Universally Composable Transaction Order Fairness: Refined Definitions and Adaptive Security

Michele Ciampi[1], Aggelos Kiayias[2] and Yu Shen[3]

[1]University of Edinburgh, michele.ciampi@ed.ac.uk
[2]University of Edinburgh and IOG, aggelos.kiayias@ed.ac.uk
[3]University of Edinburgh, shenyu.tcv@gmail.com

Abstract

While consistency and liveness are the defining properties of ledger consensus, fair ordering has emerged as an independent consideration whose importance is underscored by the observation of real world transaction inclusion strategies that manipulate fairness such as Miner Extractable Value. Receiver-order fairness is a fine-grain notion of fairness that determines the order of any two submitted transactions based on the two sequences of reception timestamps for the two transactions across all nodes in the network. Given this information, ledger serialization can be viewed as the social choice problem of producing the most agreeable transaction order based on the "preferences" of the miners or validators. In this work, our contribution is three-fold: (i) We put forward a formal Universally Composable definition for order fairness that encompasses receiver order fairness as well as input causality. (ii) We design a novel ledger protocol that preserves input causality and receiver order fairness. (iii) We capture in our composable definition and construction the role that transaction fees play when dealing with fairness.

Our protocol is based on a novel YOSO-style approach that allows encrypting transactions for a short period of time. Notably, the communication complexity required to decrypt the transactions is independent of the number of encrypted transactions. To the best our our knowledge, we are the first to provide a YOSO-style approach with such asymptotic complexity while relying on standard cryptographic assumptions.

An abridged version of this paper appears in *Proc. ASIACRYPT 2025.*

Contents

1 Introduction **3**
1.1 Our Results . . . 3
1.2 Related Work . . . 5

2 Technical Overview **6**
2.1 Receiver Order Fairness and Input Causality in the UC Setting . . . 6
2.2 Input Causality from Standard Assumptions . . . 6
2.2.1 How to Concretely Enable Causality . . . 9

3 Preliminaries **11**
3.1 The GUC Framework . . . 11
3.2 Receiver Order Fairness . . . 12
3.3 Cryptographic Building Blocks . . . 12
3.4 Proof-of-Stake Blockchains . . . 13

4 Extended Ledger Functionality **13**
4.1 Transaction Order Bookkeeping . . . 14
4.2 Transaction Fees for Input Causality . . . 16
4.3 Extend Policies with Receiver Order Fairness . . . 22

5 Protocol Details **24**
5.1 Private Accounts and Transaction Fees . . . 24
5.2 Secret-Sharing Transactions to an Ephemeral Committee . . . 26
5.3 PoS-based Role Assignment and Transaction Serialization . . . 29
5.4 The Full Protocol Description . . . 31

6 Security Analysis **40**
6.1 Composable Guarantees . . . 47

A Preliminaries (Cont'd) **53**
A.1 Adaptively Anonymous Public-Key Encryption . . . 55
A.2 Hybrid Functionalities for PoS . . . 57

B Glossary **60**

1 Introduction

Ledger consensus, a special case of state machine replication [Sch90], asks a set of servers to serialize an ever growing sequence of transactions submitted by clients. It requires two fundamental properties: consistency, namely that the view of the transaction log across servers is consistent at least up to prefixes, and liveness, namely that submitted transactions are eventually incorporated into all the servers' logs. In [KZGJ20, Kur20] a third property was highlighted, *order fairness* referring to the characteristics of the specific order under which the submitted transactions become serialized in the common log. Aside theoretical considerations, order fairness turns out to be also a critical property in practice, in light of a range of attacks and protocol behaviors such as miner extractable value (MEV), cf. [DGK$^+$20].

There are two dimensions to order fairness: one is temporal and the other is causal. The temporal aspect, cf. [KZGJ20, Kur20], at a high level mandates that "earlier" transactions should be serialized first. The causal aspect on the other hand, cf. [RB94, CKPS01], asks that the adversary cannot create a transaction that depends on an already submitted transaction `tx` and have it serialized ahead of `tx`. It is worth observing that the temporal aspect, interpreted in its most idealized sense, subsumes the causal. Nevertheless, the most idealized version of temporal fairness cannot be reasonably attained unless for a setting where all participants have zero latency connections to a trusted third party. Indeed in a setting where a $\Delta > 0$ delay in transaction propagation occurs ideal order fairness is impossible, cf. [CKS24].

Given that ideal order fairness is impossible, certain concessions should be made in the temporal dimension to make it tractable depending on the relative timing of submitted transactions. One line of reasoning focuses on "sender" order fairness and posits that order fairness should be upheld as long as two transactions are submitted Δ apart, for a suitable choice of Δ (and possibly pairing that with additional assumptions on the network delay). This angle is reflected in the fairness definitions given in [Kur20] and [CKS24]. Another line of reasoning focuses on the "receiver" end, and requires ordering of transactions taking into account the timestamp sequence that each transaction induces when it arrives to each of the consensus nodes. This is reflected in the fairness definitions of [KZGJ20, KDL$^+$23, KLS24] where each sequence of timestamps can be interpreted as a voting preference for the transaction in question, and serializing the transaction can be seen as the social choice problem of finding the most agreeable final ordering — as pointed out in [KZGJ20] this introduces an impossibility via Condorcet's paradox. Note that this latter flavor of receiver order-fairness requires fairness in serialization, even for transactions that are submitted less than Δ-apart where Δ is the network delay; on the other hand, if transactions are ordered fairly based on receiver order-fairness, then they are also automatically ordered fairly according to sender order-fairness for transactions that are submitted Δ-apart: the reason is that such transactions will result in receiver timestamp sets that are non-overlapping and hence it is required for them to be serialized following sender order fairness. This is one of the reasons that receiver-order fairness is considered to be a more refined notion for the problem at hand. It remains an open question whether it can be captured and realized in the Universal Composition (UC) setting [Can00].

