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

1The 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 naive approach of simply paying oneself with a new coin does not work here, as the same coin should be able to be elected multiple times in a sequence of PoS invocations without leaving any evidence in the ledger.

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

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, 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 $\Delta$ 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 $\Delta$. Honest receivers cannot tell when a message will arrive as they know neither when the message was sent nor the delay $\Delta$. However, as in [17], our protocol is implicitly aware of an overestimate $\Delta_{\text{max}}$ of the actual (unknown) network delay $\Delta$. However, this $\Delta_{\text{max}}$ is not used in the message passing; instead the protocol proceeds in an optimistic manner once messages are received (after at most $\Delta$ rounds from sending), and $\Delta_{\text{max}}$ is only used in the staking procedure to determine the leader(s) of each slot.

Similarly to [2], 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 $\kappa$) 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 composable security of the statement. We follow the paradigm of [2] to capture these assumptions without compromising composable security. 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), andequivocal commitments. We describe ideal functionalities capturing NIZK and key-private forward-secure encryption, and refer to their UC implementates. 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 \( F_{\text{NIZK}} \), and protocol of [22]. This functionality allows generating proofs \( \pi \) that a statement \( x \) is in a (fixed) NP language \( \mathcal{L} \), with a witness \( w \). 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 \( \Delta_{\text{max}} \) into the future are protected. Nonetheless, even this notion seems too strong to be implementable against a fully adaptive adversary in a \( \Delta \)-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 \( \tau \) so that it can be decrypted with key \( sk_{\tau}^{\text{ENC}} \), and sent over to the receiver. Upon receiving it, the receiver can decrypt it (using \( sh_{\tau}^{\text{ENC}} \)) and immediately update the key to \( sk_{\tau+1}^{\text{ENC}} \) for the next round (and erase \( sk_{\tau}^{\text{ENC}} \)). 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 \( \Delta \), and bounded only by \( \Delta_{\text{max}} \), 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 \( sk_{\tau}^{\text{ENC}} \) at time \( \tau \) does not receive \( sk_{\tau}^{\text{ENC}} \), but rather the key \( sk_{\tau + \Delta_{\text{max}}}^{\text{ENC}} \), which this party would hold in time \( \tau + \Delta_{\text{max}} \), 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 \( \Delta_{\text{max}} \) 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 \( \{ f_k \}_{k \in \mathbb{K}} \) such that \( f_k : X \rightarrow Y \) for all \( k \in \mathbb{K} \). The usual PRF security requires that any PPT distinguisher \( D \) with an oracle cannot tell the difference between an oracle \( f_k(\cdot) \), for a randomly selected \( k \) and a truly random function over \( X \rightarrow Y \). 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 \( A \) and \( x \in X, T \in \mathbb{N} \), the probability of the event \( \Pr[f_k(x) < T|x \notin Q_H] \) equals \( T/2^k \) where \( A(1^k) = k \), and \( Q_H \) is the set of queries of \( A \) 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 \( H : \{0,1\}^* \rightarrow \langle g \rangle \) 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 \( \kappa \)-bit string. Then we define \( f_k(m) \rightarrow H(m)^k \) 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 \( \langle g^y, H(m)^k, H(m)^{yk} \rangle \) 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, \( \Pr[H(x)^k < T] \leq \sum_{y \in T} \Pr[H(x)^k = y] = T \Pr[H(x) = y^{1/k}] \leq T/2^k \) in the conditional space \( x \notin Q_H \).

D. Equivocal Commitments

We make use of a standard non-interactive equivocal commitment scheme, \((\text{Init}_{\text{comm}}, \text{Comm}, \text{DeComm}, \text{Init}_{\text{comm}}, \text{Comm}, \text{Equiv})\), 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 \( BTX := (tx, \text{txid}, \tau_L, U_s) \), which includes several useful metadata: \( \text{txid} \) is a unique identifier for this transaction, \( \tau_L \) 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 \( tx \overset{\Delta}{=} (\text{stx}_1, \text{stx}_2, \ldots, \text{stx}_l) \).
Each sub-transaction consists of a recipient public key \( pk_r \), and an arbitrary message \( x \), that is \( stx \triangleq (pk_r, x) \). 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 \( \bot \). Concretely, we introduce *blinding* functions \( \text{BlindTx} \) and \( \text{Blind} \), described below, which hide parts of the ledger from read requests. Finally, we parameterize the private ledger with a general purpose leakage algorithm, \( \text{Lkg} \), which the adversary is permitted to query. An overview of the functionality can be found in Figure 1.

