathbf{c}}$ in $\mathfrak{C}_{\mathsf{free}}$ corresponding to $pk^{\mathrm{COIN}}_{\mathbf{c}}$.
    Let $\rho \leftarrow \mu_\rho^{\mathsf{root}_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}}}; y \leftarrow \mu_y^{\mathsf{root}_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}}}$
    if $y < \mathsf{ord}(G)\phi_f(v_{\mathbf{c}})$ then
        Compute $\mathsf{st}$ as in Ouroboros Genesis [1].
        Set $ptr \leftarrow H(\mathsf{head}(\mathcal{C}_{\mathsf{loc}}))$; $h \leftarrow H(\mathsf{st})$
        Set $\rho_{\mathbf{c}'} \leftarrow \mathsf{PRF}^{\mathsf{evl}}_{\mathsf{root}_{\mathbf{c}}}(\rho_{\mathbf{c}})$; $sn_{\mathbf{c}} \leftarrow \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{\mathbf{c}}}(\rho_{\mathbf{c}})$
        Set $(cm_{\mathbf{c}'}, r_{\mathbf{c}'}) = \mathsf{Comm}(pk^{\mathrm{COIN}}_{\mathbf{c}} \,\|\, v_{\mathbf{c}} \,\|\, \rho_{\mathbf{c}'})$.
        Let $\vec{\mathsf{stx}}_{\mathsf{ref}}$ be, in order, the list of leadership transactions made by $U_p$ not in $\mathcal{C}_{\mathsf{loc}}$.
        Let $\mathsf{root}$ 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 $\mathsf{path}$ be the path to $cm_{\mathbf{c}}$ in the same Merkle tree.
        Let $\mathsf{path}_{\mathbf{c}}$ be the Merkle path to $sk^{\mathrm{COIN}}_{\mathbf{c},\tau}$ in the secret-key Merkle tree.
        Let $\mathbf{x} = (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, \eta_{ep}, sl, \rho, h, ptr, \mu_\rho, \mu_y, \mathsf{root})$.
        Let $\mathbf{w} = (\mathsf{path}, \mathsf{root}_{\mathbf{c}}, \mathsf{path}_{\mathbf{c}}, \tau_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}, r_{\mathbf{c}'})$.
        Send $(\mathsf{prove}, \mathrm{sid}, \mathbf{x}, \mathbf{w})$ to $\mathcal{F}^{\mathcal{L}_{\textsc{lead}}}_{\mathrm{NIZK}}$, and denote the response $\pi$.
        Let $\mathsf{tx}_{\mathsf{lead}} = (\textsc{lead}, \vec{\mathsf{stx}}_{\mathsf{ref}}, (cm_{\mathbf{c}'}, sn_{\mathbf{c}}, ep, sl, \rho, h, ptr, \pi))$.
        Set $B \leftarrow (\mathsf{tx}_{\mathsf{lead}}, \mathsf{st}); \mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \,\|\, B$.
        Update $\mathbf{c}$: $\mathfrak{C} \leftarrow (\mathfrak{C} \setminus \{(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}) \cup \{(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}'}, r_{\mathbf{c}'}, v_{\mathbf{c}})\}$
        Send $(\textsc{multicast}, \mathrm{sid}, \mathsf{tx}_{\mathsf{lead}})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
        Send $(\textsc{multicast}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ and proceed from here upon next activation of this procedure.
        break
    end if
end for
while A $(\textsc{clock-update}, \mathrm{sid}_C)$ has not been received during the current round do
    Give up activation. Upon next activation of this procedure, proceed from here.
end while
```

B. The Ledger Maintenance Procedure

Protocol $\mathsf{LedgerMaintenance}(\ldots)$

The following steps are executed in an $(\textsc{maintain-ledger}, \mathrm{sid})$-interruptible manner:

```
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 $\textsc{welcome}$.
if $\textsc{welcome} = 1$ then proceed as in [1].