1.1 Our Results

Our results are three pronged and are described as follows.

Receiver order fairness and input causality in the UC setting. Receiver order fairness is difficult to capture in the UC setting as, typically, the ideal world does not sustain information about network receiving events for specific messages that are in transit between protocol participants. In such a setting it is impossible to enforce a receiver side order fairness as the ideal functionality lacks

the necessary information to differentiate transactions based on receiver timestamps. Moreover, even if we allow the ideal world adversary to submit such timestamps per transaction the resulting notion is vacuous as any sender order fair protocol would also realize this functionality as it allows the simulator to influence the functionality maximizing the delays of all submitted transactions.

We resolve the problem by transitioning to the global UC setting [CDPW07] and introducing a global networking functionality. The advantage of our approach is in that global functionalities are present both in the real and ideal world and can be accessed by other functionalities to share relevant information. In this way, receiving network events can be consulted by the fair ordering functionality and influence the serialization constraints imposed by the functionality to the adversary. In Section 2 we elaborate further on the connection.

Permissionless input causality. Input causality critically relies on suppressing transaction information prior to serialization to prevent any causal dependency on adversarial transactions that may settle ahead of honestly submitted transactions. In a permissioned setting, namely the setting where all participants are registered with each other, e.g., there is a PKI accessible to all that can be used to identify them, a fairly straightforward approach is to setup a threshold encryption scheme among all parties and have submitted transactions encrypted under that scheme, serialized and subsequently decrypted and processed. It is straightforward to observe that this approach does not transfer to a permissionless setting, even in the presence of a PKI, given that the setup of such a threshold encryption among all parties would be a prohibitively expensive operation. A possible workaround to this problem is to assume a trusted execution environment that can assist the parties in encrypting transaction information. Pairing this method with a permissionless blockchain protocol (cf. the approach of [CKS24]) results in a protocol that is quasi-permissionless in the sense that participation is preconditioned in the acquisition of the appropriate hardware device. Achieving a truly permissionless operation requires going beyond the above two approaches and realizing a method for encoding transactions that is efficiently shared amongst all participants in an efficient manner. The straightforward approach would combine the threshold encryption approach mentioned with YOSO—"you only speak once"—secure multiparty computation [GHK$^+$21].

In existing YOSO-style approaches, a small committee maintains a secret sate, and clients can ask the committee to perform some computation on such a private state. To achieve input causality, one could ask the committee to maintain in its secret state all the transactions issued by the clients for a brief period of time (an epoch). At the end of the epoch, the parties can ask to decrypt the transactions in the order they were submitted, and let the blockchain execute them. This is indeed the approach followed in existing approaches. The downside of such an approach is that the communication complexity of the decryption process is dominated by the number of transactions that need to be decrypted.

We instead devise a new approach, that allows the committee members to only disclose a set of secret keys (whose size is independent of the number of transactions), that can be used by anyone to decrypt all the transactions issued up to that point. Doing that while maintaining security against adaptive corruption represents a challenging aspect of our contributions.

An important consideration in the YOSO setting is that erasures are crucial for security: YOSO protocols require nodes to perform a secret sharing and immediately after in an atomic way erase any private information they have used; if that's not available then it is possible for the adversary to corrupt past committee members (whose identity has been disclosed) and reveal information that should remain hidden (in our setting it would be encoded transactions that should be hidden to ensure input causality). Our approach follows a similar blueprint, requiring clients to perform a similar type of sharing operation that we nevertheless prove that it is not critical for security to be accompanied by an erasure operation — hence in our construction, erasures are only needed for

the nodes performing the YOSO protocol (i.e., the maintainers of the blockchain).

Denial-of-Service mitigations and order fairness. A frequently overlooked question in prior work for order fairness is how to distribute the transactions for fair transaction ordering in a way that denial-of-service attacks can be mitigated. In standard sender or receiver order fairness one can assume that a fee can be associated with each submitted transaction; provided that the availability of such a fee can be checked at each propagation hop of a transaction, denial of service attacks can be mitigated: indeed, an adversary will eventually run out of funds if it wishes to flood the network with transactions.

Unfortunately, once input causality is added to the set of desiderata, the above straightforward adaptation from standard blockchain protocol design is inadequate: the reason is that checking the fee availability leaks information about the sender (e.g., points to her account) and as a result violates input causality. We observe that what is needed is a way to submit transactions so that the availability of the fee can be verified without leaking any other information about the source of the transaction. We demonstrate how it is possible to achieve that by using techniques from anonymous e-cash. Specifically, each participant in our system can transfer funds to a private account that subsequently can be used to pay for transaction fees in a privacy-preserving manner. This facilitates transaction propagation without disclosing any information that can be exploited by the adversary to violate input causality.

Remark 1. Our protocol employs a transaction fee scheme that is *flat* per unit of transaction size (note that the length of the transactions is not considered part of the attack vector for input causality). One can think of it as paying the base fees for transactions to be included on the blockchain; thus, another layer of fees that pay for the actual transaction execution can be run alongside. Moreover, while our construction is compatible with customized fees (e.g., the fee mechanism in Ethereum with tips), this would require the ledger to be adapted so that the simulator can exploit this information and act accordingly. Given that our goal is to model the fair ledger, we choose to avoid these unnecessary complications.

1.2 Related Work

To the best of our knowledge, [CKS24] is the only work that captures and implements transaction order fairness in the UC setting. In [CKS24], the ledger functionality records a sending timestamp for each transaction, but, as hinted above, it is ignorant of any individual receiving times at each node. As a result the functionality in [CKS24] is not capable to capture receiver order fairness. Regarding the implementation, their protocol relies on trusted hardware to apply encryption to the submitted transactions so input causality is achieved.