- \( \text{BlindTx} \) takes as input an annotated transaction \( BTX = (tx, \tau_L, U_s) \), a set of parties \( P \), 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 \( U_p \in P \). An adversarial version of \( \text{BlindTx} \), \( \text{BlindTx}_A \), additionally returns the time of submission, \( \tau_L \), and the submitter \( U_s \).\(^3\) We make use of the commonly used higher-order function \( \text{map} \), 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.

\[
\text{BlindSTx}(P, \ids, (pk, stx)) \triangleq \text{if } pk = \text{PUBLIC} \vee \\
\exists U_p \in P : (U_p, \id, pk) \in \ids \\
\text{then } (pk, stx) \text{ else } (\bot, [stx])
\]

\[
\text{BlindTx}(P, \ids, (tx, \tau, U)) \triangleq \\
\text{(map}(\text{BlindSTx}(P, \ids), tx), \tau, U)
\]

\[
\text{BlindTx}_A(P, \ids, (tx, \tau, U, s)) \triangleq \\
\text{(map}(\text{BlindSTx}(P, \ids), tx), \tau, U)
\]

- \( \text{Blind} \) is similar to \( \text{BlindTx} \) 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.

\( \text{Blind}_A \) is defined the same as \( \text{Blind} \), but with calls to \( \text{BlindTx} \) replaced with calls to \( \text{BlindTx}_A \).

In our system, we permit the leakage \( \text{Lkg}_{\text{read}} \) (Figure 2), which effectively simulates the protocols leadership election, and leaks the winning party. Specifically, for each time \( \tau \), 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 \( \text{Lkg}_{\text{read}} \) 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 \( \phi_f \) of their stake, where \( \phi_f \) is the independent aggregation function described in [13], [1].

In addition to this, we note Zerocash-style protocols will allow an adaptively corrupting adversary to compute the serial number of coins it sent to an honest party after corrupting them. As the serial number is by necessity committing, the simulator must know when such adversarially sent coins are spent, to ensure the consistency of the simulation. For this reason, we also leak the points adversarially sent coins are spent.

In a preliminary step of our analysis we also utilize a leak-
Figure 2. The leakage function for Ouroboros Cryptinosus

VI. THE OUROBOROS-CRYPTINOSUS PROTOCOL

In this section we provide a detailed description of our protocol Ouroboros-Cryptinosus as a synchronous (G)UC pro-
tocol. 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 (PUBLIC, TRANSFER) to be transfer transactions. While it may appear sufficient to have ideal-world transfers appear as something like “give 0.05 of Alice’s stake to Bob”, our realization of transfers using a Zerocash-like [4] design introduces some subtleties that need to be reflected in the ideal world. Specifically, we will require parties to specify which coins they are attempting to spend. Specifically, as in Zerocash, two coins are burned, and two coins created, in any transfer. As a special case, as our protocol has no other minting functionality, we allow a zero-value coin to be burned in place of the second coin. Formally, the transactions have the following form: ((PUBLIC, TRANSFER), (pk_ε, c_4), (pk_ε, c_1, c_2, c_3)), where c_4 is a coin and c_1, c_2, c_3 are, respectively, the IDs of the recipient’s secret keys, the IDs of the sender’s public keys, and the IDs of the coins. The validation predicate ensures the total value is preserved across the transfer, and that an ID is only spent by its generating party.

In the real world, the design looks slightly different, following the approach of Zerocash [4]. Specifically, parties locally maintain, for each coin c, nonces, r_c, and commitment openings, e_c, to their coins. In order to spend a coin, they reveal the deterministically derived serial number, sn_ε_c, as well as prove the existence of a valid commitment, cm_ε_c, somewhere in a Merkle tree of coin commitments. Like Zerocash, newly created coins are encrypted with the recipient party’s public key, and the sending party is unable to spend them as it would require the recipient’s private key to correctly generate the coin’s serial number. One key difference is the design of addresses, corresponding to the Ideal-world IDs. Parties will generate a new coin public/secret key pair when given a GENERATE query, and will update their secret key after spending a coin with it.

To become a leader at a time τ, parties must prove knowledge of a path in a local Merkle tree of secret keys sk_ε_coin, labeled with τ. This path is then erased by the party, to ensure leadership proofs cannot be re-made for past slots. This Merkle tree is created during key generation, with the coin’s public key being derived from the Merkle tree’s root, and the time of key generation. Each leaf is a PRF of the previous leaf, to reduce storage costs. We employ standard space/time trade-offs by keeping the top of the tree stored, and recomputing parts of the bottom of the tree as needed. It is parameterized by the number of leaves R, which we leave as a system parameter, although we note it could also be defined per-user.

A users public key is derived from the root of the Merkle tree, root, and the time it was created, τ. It is eligible for leadership so long as there are still paths in the tree to prove the existence of, after which the coin must be refreshed, by spending it. We stress that this is a rare occurrence, as the assumption of honest majority relies on coins not only being held by honest parties, but also being eligible for leadership.

The protocol will take ideal transactions as an input, and construct a corresponding Zerocash-style transaction in the real world. This transaction is then broadcast as usual in a blockchain protocol. On a READ request, the irrelevant information is not returned, and only the information corresponding to the original ideal-world transaction is returned back to the requester. In addition to transfers, we note that
other types of transaction are accepted in the ideal world. We note that these are not validated, however, making the real-world equivalent far simpler to construct. Specifically, we encrypt each subtransaction with the public key of the party it is addressed to. On a READ request, the ciphertexts that the requesting party can decrypt are decrypted, and all others are replaced with ⊥.