for transaction $\mathsf{tx} \in (\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$ do
    if $\mathsf{tx}$ is a transfer transaction then
        Attempt to decrypt each new ciphertext $c$ by sending $(\mathsf{Decrypt}, \mathrm{sid}, c)$ to $\mathcal{F}_{\textsc{FwEnc}}$. Receive the response $m$.
        if $m = (pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}}) \wedge cm_{\mathbf{c}} \in \mathsf{tx}$ then
            if $\nexists (sk^{\mathrm{COIN}}_{\mathbf{c}}, \tau) \in \mathfrak{C}_{\mathsf{free}} : \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}_{\mathbf{c}}}(\tau_{\mathbf{c}}) = pk^{\mathrm{COIN}}_{\mathbf{c}}$
                then continue
            Let $\mathfrak{C}_{\mathsf{cnd}} \leftarrow \mathfrak{C}_{\mathsf{cnd}} \cup \{(pk^{\mathrm{COIN}}_{\mathbf{c}}, \rho_{\mathbf{c}}, r_{\mathbf{c}}, v_{\mathbf{c}})\}$.
            Let $\mathsf{log} \leftarrow \mathsf{log} \,\|\, (\mathsf{tx}, \textsc{receive}, (pk^{\mathrm{COIN}}_{\mathbf{c}}, v_{\mathbf{c}}))$.
        end if
    else if $\mathsf{tx}$ is a generic transaction then
        Attempt to decrypt each subtransaction ciphertext $c$ by sending $(\mathsf{Decrypt}, \mathrm{sid}, c)$ to $\mathcal{F}_{\textsc{FwEnc}}$. Receive the response $m$.
        if $m \neq \bot$ then $\mathsf{log} \leftarrow \mathsf{log} \,\|\, (\textsc{plaintext}, c, m)$
    end if
end for
for $(sk^{\mathrm{COIN}}_{\mathbf{c}}, \tau_{\mathbf{c}}) \in \mathfrak{C}_{\mathsf{free}}$ do
    if $\exists sk^{\mathrm{COIN}}_{\mathbf{c}} \in \mathfrak{C}_{\mathsf{cnd}}$ whose transaction $\in \mathcal{C}^{\lceil k}_{\mathsf{loc}}$ then
        Move such candidates to $\mathfrak{C}$.
    end if
    Erase $sk^{\mathrm{COIN}}_{\mathbf{c},\tau}$.
end for
Use the clock to update $\tau, ep \leftarrow \lceil \tau/R \rceil$, and $sl \leftarrow \tau$.
Set $\mathsf{buffer} \leftarrow \mathsf{buffer} || (\mathsf{tx}_1, \ldots, \mathsf{tx}_k)$, $t_{\mathrm{on}} \leftarrow \tau$, $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots,$ and $\mathcal{C}_M\}$
Invoke Protocol $\mathsf{SelectChain}(\mathcal{C}_{\mathsf{loc}}, \mathcal{N}, k, s, R, f)$.
Update $\mathcal{F}_{\textsc{FwEnc}}$ as many times as necessary for its time to be a least $\tau - k$.
if $t_{\mathrm{work}} < \tau$ then proceed as in [1].
```

Appendix 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 $(\mathbf{x}, \mathbf{w}) \in \mathcal{L}_{\textsc{lead}}$ iff all of the following hold:

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

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:

- $\mathbf{x} = (\{cm_{\mathbf{c}_3}, cm_{\mathbf{c}_4}\}, \{sn_{\mathbf{c}_1}, sn_{\mathbf{c}_2}\}, \mathsf{root})$
- $\mathbf{w} = ((\mathsf{root}_{\mathbf{c}_1}, \tau_{\mathbf{c}_1}, \rho_{\mathbf{c}_1}, r_{\mathbf{c}_1}, v_1, \mathsf{path}_1), (\mathsf{root}_{\mathbf{c}_2}, \tau_{\mathbf{c}_2}, \rho_{\mathbf{c}_2}, r_{\mathbf{c}_2}, v_2, \mathsf{path}_2), (pk^{\text{COIN}}_{\mathbf{c}_3}, \rho_{\mathbf{c}_3}, r_{\mathbf{c}_3}, v_3), (pk^{\text{COIN}}_{\mathbf{c}_4}, \rho_{\mathbf{c}_4}, r_{\mathbf{c}_4}, v_4))$
- $\forall i \in \{1, 2\} : pk^{\text{COIN}}_{\mathbf{c}_i} = \mathsf{PRF}^{\mathsf{pk}}_{\mathsf{root}_{\mathbf{c}_i}}(\tau_{\mathbf{c}_i})$
- $\forall i \in \{1..4\} : \mathsf{DeComm}(cm_{\mathbf{c}_i}, pk^{\text{COIN}}_{\mathbf{c}_i} \parallel v_i \parallel \rho_{\mathbf{c}_i}, r_{\mathbf{c}_i}) = \top$
- $v_1 + v_2 = v_3 + v_4$
- $cm_{\mathbf{c}_1}$ is in root at path path_1.