We compare our protocol to other approaches that achieve (or could be used to achieve) input causality against *adaptive* adversaries. A common way to achieve input causality is to encrypt the transactions via a witness encryption scheme [GGSW13], where the witness is represented by an extension of the current chain. This guarantees that the transactions can be decrypted only later in the future. This approach, other than being problematic in some cases (like proof-of-stake blockchains), requires to rely on iO type of assumptions. A similar approach is proposed in [CDK$^+$22], where a milder form of witness encryption is used. The authors argue that this notion can be instantiated from standard cryptographic assumptions. The approach of [CDK$^+$22] requires executing their witness-encryption scheme on a statement that depends on the entire blockchain. Moreover, decrypting a transaction requires at least one maintainer to issue a message. Hence, the communication complexity of the decryption process depends (at least linearly) on the number of transactions. A similar complexity would occur in other YOSO-style approaches

like [BGG+20, GHK+21, GHM+21]. Moreover, [BGG+20, GHK+21] tolerates a corruption threshold of 1/4, whereas we only require honest-majority.

Another approach is to use less standard assumptions like Time-lock puzzles/VDFs [CDEG23] to make sure that transactions are decrypted only after a certain time. Our protocol can be instantiated (in the random oracle model) from CDH and LWE assuming erasures for the maintainers, or only from CDH (and random-oracle) if we allow erasure also on the clients.

2 Technical Overview

2.1 Receiver Order Fairness and Input Causality in the UC Setting

To capture receiver order fairness and input causality in UC, we define the functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$. This functionality captures the property of *input causality* similarly to [CKS24]. In particular, this ledger functionality only leaks to the adversary transaction identifiers that do not carry any information about the transactions themselves. The adversary has to commit first to an ordering, and then subsequently, the contents of the transactions are revealed so that the ledger can be assembled.

The main difference compared to [CKS24] is that we aim to capture the property of receiver-order fairness instead of sender-order fairness. As mentioned in the previous section, to capture receiver-order fairness, the functionality should order the transactions, taking into account the timestamp sequence that each transaction induces when it arrives at each of the ledger maintainers. Capturing this in UC necessarily requires the ledger functionality to be aware of the existence of the communication network. Such a network disappears in the ideal world in existing UC formalizations of secure ledgers. In our formalization, instead, we treat the network as a global ideal resource that is available both in the ideal and the real world. In the real world, the parties use the network to send transactions and communicate protocol messages. In the ideal world, the behavior of the honest parties is emulated by $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$, which interacts with the network on behalf of these. This allows $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ to obtain the time at which the transactions are delivered to each honest party. We note that in the ideal world, there will be no actual transactions floating in the network, but only strings (determined by the ideal world adversary) that mirror the behavior of transactions floating in the network during the execution of a real world protocol.

We realize a protocol that enjoys both the property of receiver-order fairness and input causality. To achieve the property of receiver order fairness, we follow the approach of [CKS24] with the following main differences: (i) our protocol is presented in the Proof-of-Stake setting ([CKS24] is Proof-of-Work) and (ii) capturing receiver-order fairness requires building a proper transaction graph as in [KLS24]. We provide a formal description of the protocol in the technical part of the paper. Instead, we devise a completely new approach to achieve input causality. In [CKS24], input causality was achieved using trusted execution environments, while in this paper, we solve the problem using standard polynomial-time cryptographic assumptions. The rest of this section focuses on describing our novel approach to achieving input-causality.

2.2 Input Causality from Standard Assumptions

We present a high-level overview of a protocol that achieves input causality and retains its security against adaptive corruption. We present the protocol assuming that the execution of a blockchain protocol is divided into fixed, discrete units of time called *epochs*. In each epoch there is an arbitrary number of N active maintainers, among which the majority is honest. Moreover, each maintainer

has associated a public key pk, known to all the other protocol participants (in the formal part of the paper, we will describe our approach in the Proof-of-Stake setting).

As highlighted in the previous section, the common blueprint that one can follow to achieve input causality is the following. The parties encrypt their transactions and post the corresponding ciphertexts on the blockchain. At the end of the epoch, the transactions are decrypted and parsed in the order their corresponding ciphertexts appeared on the blockchain (the process is repeated through multiple epochs). A naïve approach to realize the above would be to let each transaction issuer encrypt their transactions and later decrypt them at the end of the epoch. This naïve approach fails because an adversary can, after having looked at the decrypted transactions of the honest parties, selectively decrypt only a subset of his transactions. A simple solution to this problem is to let the transaction issuers secret-share the transactions via a verifiable secret sharing scheme (VSS). A client that wishes to issue a transaction, VSS the transaction, thus obtaining N shares, and encrypts the i-th share using the public key of the i-th maintainer[1]. Finally, it publishes all the obtained ciphertexts on the blockchain (eventually with a proof that the ciphertexts contain valid VSS shares of a transaction). At the end of the epoch, the N maintainers will decrypt the ciphertexts and make the VSS shares public. The transactions will then be executed following the order in which corresponding ciphertexts appeared on the ledger state. As long as the majority of the maintainers are honest, this approach guarantees that no adversary can inject transactions or reorder them based on the honest parties' transactions submitted during the epoch. Moreover, security against adaptive corruption is trivially guaranteed if we assume the VSS and the public-key encryption scheme to be secure against adaptive corruption.

This approach, however, creates a significant overhead. Each transaction requires generating and publishing on the blockchains N ciphertexts, which then need to be all opened at the end of the epoch.