B. Protocol overview

The protocol Ouroboros-Crypsinous assumes as hybrids a network \( \mathcal{F}_{S_{\text{MC}}} \), a non-interactive-zero-knowledge scheme \( \mathcal{F}_{\text{NIZK}} \), a forward-secure encryption scheme \( \mathcal{F}_{\text{ENC}} \), 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 \( c \), 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 \( \eta_{ep} \), by being evaluated at the slot index \( sl \) and \( \eta_{ep} \), seeded with the “winning coin’s secret key” \( \text{root} \). \( \rho_{\text{ep}} \) is generated similarly to Ouroboros Genesis – it is initially supplied through the CRS, then for subsequent epochs, it is sampled in a maliciously unpredictable way from “randomness contributions” \( \rho \) 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 \( T_e \)—which depends on \( c \)’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 \( \pi \), signing the rest of the block content, with the knowledge of the nonce \( \rho_{\text{ep}}, \eta_{\text{ep},sl} \) for the slot \( sl \) the leadership transaction is for, proving that the nonce and secret key correspond to some unspent coin commitment \( cm_c \). 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: \( \rho_{\text{ep}} = \text{PRF}_{\text{root}}(\rho_{\text{ep}}) \). 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 \( \rho_{\text{ep}} \) and only maintain \( \rho_{\text{ep}} \) 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 \( C_{loc}^i \)—\( U_p \)'s local view of the blockchain.\(^4\) Such a local blockchain is a sequence of blocks \( B_i \) \((i \geq 0)\) where each \( B \in C_{loc} \) has the following format: \( B = (B_{lead}, \text{st}) \); where \( B_{lead} = (\text{LEAD}, \text{stXref}, \text{stXproof}) \), and \( \text{stXproof} = (c_{\text{e}1}, \text{sn}_{\text{e}1}, \text{ep}, sl, c, \rho, h, \rho^4, \pi) \). Here, \( \text{st} \) 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, \((c_{\text{e}1}, \rho^4, \pi) \) is the commitment of the newly-created coin, \( \text{sn}_{\text{e}1} = \text{PRF}_{\text{root}_{\text{e}1}}(\rho^4) \) is the serial number of the coin \( c \), which is revealed to demonstrate the coin has not been spent. We define \( \rho = \mu^{\text{root}_{\text{e}1}}(\rho^4) \), where \( \mu \in G_{RO} \) evaluated at \( \text{NONCE} \parallel \text{sl} \). \( \rho \) is the randomness contribution to the next epoch’s randomness, \( \rho^4 \parallel \pi \) is the hash of the previous block, and \( \pi \) is a NIZK proof of the statement \( \text{LEAD} \) (defined in Appendix D). The component \( \text{stXref} \) 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 \( B_{lead} \), transfer transactions \( B_{txr} \), 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 (PUBLIC), or a specific party. The corresponding real-world transaction is a vector of the same subtransactions, which are either directly the content of the ideal world transaction, in the case of a transaction addressed to PUBLIC, or an encryption of the content using \( F_{\text{FwEnc}} \), to the party specified as the recipient. Upon reading the state, parties attempt to decrypt ciphertexts, and failing that, replace it with \( \bot \). To disambiguate transactions, we prefix generic transactions with the label 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 \( B_{txr} = ((\text{PUBLIC}, \text{TRANSFER}), (pk_r, (id_4 \parallel v_4), (pk_a, (id_1 \parallel v_1), (id_2 \parallel v_2), (id_3 \parallel v_3)))) \) — with a Zerocash-like construction. We define a real transfer transaction to be: \( B_{txr} = (\text{TRANSFER}, \text{stXproof}, c_r) \), where \( \text{stXproof} = (\{cm_{e1}, cm_{e4}\}, \text{sn}_{e1}, \text{sn}_{e2}, \text{root}, \pi) \), and \( c_r \) is a \( F_{\text{FwEnc}} \) encryption for the slot the transaction was submitted in \( \text{stXproof} = (pk_r, \text{root}, \pi) \). Similar to leadership transactions, \((cm_{e1}, \text{root}) = \text{Comm}(pk_{e1} \parallel \text{ve}_1 \parallel \rho_{e1}) \), and \((cm_{e2}, \text{root}) = \text{Comm}(pk_{e2} \parallel \text{ve}_2 \parallel \rho_{e2}) \). \( \text{sn}_{e1} \) and \( \text{sn}_{e2} \) are revealed to spend the coins \( c_1 \) and \( c_2 \) respectively, and \( \pi \) proves the statement XFER (defined in Appendix D), specifically proving the existence of \( cm_{e1} \) and \( cm_{e2} \), 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 \( F_{\text{FwEnc}} \) implies that parties will not be able to decrypt ciphertexts addressed to them indefinitely, however they are still required to respond with the corresponding ideal-world information to \text{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 \((\text{root}_{e1}, \rho_{e1}, \rho_{e2}, \pi) \).