- $cm_{\mathbf{c}_2}$ is in root at path path_2, or $v_2 = 0$ and $\mathsf{root}_{\mathbf{c}_2} = \rho_{\mathbf{c}_2} = \mathsf{PRF}^{\mathsf{zdrv}}_{\mathsf{root}_{\mathbf{c}_1}}(\rho_{\mathbf{c}_1})$.
- $\forall i \in \{1, 2\} : sn_{\mathbf{c}_i} = \mathsf{PRF}^{\mathsf{sn}}_{\mathsf{root}_{\mathbf{c}_i}}(\rho_{\mathbf{c}_i})$

Appendix E
Security Analysis (Continued)

Proof of Lemma 1: To begin with, we note that if $\mathsf{Lkg} = \mathsf{Lkg}_{\mathsf{id}}$, the simulator $\mathcal{S}_1$, described in detail in Section B-A receives the same information in our private ledger $\mathcal{G}_{\text{PL}}$, as it would in the standard ledger $\mathcal{G}_{\text{LEDGER}}$ of [1]. Specifically, it gets to see all transactions, in the buffer and the state in full, as well as the sequence of honest inputs $\vec{\mathcal{I}}^T_H$ (we note this hides transactions, but these are revealed separately). We further note that the interface of $\mathcal{G}_{\text{PL}}$ to the environment is similar to that of $\mathcal{G}_{\text{LEDGER}}$, specifically that it is a stricter interface – it reveals less information to the environment, and restricts the possible actions of the environment by forcing a format on transactions. Together, it is clear to see that if a protocol realizes $\mathcal{G}_{\text{LEDGER}}$, it can also realize the leaky $\mathcal{G}_{\text{PL}}$. We can strengthen this result, and argue that it is even equivalent to a ledger with a different validation predicate. Concretely, we note the reduction from [1] still holds if a mapping between $\mathcal{G}_{\text{PL}}$ and $\mathcal{G}_{\text{LEDGER}}$ transactions (as in $\mathsf{Ouroboros\text{-}Crypsinous}$) is used. In particular, we note that parties compute some function $f(\mathsf{tx})$ for each incoming (private ledger) transaction tx, and use this similarly to a standard ledger transaction. Likewise, on a read request, parties first get the underlying sequence of transactions, and then apply an inverse mapping $f^{-1}(U_p, \mathsf{tx})$. We note that storing mapped transactions instead of real transaction can be considered merely a matter of representation, as long as the properties required of them are preserved, whenever transactions are accessed. Specifically, $\mathsf{BlindTx}(\{U_p\}) = f^{-1}(U_p) \circ f$, and the validity predicate on the mapped transactions must behave the same as the validity predicate on the original transactions. If this is the case, the mapping is transparent to the ledger functionality itself, and is therefore equivalent to having no mapping at all. ■

Proof of Lemma 2 (sketch): We note that the combination of NIZKs with an erasable secret key, $sk^{\text{COIN}}_{\mathbf{c}}$, effectively provide a signature of knowledge of the given secret key. While the identity of the corresponding public key is never revealed, we note that [1] does not make use of the identity except to check it is the same as the party winning the lottery. This property is guaranteed by a proof of LEAD as well. Finally, we note that the erasure properties of $sk^{\text{COIN}}_{\mathbf{c}}$ leaves correspond to updating the $\mathcal{F}_{\text{KES}}$ secret key after signing. Likewise, we note that the usage of MUPRFs in NIZKs provides the same verifiable randomness as VRFs did in [1], again with the exception of not revealing the public key used – which again is fine, as it is not required except to check that it is the same identity as used in the signature.