A simple way to reduce the overhead is to use a smaller committee of size $n \ll N$. The committee can be elected via standard techniques, and the secret sharing of the transactions can be performed against such a smaller committee. This approach has two drawbacks. The first is that it is not secure against adaptive corruption. The adversary, right after the committee is created and the participants' identities are announced, can potentially corrupt all the members of the committee. The second drawback is that every transaction would need a dedicated phase of secret sharing and reconstruction (i.e., the communication complexity of the reconstruction phase depends both on the number of issued transactions and on the size of the committee).

The first problem is very well known, and it has been solved in [BGG$^+$20, GHK$^+$21, GHM$^+$21, CDK$^+$22] in the multi-party computation setting with the so-called YOSO approach. At a high level, the YOSO approach allows hiding the identities of the committee members until they need to send a protocol message. But before sending their messages, the committee members erase their state and delegate the rest of the computation to a new committee. This means that if the adversary corrupts the parties that have spoken, their memory would look empty. Moreover, the messages that the old committee sends to the new committee are generated in such a way that the identities of the new committee members remain hidden to the adversary.

We devise an approach that follows the YOSO approach but whose security holds in the honest majority setting, uses lightweight cryptography, and the communication complexity of the decryption phase is *independent* of the number of encrypted transactions. Achieving this while maintaining security with adaptive corruption represents the most challenging and novel aspect of our construction, that we now highlight.

Stage 1: Creation of the ephemeral committees. Each maintainer P_i with $i \in [N]$ has

[1]In this example we are assuming that the VSS is non-interactive.

associated to it a *long-term* public key, known by all the protocol maintainers. Each P_i tries to *self-elect* themselves as the *leader*. This can be done using standard techniques, for example, by locally evaluating a VRF and comparing its output against a target value. If the comparison is successful, then the leader can prove non-interactively that they are indeed entitled to be a leader issuing a VRF proof. We will make sure that among all the leaders elected in every epoch, the majority of them are honest (we refer to the technical section for more detail).

If P_i is elected, it first generates an *ephemeral* key-pair $(\mathsf{epk}_i, \mathsf{esk}_i)$ for a public-key encryption scheme, and performs a t-out-of-n (with $n < N$)[2] secret sharing of the secret key esk_i with $t = n/2 + 1$, thus obtaining the shares $\mathsf{esk}_{i,1}, \ldots, \mathsf{esk}_{i,n}$. Then P_i randomly chooses a *secret-holding committee* — a subset of n parties among all maintainers, in such a way that, with overwhelming probability, the majority of the members are honest. For each P_j elected to be part of the secret-holding committee of P_i, P_i encrypts $\mathsf{esk}_{i,j}$ under pk_j thus obtaining $e_{i,j}$. Then it posts all the ciphertexts $(e_{i,1}, \ldots, e_{i,n})$, the ephemeral public key epk_i, and the VRF proof on the blockchain, along with a zero-knowledge proof, proving that the ciphertexts contains a valid secret sharing of the secret key associated to epk_i.[3]

To prevent the adversary (upon corruption) from learning P_i's ephemeral secret key, we require P_i to erase his private state (hence also the ephemeral key) before posting its messages on the blockchain. Note that an adversary may understand the identities of the next committee members (and consequently corrupt all of them) by just looking at the ciphertexts $(e_{i,1}, \ldots, e_{i_n})$, as the ciphertexts could leak information about the public keys themselves. This problem can be easily solved using an encryption scheme that hides the public keys (more details are provided in the technical section)[4].

Stage 2: Transaction issuing. For a client to send a transaction $\mathtt{tx}$, it first collects all ephemeral public keys that appeared on the blockchain that come with a valid VRF proof (i.e., the ephemeral public keys of the self-elected leaders). Let these keys be represented by the set $\mathcal{K} = \{\mathsf{epk}_1, \ldots, \mathsf{epk}_k\}$.

Then, the client encrypts the transaction by sampling a random string $s \leftarrow \{0,1\}^\kappa$ and computing $e \leftarrow \mathtt{tx} \oplus H(s)$, where H is modeled as a random oracle (more detail on this hereafter in this section). The client then performs a $(k/2+1)$-out-of-k secret sharing of s, thus obtaining $\{s_i\}_{i\in[k]}$, and encrypts the i-th share using the i-th ephemeral key (for each $i \in [k]$) thus obtaining the ciphertexts $c_1^{\mathtt{tx}}, \ldots, c_k^{\mathtt{tx}}$. The client then posts on the blockchain the *encrypted transaction* $((c_1^{\mathtt{tx}}, \ldots, c_k^{\mathtt{tx}}), \pi^{\mathtt{tx}}, e)$, where $\pi^{\mathtt{tx}}$ is a zero-knowledge proof proving that $c_i^{\mathtt{tx}}$ is a valid encryption of the i-th share of s under the public key epk_i. We stress that we prove nothing about the relation between the random pad s and e. Indeed, an adversary could generate a secret sharing of s, and computes $e \leftarrow \mathtt{tx} \oplus H(s')$ with $s \neq s'$.[5]

Stage 3: Key reconstruction. Upon reaching the end of the epoch, parties can decrypt the transactions that have been sent so far by reconstructing sufficiently many ephemeral secret keys. We recall that each (honest) leader must have secret-shared their ephemeral secret key to a secret-holding committee. Because we guarantee that the majority of the leaders are honest and that

[2]Parameter n depends on the corruption threshold, the smaller the corruption threshold is, the smaller n can be.

[3]Each party will select its secret-holding committee; as such, every self-elected party may disperse its ephemeral key to different committees. Also note that we will not simply generate proof about the ciphertexts being generated correctly, as the public-key encryption scheme we rely on uses the random oracle. We refer to the technical part of the paper for more details on how to solve this problem.

[4]We note that we need the public-key scheme to be resilient against adaptive corruption (with erasure), this is a property satisfied by the public-key scheme we use.