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 \((\text{SUBMIT}, \text{sid}, tx)\) is used for sending a new transaction to the ledger. The party maps \( tx \) to a corresponding \( B_{txr}^{\text{real}} \), which is stored in the parties local transaction buffer, and multicast to the network.
- The command \((\text{GENERATE}, \text{sid}, \text{tag})\) is used for creating new addresses, which can be used by other parties to transfer funds to this current party (if \( \text{tag} = \text{COIN} \)), and for initially creating a parties public keys (if \( \text{tag} = \text{ID} \)).
- The command \((\text{READ}, \text{sid})\) is used for the environment to ask for a read of the current ledger state. On receipt, the party maps each transaction \( \text{st}^{\text{-1}} \) to its ideal-world equivalent, and returns this ideal-world chain.
- The command \((\text{MAINTAIN-LEDGER}, \text{sid})\) 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 \( \tau \), 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 \text{LedgerMaintenance}, a detailed specification of the protocol \text{ReadState} for answering requests to read the ledger’s state,
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 $F_{\text{FWENC}}$. It receives any initial stake it may have as a single coin from $F_{\text{INIT}}$. 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, $c$, is both eligible for leadership in $ep$, and a PRF-value depending on $sl$ and the coin nonce $\rho_c$ and secret key $\text{root}_c$, is smaller than a threshold value $T_c$. 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 $\text{STX}_{\text{Get}}$.

Each coin $c$'s value $v_c$ induces a relative stake for the coin, $\alpha_c$. We use the same function $\phi_f(\alpha_c)$ as [1] to determine the probability of a coin winning the leadership election, with the corresponding threshold, $T_c = \text{ord}(G)\phi_f(\alpha_c)$, where $G$ is the target group of our MUPRFs. We note that due to the independent aggregation property of $\phi_f$, 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 from 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 $C_{\text{loc}}$ 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 $\text{tx}$ is mapped, if it is addressed to a party $U_r$, to a ciphertext encrypted with $F_{\text{FWENC}}$, party $U_r$’s public key, and the current time $\tau$. 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 (READ,sid) that is used to read the current contents of the state. Note that in the ideal world, the result of issuing such a command is for the ledger to output a (long enough prefix) of the ideal-world state of the ledger, with parts the party does not have access to being hidden. As the format of real-world transactions differs, we need to invert the map from real transactions to the corresponding ideal transactions. For generic transactions, this is a little tricky, as the use of forward-secure encryption implies that the information associated with the transaction in the ideal world is erased in the real world. To circumvent this, parties maintain 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 $C_1$. 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 \( c_1 \) and \( c_2 \) 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 \( c_2 \) is \( \bot \) 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 \( \text{STX}_{\text{chng}} \) to their own public key – i.e. the first value generated by \( \text{(GENERATE, sid, ID)} \) by the party. If the sending party is corrupted, it may spend the coins of other corrupted parties. If other transactions in the \( \text{buffer} \) 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 \( v_2 \) is recorded for \( c_3 \), and respectively with \( v_4 \), and \( c_4 \). The values spent are erased from the values lists of \( c_1 \) and \( c_2 \)’s IDs (with the exception of the id \( \bot \)).

b) Real World Validation: The real world validation predicate maintains three sets, the sets of coin commitments \( C_{\text{spend}} \), \( C_{\text{lead}} \) for spending and leadership respectively, initialized to the initial set of coin commitments \( C_1 \), and the set of spent serial numbers \( S \), initialized to \( \emptyset \). 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 \( \text{STX}_{\text{ref}} \) are valid adopted leadership transactions, and the NIZK proof is valid with respect to the Merkle root of the current tree, with these adopted transaction inserted, as well as \( \eta_{\text{cpr}} \), 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 \( S \). 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, \( \text{ptr} \) must be the hash of the previous block, and \( h \) must be the hash of the remaining transactions. After it is successfully validated, \( \mathcal{S} \leftarrow \mathcal{S} \cup \{ sn \} \), \( C_{\text{lead}} \leftarrow C_{\text{lead}} \cup \{ cm \} \), \( C_{\text{spend}} \leftarrow C_{\text{spend}} \cup \{ cm \} \).