Combined with the fact that LEAD verifies the same leadership conditions as in the leadership proofs of [1], we note that it is possible to create an Ouroboros Genesis leadership proof for a party, if and only if it is possible to create an Ouroboros Crypsinous leadership transactions for a corresponding coin. ■

Proof of Lemma 3 (sketch): We note the implicit mappings between $\mathcal{G}_{\text{PL}}$ and $\mathcal{G}_{\text{LEDGER}}$ transactions must be quasi-bidirectional – converting back from ledger transactions into $\mathcal{G}_{\text{PL}}$ transactions should be equivalent to $\mathsf{BlindTx}$. Further, the mapped transactions should verify in the equivalent $\mathcal{G}_{\text{LEDGER}}$ if and only if the original transactions verify in $\mathcal{G}_{\text{PL}}$. For generic transactions, this is given by the security of $\mathcal{F}_{\text{FwEnc}}$. As the mapped transactions have encryptions, they can be read only by the intended recipients. The effect is simply that if parties retain the readable components of a transaction, they obtain exactly the blinded version of it. For transfer transactions, things are more complex, as there is a stateful validation involved.

We prove this by induction over a sequence of $\mathcal{G}_{\text{PL}}$ transactions, and the corresponding $\mathcal{G}_{\text{LEDGER}}$ transactions. Specifically, we maintains that in both worlds, a notion of coins exists, which are kept and used by parties. We note that initially – and by induction, at any time – a bijective relation exists between these sets of coins, mapping each "ideal" $\mathcal{G}_{\text{PL}}$ coin to a "real" $\mathcal{G}_{\text{LEDGER}}$ coin and visa-versa. We demonstrate that this invariant is preserved across honest transfer and leadership transactions, by the security properties of NIZKs and PRFs. Further, we argue that the simulator $\mathcal{S}_1$ can construct "ideal" transactions corresponding to any possible adversarial transaction, that also preserve the invariant. We note that as the set of coins changes iff a transaction is valid, this invariant directly implies the equivalence of the validation predicates. ■

Proof of Theorem 1: 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 Corollary 1. 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, we note that Ouroboros Crypsinous allows ID generation, which are generated as PRF outputs of a PRF seeded with a random, secret value, which will lead to unique IDs for honest parties with overwhelming probability. We conclude that $\mathsf{Ouroboros\text{-}Crypsinous}$ realizes $\mathcal{G}_{\text{PL}}$ with $\mathcal{S}_1$, under the leakage predicate $\mathsf{Lkg}_{\mathsf{id}}$. ■

Proof of Lemma 4: We note there are four primitives that are simulated in simulated transactions: Commitments, NIZKs, $\mathcal{F}_{\text{FwEnc}}$ encryptions, and serial numbers. Due to the simulation security of NIZKs, and the equivocality of the commitments, we know they are indistinguishable from real NIZKs and commitments respectively. We note for $\mathcal{F}_{\textsc{FwEnc}}$, 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. For serial numbers, we note that if a coin is honestly generated, once its spent its nonce ρ_{c} is erased, and irretrievable. Therefore, an adversary corrupting a party will be unable to reconstruct the serial number, and it is indistinguishable from the random, simulated, serial number. By contrast, if the nonce is adversarially generated, the simulator is informed *which* coin is being spent, and inserts the appropriate *correct* serial number. As transactions consist of these primitives, and the simulator 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. ■

Proof of Lemma 5 (sketch): We note that a party's state consists of the coin bookkeeping variables $\mathfrak{C}$, $\mathfrak{C}_{\mathsf{free}}$, and $\mathfrak{C}_{\mathsf{cnd}}$, log and the state from [1], consisting of the local chain $\mathcal{C}_{\mathsf{loc}}$, and buffer. We note that while on corruption, the simulator can extract ideal world coins from $\mathcal{G}_{\text{PL}}$, and map them to their real-world equivalent. Likewise, even before corruption, the simulator knows the local chains and buffer of all parties due to its control of the network.