[5]We clearly cannot prove anything about the relation between s and c as such a statement would involve the description of H, which in our case is modeled as a random oracle.

in each of the secret-holding committees sampled by honest parties, there is an honest majority, then all the encrypted transactions posted on the blockchain (both malicious and honest) can be decrypted. In more detail, a party P_j with long-term secret key sk_j successfully decrypts a share $\mathsf{esk}_{i,j}$ (received from the leader P_i in Stage 1) and posts $\mathsf{esk}_{i,j}$ of the blockchain along with a proof of correct decryption.

If the majority of members of the secret-holding committee of P_i are honest, then the shares posted on the blockchain by these committee members will allow the reconstruction of the ephemeral secret key esk_i. This key can then be used to decrypt the i-th share in all the encrypted transactions. Because we can guarantee that the majority of the leaders are honest, then we can decrypt the majority of the shares of all the encrypted transactions. For example in the case of the encrypted transaction $(c_1^{\mathtt{tx}}, \ldots, c_k^{\mathtt{tx}}), \pi^{\mathtt{tx}}, e)$, we can decrypt at least the ciphertexts $c_{i_1}^{\mathtt{tx}}, \ldots, c_{i_{k/2+1}}^{\mathtt{tx}}$ thus obtaining the shares that allow for the reconstruction of the random-pad r. The transaction is finally obtained by computing $H(r) \oplus e$.

On the security against adaptive corruption of our protocol. Assuming erasures, we can tolerate the adaptive corruption of the maintainers. Indeed, as we have highlighted, the maintainers, right before sending their messages, delete their entire secret state, except the long-term secret key. Moreover, the adversary is not aware of who the honest leaders are due to the private election we perform via the VRF. Also, because the leaders erase their state, even if the leaders are corrupted after sending their messages (along with a VRF proof proving that they are the leaders), at that point it is too late to learn any information about the ephemeral keys or the identities of the secret-holding committees. We remark that the long-term keys must be keys of a scheme secure against adaptive corruption. This will allow the simulator to fake the values encrypted during the simulation (more detail in the formal part of the paper).

We still need to argue about the security of the clients. While erasure is a reasonable assumption to have for the maintainers of a blockchain (e.g., maintainers may have dedicated hardware and software that provides a higher level of security by erasing old private states), it may not be a reasonable assumption to have on the clients. We argue that our protocol retains its security without erasure in the case of adaptive corruption of the clients.

During the simulation, whenever the environment requires the client to issue a transaction, we do the following. We take a random value s, and compute $((c_1^{\mathtt{tx}}, \ldots, c_k^{\mathtt{tx}}), \pi^{\mathtt{tx}}, e)$ following the honest procedure discussed above, with the only difference that e is a random value (recall that the honest party would have instead computed $e \leftarrow \mathtt{tx} \oplus H(s)$). Upon corruption, the simulator receives $\mathtt{tx}$ and acts as follows. Compute $e^\star \leftarrow \mathtt{tx} \oplus e$, and program the random-oracle to return $e^\star$ when queried on input s. The secret state returned to the adversary would correspond to the randomness used to generate $(c_1^{\mathtt{tx}}, \ldots, c_k^{\mathtt{tx}})$, $\pi^{\mathtt{tx}}$, and $\mathtt{tx}$ (note that $\pi^{\mathtt{tx}}$ is not simulated). We recall that clients also need to issue zero-knowledge proofs. In [CSW22] the authors provide a scheme that does not rely on erasure and achieves adaptive corruption relying on LWE or LPN and DDH. If one wishes to relax the security of the protocol and admit erasure also for the client then any UC-secure zero-knowledge scheme can be used[6].

2.2.1 How to Concretely Enable Causality

In the approach discussed so far, the clients encrypt their transactions and post them on the blockchain. Once the epoch terminates, the transactions are decrypted and executed in the order they originally appeared. In particular, this means that the blockchain should accept an encrypted

[6]Assuming erasure any non-interactive zero-knowledge scheme can be made secure with adaptive corruption by requiring the prover to erase the randomness and the witness before issuing the proof

transaction without even knowing whether the transaction is valid or not. Implicitly, this would enable a denial-of-service attack where an adversary can flood the network with invalid encrypted transactions.

The obvious way to solve this problem is by requiring parties to pay a fee to post an encrypted transaction. This, however, creates an inherent fairness problem. The encrypted transaction would now be connected to whatever account is paying the fee. Using the same account twice to pay the fees for two different transactions would violate fairness, as this leaks unwanted information (i.e., the same account has issued two transactions). We solve this problem by employing an account-based protocol inspired by Zerocash [BCG+14], that is secure against adaptive corruption. Our mechanism works as follows.

A party, to issue encrypted transactions, needs to have a *private account*. Once the money is transferred to this private account, the fees can be paid from such account. We call this a private account because every transaction that appears on the blockchain, whose fee comes from a private account, would only leak the fact that the transaction is issued from one of the potentially many private accounts created. No information about what the specific account is is leaked. Clearly, the more private accounts we have, the bigger the anonymity set, and the better the fairness is.

To create a private account with an initial balance B, a party samples a random $\mathsf{sk}, r \in \mathbb{Z}_q$ and creates a commitment $\mathsf{com} = g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}}$ (h_1 and h_2 are public setup parameters), and publishes on the blockchain the commitment, along with a zero-knowledge proof that the commitment is well formed and computed accordingly to the balance B. The committed data is also added to a Merkle tree, whose root is on the blockchain (this Merkle tree contains all the commitments generated so far by all the parties running the protocol). The blockchain accepts the commitment only if the issuer of the transactions has at least B coins in this public account and if the zero-knowledge proof is accepted. If the commitment is accepted, then the balance of the public account of the party is reduced by B (which is transferred to the private account).