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 \( C_{\text{spend}} \), and that \( \{ sn_1, sn_2 \} \cap \mathcal{S} = \emptyset \). If so, the effect is updating \( \mathcal{S} \leftarrow \mathcal{S} \cup \{ sn_1, sn_2 \} \), and \( C_{\text{spend}} \leftarrow C_{\text{spend}} \cup \{ cm, cm_3 \} \). Finally, at the start of an epoch, old enough spending coins are allowed for leadership proofs: \( C_{\text{lead}} \cup C_{\text{spend}} \), where \( C_{\text{spend}} \) 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 \( \text{STX}_{\text{ref}} \) (i.e. it is not \textit{this block’s} leadership transaction, it is considered an \textit{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 \( C_{\text{lead}} \) at some point. The effects of the transaction remain the same, although is no longer the leader of a block. A blocks transactions are validated \textit{prior} to the leadership transaction, as this may depend on adopted leadership transactions. The Merkle tree root of \( C_{\text{lead}} \) 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 \((\text{PUBLIC, TRANSFER})\).

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 \( \text{Lkg} \) set to the identity function \( \text{Lkg}_{\text{id}} \); i.e. the ledger leaks its entire content to the simulator, described in Appendix B. We argue that the simulator \( S_1 \) can simulate any real-world attacks on Ouroboros-Crypsinous against a leaky \( \mathcal{G}_{\text{PL}} \). In the second stage, we instantiate \( \text{Lkg} \) to \( \text{Lkg}_{\text{lead}} \), in which only the leaders of a given slot are leaked. We argue that \( S_2 \) is secure against this, as it behaves indistinguishably from \( S_1 \). The smaller steps of this proof are stated here, and argued in Appendix E.

Theorem 1. Ouroboros-Crypsinous \( \mathcal{UC} \)-emulates \( \mathcal{G}_{\text{PL}} \) with \( \text{Lkg} = \text{Lkg}_{\text{id}} \), under the \textit{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 \( \mathcal{G}_{\text{LEADER}} \), 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.
interface, Ouroboros-Crypsinous realizes the private ledger with \( Lkg = Lkg_{\text{cid}} \).

**Lemma 1.** The private ledger \( \mathcal{G}_{\text{PL}} \) with \( Lkg = Lkg_{\text{cid}} \) 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 \( Lkg = Lkg_{\text{lead}} \) under the DDH assumption, in the random oracle model.

The leakage \( Lkg_{\text{lead}} \) leaks only the leader of any given slot. We utilize a modified version of \( S_1 \), 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, \( S_2 \) 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 \( S_1 \), although requiring less leakage from the private ledger functionality. As a result, the same security argument as for \( S_1 \) 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 \( t_{c_2} \) and \( sk_{c_2} \) 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 \( \phi_f \) 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.

#### Table I

**Number of multiplicative constraints in SNARK relations**

<table>
<thead>
<tr>
<th>Constraint count</th>
<th>( \mathcal{L}_{\text{MUL}} )</th>
<th>( \mathcal{L}_{\text{LEAD}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check ( p_{c_2} )</td>
<td>2 \times 27,904</td>
<td>27,904</td>
</tr>
<tr>
<td>Check ( p_{c_2}, c_{c_2} )</td>
<td>43,808</td>
<td>43,808</td>
</tr>
<tr>
<td>Path for ( c_{m_{c_2}} )</td>
<td>1 (layer of 32)</td>
<td>1 (layer of 32)</td>
</tr>
<tr>
<td>Path for ( \text{root}<em>{c</em>{c_2}} )</td>
<td>34,225</td>
<td>34,225</td>
</tr>
<tr>
<td>Check ( m_{c_1} )</td>
<td>2 \times 27,904</td>
<td>27,904</td>
</tr>
<tr>
<td>Check ( c_{m_{c_1}} )</td>
<td>2 \times 2,542</td>
<td>2 \times 2,542</td>
</tr>
<tr>
<td>Check ( v_1 + v_2 = v_3 + v_4 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ensure that ( v_1 + v_2 &lt; 2^d )</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>Check ( y, \rho )</td>
<td>2 \times 3,252</td>
<td>2 \times 3,252</td>
</tr>
<tr>
<td>Check (approx.) ( y &lt; \text{ord}(G)\phi_f(v) )</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Total</td>
<td>209,466</td>
<td>201,493</td>
</tr>
</tbody>
</table>

#### 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 with 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.

7https://github.com/zcash/zcash/issues/2634
REFERENCES


[13] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros prao: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In Jesper Buus Nielsen and Vincent Rijmen, editors, OuroborosCrypsinousblockchainconsensusprotocol. https://github.com/ethereum/research/The functionality $F_{INIT}$ is parameterized by the number of initial stakeholders $n$ and their respective stakes $s_1, \ldots, s_n$. $F_{INIT}$ interacts with stakeholders $U_1, \ldots, U_n$ as follows:

- In the first round, upon a request from some stakeholder $U_i$ of the form $(\text{claim}, \text{sid}, U_i, pk_{ENC})$, then $F_{INIT}$ stores the keys tuple $(U_i, pk_{ENCI})$. It then samples $sk_{COIN}$ the way Ouroboros-Crypsinous does on GENERATE requests, $\rho_i$ randomly, computes $pk_{COIN} \leftarrow PRF_{sk_{COIN}}(0)$, and commits $(\text{em}_{pk_{COIN}}, \text{re}_i) = \text{Comm}(pk_{COIN}, s_i || \rho_i)$. Then, returns the tuple $(sk_{COIN} \leftarrow \rho_i, \text{re}_i, s_i)$. Once all parties have registered, it samples and stores a random value $\eta_i \leftarrow \{0, 1\}^\lambda$. It then


APPENDIX A

HYBRID WORLD FUNCTIONALITIES

Functionality $F_{INIT}$
The (proof-malleable) non-interactive zero-knowledge functionality \( F_{NZK} \) allows proving of statements in an NP language \( \mathcal{L} \).