What remains to show is that the simulator can not just construct a plausible honest party's state, but that this state is consistent with the ledger as is. In particular, we note that it must be consistent with transactions the adversary has observed so far. We note that this is handled by the erasure of secret keys, the equivocality of commitments, and the forward-security of $\mathcal{F}_{\textsc{FwEnc}}$. In particular, we note that transactions expose four types of values: PRFs of secret keys, in the form of serial numbers, nonces, and the fact that the leadership target was met, commitments to public keys, encryptions of coin nonces, and NIZK proofs with secret keys in their witness. We note the first is secure due to the erasure of the secret keys after generating the PRF, the second is secure due to our usage of equivocality, the third is secure by the forward-security of $\mathcal{F}_{\textsc{FwEnc}}$, and the last is secure due to the zero-knowledge property of the NIZK. ■

Proof of Theorem 2: We conclude from Theorem 1, Lemmas 4 and 5, and the fact that $\mathcal{S}_1$ and $\mathcal{S}_2$ differ only in simulating transactions and corruption, that Theorem 2 holds. ■

Appendix F
Key-Private Forward-Secure Encryption

We extend the notion of forward-secure encryption (FSE) with a notion of *key-privacy*, described in detail in the full version of this paper. We note that, while this definition itself is novel, existing schemes already 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, we note that [10]'s construction of FSE from HIBE is straightforward, with the ciphertexts simply being the underlying HIBE scheme's ciphertexts. We note therefore, if some property holds about HIBE ciphertexts, it holds about FSE ciphertexts in our construction. The core argument of the anonymity of [7], arising from Lemmas 8 and 9, is the indistinguishably of ciphertexts with random group elements – and therefore their independence of the encrypting identity. We note that the ciphertexts' pseudorandomness implies a stronger notion than just anonymity – the ciphertext also does not reveal any information about the HIBE public key. In particular, as ciphertexts are indistinguishable, our enhanced security game given in the full version of this paper is satisfied.

We note this construction is logarithmic to the number of time slots. As this is bounded exponentially by the security parameter κ, we note the use of this forward-secure encryption has a linear increase in cost with respect to the security parameter compared to standard encryption.

a) Lifting to a UC-Protocol: A key-private and forward-secure encryption scheme, when used in the $\mathcal{F}_{\textsc{KeyMem}}$-hybrid model, can be used to realize $\mathcal{F}_{\textsc{FwEnc}}$. The realization employs the Fujisaki-Okamoto transform [15] to make the encryption scheme CCA security. We leave the precise description, and proof of this protocol to the full version of this paper, however the general premise is to utilize $\mathcal{F}_{\textsc{KeyMem}}$ to ensure the adversary never has access to secret keys of ciphertext which cannot be simulated, and to leverage the key-privacy to show that simulated ciphertexts are indistinguishable from real ciphertexts.

Appendix G
Commitment Notation

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. $\mathsf{Init}_{\mathsf{comm}}$ takes the function of the CRS assumption, by generating a public key pk^{COMM} which is given as an argument to Comm and DeComm. In addition to satisfying the traditional commitment properties, of binding, hiding, and correctness, the scheme also satisfies equivocality. $\widehat{\mathsf{Init}}_{\mathsf{comm}}$ provides an equivocation key in addition to pk^{COMM}. This equivocation key can be used to break 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.

We write $(cm, r) \leftarrow \mathsf{Comm}(m)$ to create the commitment cm for message r, 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, such that $\mathsf{DeComm}(cm, m, r) = \top$. In all these, we leave the public key pk^{COMM} implicit, as is assumed to be globally known via the CRS.