To pay a fee using the private account the party P first computes the fee of this transaction $v = \mathsf{fee}(\mathtt{tx})$, then it randomly samples $\mathsf{sk}', r' \in \mathbb{Z}_q$ and creates a new commitment $\mathsf{com}' = g^{r'} \cdot h_1^{\mathsf{B}-v} \cdot h_2^{\mathsf{sk}'}$; in addition, P also reveals a tag $\mathsf{tag} = h_3^{\mathsf{sk}}$ which nullifies the previous account state (to prevent double-spending). All this information is posted on the blockchain, along with a zero-knowledge proof that asserts (i) the well-formedness of the previous account state com and the new state com' with balance difference v; (ii) com appears as a leaf in the Merkle-Tree; (iii) tag is well-formed with respect to the previous commitment key sk. If the transaction is accepted, com' is added to the Merkle tree, and the protocol terminates. To pay the next fee, the party simply needs to iterate the above process but using com' instead of com.

We note that the commitments are simple Pedersen commitments. Intuitively, this means that the balance updates are not visible to the adversary. Moreover, the zero-knowledge proof guarantees that the adversary cannot connect com to com', hence, it is not possible to understand from which account the money is coming.

Security against adaptive corruption is guaranteed by the security of the zero-knowledge scheme and the following. The simulator, when asked to create an account, will simply generate com, as the Pedersen commitment of a random value, and fake the zero-knowledge proof. Any payment will just generate a new Pedersen commitment of a random value that will be added to the Merkle tree. Upon corruption, the simulator receives all the balances of the private account. At this point, the simulator (using the knowledge of h_1 and h_2) equivocates the Pedersen commitments to whatever balance values it has received and returns the equivocated randomness and balances to the adversary as part of its internal state.

Impact on the ideal world. To accommodate this additional critical aspect, we also modify

our ledger functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$. The ledger functionality maintains for each party an address of a private account. Any time a party wants to issue a transaction, based on the flat fee mechanism, a corresponding amount of coin will be subtracted from the private account. In a nutshell, the functionality maintains a list of accounts and allows parties to issue transactions fairly, only if these accounts have enough money. The functionality admits an additional command that is used to load money into this virtual account. For more details on how the functionality works, we refer to the technical part of the paper.

3 Preliminaries

3.1 The GUC Framework

We provide our protocols and security proofs in Canetti's universal composition (UC) framework [Can00]. We assume that the reader is familiar with simulation-based security and has basic knowledge of the (G)UC framework.

We provide a brief review of all aspects of the execution model that are needed for our protocols and proof, but omit some of the low-level details and refer the interested reader to relevant works wherever appropriate. Recall the mechanics of activations in UC: In a UC protocol execution, an honest party (ITI) gets activated either by receiving an input from the environment, or by receiving a message from one of its hybrid functionalities (or from the adversary). Any activation results in the activated ITI performing some computation on its view of the protocol and its local state, and ends with either the party sending a message to some of its hybrid functionalities, sending an output to the environment, or not sending any message at all. In any of these cases, the party loses the activation (and in the latter case the activation goes to the environment by default). We denote the identities of parties by P_i, i.e. $\mathsf{P}_i = (\mathrm{pid}_i, \mathrm{sid}_i)$, and call P_i a party for short. The index i is used to distinguish two identifiers, i.e., $\mathsf{P}_i \neq \mathsf{P}_j$, and otherwise carries no meaning. We will assume a central adversary $\mathcal{A}$ who gets to corrupt miners and might use them to attempt to break the protocol's security. As is common in (G)UC, the resources available to the parties are described as hybrid functionalities (which we detail soon). Our protocols are synchronous (G)UC protocols [KMTZ13]: parties have access to a (global) clock setup, denoted by $\mathcal{G}_{\mathsf{Clock}}$ and can communicate over a network diffuse functionality $\mathcal{F}_{\mathsf{Diffuse}}$. We next sketch its main components: All functionalities, protocols, and setups have a dynamic party set. I.e., they all include special instructions allowing parties to register and de-register, and allow the adversary to learn the current set of registered parties. Additionally, global setups allow any other setup (or functionality) to register and de-register with them and also allow other setups to learn their set of registered parties.

Clock and diffusion functionality. Following the treatment in [BMTZ17], we model the synchronous processors and bounded-delay network as $\mathcal{G}_{\mathsf{Clock}}$ and $\mathcal{F}_{\mathsf{Diffuse}}$ respectively (except for transaction-related messages, details see Section 4.1). The global clock $\mathcal{G}_{\mathsf{Clock}}$ maintains a round index for each session, and this index can only be forwarded when all registered honest parties (in this session) have finished their computation for this round. The diffuse functionality $\mathcal{F}_{\mathsf{Diffuse}}$ with delay Δ guarantees that when a message has reached at least one honest party at round r, it will be delivered to all honest parties before round $r + \Delta$ (This captures that a message is diffused to all honest parties by echoing). A detailed description of these functionalities is presented in Appendix A.

Global random oracles. Our constructions and proofs are based on different kinds of global random oracle functionalities (see Appendix A). The hash function H used in the PoS blockchain is modeled as a global random oracle $\mathcal{G}_{\mathsf{RO}}$. We also adopt the restricted programmable and observable

global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ following the modelling in [CDG+18] for our PKE and NIZK schemes. We allow the adversary to observe and program the $\mathcal{G}_{\mathsf{rpoRO}}$; meantime, every party (in the same session) can check if a point is programmed by calling the "IsProgrammed" interface.

3.2 Receiver Order Fairness

We briefly review basics of receiver order fairness about profiles, dependency graphs and various definitions. The notations that we adopt follows those in [KLS24] and we refer the reader to [KLS24] for details.