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

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

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

**Functionality \( F_{\text{ENC}} \)**

\( F_{\text{ENC}} \) is parameterized by, a security parameter \( \kappa \), a set of parties \( \mathcal{P} \), and a maximum delay \( \Delta_{\text{max}} \).

**Key Generation.** Upon receiving a message \((\text{KeyGen}, \, \text{sid})\) from a party \( U_p \), verify that \( U_p \in \mathcal{P} \), and that this is the first key generation. If so, send \((\text{KeyGen}, \, \text{sid}, \, U_p)\) to \( A \), 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 \((\text{Encrypt}, \, \text{sid}, \, pk, \, \tau, \, m)\) 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 \( \tau < \tau_q + \Delta_{\text{max}} \). If so, send \((\text{Encrypt}, \, \text{sid}, \, \tau, \, m, \, U_p)\) to \( A \). Otherwise, send \((\text{DummyEncrypt}, \, \text{sid}, \, pk, \, \tau, \, m, \, U_p)\) to \( A \).
- Receive a reply \( c \) from \( A \), and send \((\text{ciphertext}, \, c)\) to \( U_p \). Further, if the conditions of the previous step were satisfied, record the tuple \((U_q, \, m, \, \tau, \, c)\).

**Decryption.** Upon receiving a message \((\text{Decrypt}, \, \text{sid}, \, \tau', \, c)\) from party \( U_p \in \mathcal{P} \):
- If \( \tau' < \tau_p \), return \( \perp \).
- Else, if a tuple \((U_m, \, m, \, \tau', \, c)\) was recorded, return \( m \) to \( U_p \).
- Otherwise, send \((\text{Decrypt}, \, \text{sid}, \, \tau_p, \, c, \, U_p)\) to \( A \). receive a reply \( m \), and if \( m \neq \perp \), forward \( m \) to \( U_p \).

**Update.** Upon receiving a message \((\text{Update}, \, \text{sid})\) from party \( U_p \in \mathcal{P} \):
1. Send \((\text{Update}, \, \text{sid}, \, U_p)\) to \( A \).
2. Update \( \tau_p := \tau_p + 1 \).

**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 Cryptosinus 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 Cryptosinus**

**Protocol** Ouroboros-Cryptosinus\(_h\)(\(U_p, \, \text{sid}\))

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

**Interacting with the Ledger (cf. Section VI-D):** Upon receiving a ledger-specific input \( I \in \{(\text{Submit} \ldots), \, (\text{Read} \ldots), \, (\text{Maintain-Ledger} \ldots)\}\) verify first that all resources are available. If not all resources are available, then ignore the input; else execute one of the following steps depending on the input \( I \):
- If \( I = (\text{Submit}, \, \text{sid}, \, \text{(PUBLIC TRANSFER)} \parallel tx) \) then set invoke the protocol SubmitXfer\((tx, \, \text{Con}, \, \log)\).
- Else if \( I = (\text{Submit}, \, \text{sid}, \, \text{tx}) \) then set invoke the protocol SubmitGeneric\((\text{sid})\).
- Else if \( I = (\text{generate}, \, \text{sid}, \, \text{tag}) \) then
  - If \( \text{tag} = \text{COIN} \), sample \( sk_{\text{COIN}} \leftarrow \{0, \, 1\}^\kappa \), and let \( sk'_{\text{COIN}} = \text{PRF}_{sk_{\text{COIN}}} (1) \), for \( i \in \tau, \ldots, \tau + R \). Let root be the root of the Merkle tree over \( sk_{\text{COIN}} \ldots, \, sk'_{\text{COIN}} \), and \( pk_{\text{COIN}} \leftarrow \text{PRF}_{sk_{\text{COIN}}} (\tau) \). Insert the Merkle tree, and \( \tau \) into \( \text{Con} \), and return \( pk_{\text{COIN}} \).
  - If \( \text{tag} = 1\text{D} \), and this is the first query for \( 1\text{D} \), send \((\text{KeyGen}, \, \text{sid})\) to \( F_{\text{ENC}} \). Denote the response by
A. The Staking Procedure