Let $\mathbb{T}$ denote the (finite) set of all possible transactions with elements $\mathtt{tx}$. A *transaction profile* (or "profile" for short) is a bijection $\mathsf{R} : \mathbb{T} \to [m]$ where $m = |\mathbb{T}|$. We adopt "$\prec$" to describe the "order before" relation on $\mathbb{T} \times \mathbb{T}$. In receiver fairness based on local relative order, we adopt $\varphi \in \mathbb{R}^+$ as the fairness parameter. We say $\mathtt{tx} \prec^{\varphi} \mathtt{tx}'$ if, for a set of profiles, $\mathtt{tx} \prec \mathtt{tx}'$ holds in at least φ fraction of the profiles. An $(\mathcal{R}, \varphi)$-dependency-graph is a directed graph $G_{\mathcal{R},\varphi}$ constructed from $\mathcal{R} = (\mathsf{R}_1, \mathsf{R}_2, \ldots, \mathsf{R}_n)$ as follows. For each transaction $\mathtt{tx}_i$, add a vertex v_i to $G_{\mathcal{R},\varphi}$; then, for any pair of vertices $\mathtt{tx}_i, \mathtt{tx}_j$, add an edge (v_i, v_j) if $\mathtt{tx}_i \prec^{\varphi} \mathtt{tx}_j$.

Receiver order fairness can be roughly classified into two categories: The "timed" variant [Kur20, ZSC+20] which defines order based on the local timestamps (or indices) they are assigned by protocol participants; and the "local-order" variant [KZGJ20, KLS24] which defines a fair order based on transaction profiles. Notably, defining fairness based on profiles can be viewed as applying a function F on the transaction dependency graph.

3.3 Cryptographic Building Blocks

We denote the security parameter by κ and use "$\|$" as string concatenation.

Commitment schemes. We use the Pedersen commitment scheme [Ped92] which is perfectly hiding and computationally binding assuming the hardness of discrete logarithm problem. Let $\langle g \rangle$ be a cyclic group of prime order q and $h \xleftarrow{\$} \langle g \rangle \backslash \{1\}$ denote a random group element. To commit a message m, the committer first samples a $r \xleftarrow{\$} \mathbb{Z}_q$ and then gives out $\mathsf{com} = g^r h^m$. A generalized Pedersen commitment scheme makes it possible to commit to n messages $(m_1, \ldots, m_n)$ at once, by adopting n group elements $h_1, \ldots h_n$ and computing $\mathsf{com} = g^r \Pi_{i=1}^n h_i^{m_i}$.

Secret sharing. We use the following notion for Shamir secret sharing [Sha79]. To share a secret $s \in \mathbb{Z}_q$ in a t-out-of-n fashion, we write $\{s_i\}_{i\in[n]} \leftarrow \mathsf{SS}_t(s)$ as choosing uniform coefficients $f_1, \ldots, f_t \in \mathbb{Z}_q$ to form the polynomial $f(x) = s + \sum_{i=1}^t f_i \cdot x^i$, and setting $s_i \leftarrow f(i)$. Slightly abusing the notation, we write $\mathsf{SS}_t(s; f)$ to denote the shares acquired by secret sharing with a given polynomial f of degree t.

To reconstruct, we write $s \leftarrow \mathsf{interpolate}(S, \{s_i\}_{i\in S})$ as computing $f(0) = \sum_{i\in S} \lambda_{i,0}^S \cdot f(i)$ for a $(t+1)$-size set $S \subset \mathbb{Z}_q$. This scheme guarantees that no information about s is leaked by any subset of shares with size no larger than t, and s can be reconstructed from any $(t+1)$-size subsets.

Adaptively anonymous PKE. We employ a PKE scheme with anonymity against selective-opening attacks. Roughly speaking, this requires that the adversary, given a set of public keys $\mathsf{pk}_1, \ldots \mathsf{pk}_n$ and ciphertexts $c_1, \ldots c_m$ (each encrypted under a public key chosen randomly from $\mathsf{pk}_1, \ldots \mathsf{pk}_n$), cannot open more than fm keys that were used to encrypt after he adaptively corrupts and learns fn secret keys.

Concretely, we adopt the scheme of unique-recipient key-and-message non-committing encryption scheme based on ElGamal in the random oracles model due to Canetti *et al.* [CKR+23] and

denote this scheme as anonPKE. We present anonPKE (Algorithm 6) and its related security properties in Appendix A.1.

Adaptive UC NIZK. We model the NIZK proof system that we will use as $\mathcal{F}_{\mathsf{NIZK}}$ (Functionality 5). For our construction we use an adaptively secure NIZK proof system due to Canetti, Sarkar and Wang [CSW22], which provides full adaptive soundness, as well as adaptive zero-knowledge in the CRS model.

3.4 Proof-of-Stake Blockchains

Our protocol is built on top of a Nakamoto-style longest chain Proof-of-Stake (PoS) protocol specifically Ouroboros-Genesis [BGK+18], which achieves composable security with dynamic availability against adaptive corruption. We defer the details of this protocol to Section 5.3 and briefly review its basic notations.