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

1. for \((pk_{\text{CON}}, \rho_c, \tau_c, v_c) \in \mathcal{C}\) do
2. if \(c\) is not eligible for leadership then continue
3. Send \((\text{eval}, (\text{id}_R, \text{NONCE} || \eta_p, || sl))\) to \(\mathcal{G}_R\), and denote the response \(r\).
4. Send \((\text{eval}, (\text{id}_R, \text{LEAD} || \eta_p, || sl))\) to \(\mathcal{G}_R\), and denote the response \(\eta\).
5. Lookup \(sk_{\text{CON}, r}, \text{root}_c,\) and \(\tau_c\) in \(\mathcal{C}_\text{free}\) corresponding to \(pk_{\text{CON}}\).
6. Let \(\rho \leftarrow \mu_{\text{root}_c} \equiv r\); \(y \leftarrow \mu_{\text{root}_c} \equiv r\).
7. if \(y < \text{ord}(G)\phi_f(v)\) then
8. Compute \(s_1\) as in Ouroboros Genesis [11].
9. Set \(pt\) to \(H(\text{head}(\mathcal{C}_{\text{loc}})); h \leftarrow H(s_1)\).
10. Set \(\rho_c \leftarrow \text{PRF}^\text{root}_c(\rho_c); s_n \leftarrow \text{PRF}^\text{root}_c(\rho_c)\).
11. Set \((cm_{c}, \tau) \leftarrow \text{Comm}(pk_{\text{CON}} || \tau_c || \rho_c)\).
12. Let \(\text{tx}_\text{lead}\) be, in order, the list of leadership transactions made by \(U_p\) not in \(\mathcal{C}_{\text{loc}}\).
13. Let \(\tau_c\) be the root of the Merkle tree \(\mathcal{C}_{\text{lead}}^\text{root}\) in \(\mathcal{C}_{\text{loc}}\), after applying all transactions in \(\text{tx}_\text{lead}\). Let path be the path to \(cm_{c}\) in the same Merkle tree.
14. Let path be the Merkle path to \(sk_{\text{CON}, r}\) in the secret-key Merkle tree.
15. Let \(x = (cm_{c}, \rho_{c}, \eta_p, sl, p, h, \rho_{\text{root}_c}, \mu_{\text{root}_c}, \text{root}_c)\).
16. Let \(w = (\text{path}, \text{root}_c, \text{path}_c, \tau, \rho_{\text{root}_c}, \rho_{c}, \tau, \text{root}_c, \tau)\).
17. Send \((\text{prove}, \text{id}, x, w)\) to \(\mathcal{F}_{\text{LEAD}}\), and denote the response \(\pi\).
18. Let \(\text{tx}_{\text{lead}} = (\text{LEAD}, \text{tx}_{\text{ref}}, (cm_{c}, \rho_{c}, \eta_p, sl, p, h, \rho_{\text{root}_c}, \text{root}_c, \tau))\).
19. Set \(B \leftarrow (\text{tx}_{\text{lead}}, \text{id}); \mathcal{C}_{\text{loc}} \leftarrow \mathcal{C}_{\text{loc}} || B\).
20. Update \(c\); \(\mathcal{C} \leftarrow (\mathcal{C} \setminus \{(pk_{\text{CON}}, \rho_{c}, \tau_c, v_c)\}] || \{(pk_{\text{CON}}, \rho_{c}, \tau_c, v_c)\})\).
21. Send \((\text{MULTICAST}, \text{id}, \text{tx}_{\text{lead}})\) to \(\mathcal{F}_{\text{N-MC}}\) and proceed from here upon next activation of this procedure.
22. Send \((\text{MULTICAST}, \text{id}, \text{tx}_{\text{lead}})\) to \(\mathcal{F}_{\text{N-MC}}\) and proceed from here upon next activation of this procedure.
23. break
end if
end for
24. while a \((\text{CLOCK-UPDATE}, \text{id}_c)\) has not been received during the current round do
25. Give up activation. Upon next activation of this procedure, proceed from here.
end while

B. The Ledger Maintenance Procedure

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

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 \((x, w) \in \mathcal{L}_{\text{LEAD}}\) iff all of the following hold:

- \(x = (cm_{e_1}, s_{n_{e_1}}, \eta, sl, p, h, \rho_{\text{root}_c}, \mu_{\text{root}_c}, \text{root}_c)\)
- \(w = (\text{path}, \text{root}_c, \text{path}_c, \tau_c, \rho_c, \tau_1, v, r_{e_1})\)

\(pk_{\text{CON}} = \text{PRF}^\text{root}_c(\tau_c)\)

\(\rho_{e_2} = \text{PRF}^\text{root}_c(\rho_{e_1})\)

\(\forall i \in \{1, 2\} : \text{DeComm}(cm_{e_1}, pk_{\text{CON}} || v, \rho_{c}, r_{e_1}) = \top\)

\(cm_{e_1}\) is in root at path path

\(\text{path}_c\) is a valid path to a leaf-preimage at position \(sl - \tau_c\) in the Merkle tree with root root

\(s_{n_{e_1}} = \text{PRF}^\text{root}_c(\rho_{e_1})\)

\(y = \mu - \rho_{e_1}\)

\(y < \text{ord}(G)\phi_f(v)\)

Note that \(x\) of LEAD contains values \(sl, h, \rho,\) and \(\rho_{\text{root}_c}\) that seemingly nothing is proved about. As a UC proof system is non-
malleable, this makes them part of the signature of knowledge message.

A tuple $(x, w) \in \mathcal{L}_{\text{XFER}}$ iff all of the following hold:

- $x = \{cm_{e_2}, cm_{e_1}, \{sn_{e_1}, sn_{e_2}\}, \text{root}\}$
- $w = \{(\text{root}_{e_1}, \tau_{e_1}, \rho_{e_1}, \tau_{e_2}, \rho_{e_2}, \tau_{e_2}, v_1, \text{path}_1), (\text{root}_{e_2}, \tau_{e_2}, \rho_{e_2}, \tau_{e_2}, v_2, \text{path}_2), (pk_{G}^{\text{CON}}, \rho_{e_3}, \tau_{e_3}, v_3), (pk_{G}^{\text{CON}}, \rho_{e_4}, \tau_{e_4}, v_4)\}$
- $\forall i \in \{1,2\} : \rho_{e_i} = \text{PRF}_{\text{root}_{e_i}}(\tau_{e_i})$
- $\forall i \in \{1..4\} : \text{DeComm}(cm_{e_i}, pk_{G}^{\text{CON}} \parallel v_i \parallel \rho_{e_i}, \tau_{e_i}) = \top$
- $v_1 + v_2 = v_3 + v_4$
- $cm_{e_i}$ is in root at path path$_i$.
- $cm_{e_i}$ is in root at path path$_2$, or $v_2 = 0$ and root$_{e_2} = \rho_{e_2} = \text{PRF}_{\text{root}_{e_2}}(\rho_{e_1})$.
- $\forall i \in \{1,2\} : sn_{e_i} = \text{PRF}_{\text{root}_{e_i}}(\rho_{e_i})$

**APPENDIX E**

**SECURITY ANALYSIS (CONTINUED)**

**Proof of Lemma 1**: To begin with, we note that if $\text{Lkg} = \text{Lkg}_{\text{id}}$, the simulator $S_1$, 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 $\mathcal{T}^H$ (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 Ouroboros-Crypsinous) is used. In particular, we note that parties compute some function $f(tx)$ 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 $f^{-1}(U_p, tx)$. 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, $\text{BlindTx}(\{U_p\}) = f^{-1}(U_p) \circ f$, 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, $sk_{G}^{\text{CON}}$, 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 $sk_{G}^{\text{CON}}$ leaves correspond to updating the $F_{KES}$ 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 leader-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 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 $F_{\text{FWENC}}$. 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 maintain 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 bijection 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 $S_1$ can construct “ideal” transactions corresponding to any possible adversarial transaction, that also preserve the invariant. We note that as the set of coins changes if 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, $\text{Lkg}_{\text{id}}$, 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 Ouroboros-Crypsinous realizes $\mathcal{G}_{\text{PL}}$ with $S_1$, under the leakage predicate $\text{Lkg}_{\text{id}}$.

**Proof of Lemma 4**: We note there are four primitives that are simulated in simulated transactions: Commitments, NIZKs, $F_{\text{FWENC}}$ 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 $\mathcal{F}_{\text{FSENC}}$, 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 nonce $\nu$ 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 $c$, $\nu$, and $\ell$ and the state from [1], consisting of the local chain $c_{\text{loc}}$, and buffer. We note that while on corruption, the simulator can extract ideal world coins from $G_{\text{pk}}$ 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 equivocability of commitments, and the forward-security of $\mathcal{F}_{\text{FSENC}}$. 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 equivocability, the third is secure by the forward-security of $\mathcal{F}_{\text{FSENC}}$, 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 $S_1$ and $S_2$ 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 indistinguishability 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 $\kappa$, 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-privat and forward-secure encryption scheme, when used in the $\mathcal{F}_{\text{KEYMEM}}$-hybrid model, can be used to realize $\mathcal{F}_{\text{FSENC}}$. 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 $\mathcal{F}_{\text{KEYMEM}}$ 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, $\text{Init}_{\text{comm}}$, $\text{Comm}$, $\text{DeComm}$, $\text{Init}_{\text{comm}}$, $\text{Comm}$, and $\text{Equiv}$. $\text{Init}_{\text{comm}}$ takes the function of the CRS assumption, by generating a public key $\text{pk}_{\text{COMM}}$ which is given as an argument to $\text{Comm}$ and $\text{DeComm}$. In addition to satisfying the traditional commitment properties, of binding, hiding, and correctness, the scheme also satisfies equivocality. $\text{Init}_{\text{comm}}$ provides an equivocation key in addition to $\text{pk}_{\text{COMM}}$. This equivocation key can be used to break the binding property – $\text{Comm}$ can generate a commitment without a message, and $\text{Equiv}$ 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 Crypsinos’ design.

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