In Ouroboros-Genesis, time is divided into subsequent, non-overlapping, equally long intervals called slots. The protocol proceeds with epochs which are collections of R slots. A block $\mathcal{B}$ is of form $(h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$ where h is a hash of the previous block, $\mathtt{st}$ is the encoded data of $\mathcal{B}$, and $\mathtt{sl}$ is its slot number. The value $crt = (\mathsf{P}, y, \pi)$ certifies that $\mathcal{B}$ was indeed proposed by an eligible slot leader P for slot $\mathtt{sl}$ via the output y of P's VRF evaluation and its corresponding proof π, and $\rho = (y_\rho, \pi_\rho)$ is an independent VRF output used to derive the future epoch randomness. Finally, σ is the signature by P on the entire block. The first block $\mathcal{B}_0$ is referred to as the genesis block $\mathbf{G}$. We denote the length of a chain $\mathcal{C}$ by $\mathrm{len}(\mathcal{C})$. For a chain $\mathcal{C}$ and an interval of slots $I \triangleq [\mathtt{sl}_i, \mathtt{sl}_j]$, we denote by $C[I] = C[\mathtt{sl}_i : \mathtt{sl}_j]$ the sequence of blocks in $\mathcal{C}$ such that their slot numbers fall into the interval I. The protocol runs with hybrid functionalities — $\mathcal{F}_{\mathsf{INIT}}$ for initializing genesis block, $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$ that idealizes verifiable random function (VRF) and key-evolving signature (KES) schemes respectively. We provide a detailed description of these functionalities in Appendix A.2.

4 Extended Ledger Functionality

We extend the fair-order ledger functionality $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$ in [CKS24] towards capturing receiver order fairness and a more refined notion of input causality. We denote our new ledger functionality as $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$, and our adaption consists of two aspects. On one hand, in Section 4.1, we revise the functionality so that it bookkeeps all events of honest party receiving transactions, by registering onto a *global* transaction diffusion network $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$. Then, we show how the ledger (precisely, its extend policy) can make use of these newly recorded transaction profiles to run validity checks on block proposals from the adversary. On the other hand, in Section 4.2 we introduce a transaction fee mechanism which imposes restrictions on parties' capability to send transactions based on the current ledger state. In more detail, this is achieved by incorporating a private state into the ledger that parties can interact with via issuing transactions. In Section 4.3, we instantiate two receiver order fairness policies — order linearizability [ZSC+20] and bounded unfairness [KLS24].

Ledger functionality with parametric extend policy. As a warm-up, we begin with a brief overview of the extend policy of the original ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$ [BMTZ17] and its extension in $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$ [CKS24]. In [BMTZ17], the ledger checks adversarial block proposals using ExtendPolicy which, in terms of transaction validity, runs a state-buffer validation predicate that mainly verifies if an upcoming transaction remains valid with respect to the ledger state (this captures the Bitcoin ledger where no order policy is enforced).

In [CKS24], such transaction validation mechanism is further extended by inserting a parametric ValidOrder predicate into the ExtendPolicy that implements customized transaction order validation

policy. Recall that the ledger bookkeeps all sending timestamps in its transaction buffer, such parametric feature enables *sender* order fairness, by checking in $\mathsf{ValidOrder}$ if a transaction sent to ledger at time τ precedes any transaction sent before time $\tau - \delta$. Here, δ is a parameter that reflects the "quality" of order fairness that ledger achieves (it degrades to the original validation predicate if $\delta > L$ where L is the ledger liveness parameter.). In addition, the ledger of [CKS24] achieves input causality by hiding transaction content from the adversary (though, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ still leaks their length to $\mathcal{A}$) until their order is committed.

4.1 Transaction Order Bookkeeping

In order to enforce transactions being serialized with receiver-side order fairness, the ledger shall learn profiles of all honest parties. I.e., their local order and corresponding timestamps[7]. Note that, with a local diffusion functionality $\mathcal{F}_{\mathsf{Diffuse}}$ (which all existing ledgers assume that transactions are diffused with), it is impossible for the ledger to understand the events of transactions received by honest parties in that the ledger can neither learn this information from the dummy parties as it is not part of the environment input, nor extract it from the diffuse functionality which does not exist in the ideal world.

To tackle this issue, we separate the transaction propagation mechanism from other types of protocol messages, and introduce a *global* diffusion functionality $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ to handle all transaction-related messages. Looking ahead, when a transaction `tx` is submitted in the clear (i.e., causality is not of concern), $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ processes `tx` in the clear in both the real and ideal world; otherwise, with encrypted transactions, in our treatment $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ will diffuse these encrypted messages in the real world, and random strings (as an idealization of encrypted transactions) in the ideal world which we detail soon.

Our global diffusion functionality $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ has the same interface as local diffuse networks $\mathcal{F}_{\mathsf{Diffuse}}$ used in previous works, and in the real world they behave identically that deliver transactions subject to delays posed by the adversary up to a bounded number of Δ rounds. Their major difference lies at two aspects. On one hand, $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ is global thus exists in the ideal world, and since $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ is shared among all concurrent protocol instances, it handles transactions per session. On the other hand, $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ allows functionalities (e.g., the ledger) to register and learn the events that a set of the transactions has been delivered to a specific honest party. Under the circumstance that multiple transactions are about to deliver to a party P in one fetch request, the registered functionality learns exactly the same order as those messages written on the tape of P. When honest parties and the functionality $\mathcal{F}$ are additionally registered with the global clock $\mathcal{G}_{\mathsf{Clock}}$, $\mathcal{F}$ learns the specific time that a sequence of transactions has been delivered to an honest party. Note that $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ can pose restrictions on what kind of functionality is "legitimate" to register, by checking the extended id with machine description (refer to UCGS [BCH+20] for more details).

Codewise, $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ maintains a set of registered ledger functionalities with their session id $\{(\mathcal{F}, \mathrm{sid})\}$. The diffuse interface accepts messages from either an honest party, or a registered functionality on behalf of an honest party in the same session. Once a sequence of messages $(m_1, \ldots, m_\ell)$ is delivered to an honest party $\mathsf{P} = (\cdot, \mathrm{sid})$, $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ leaks them to all registered functionalities of the same session id. We present $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ in Functionality 1, and highlight the part that leaks local message receiving events to the registered functionalities.

[7]Different receiver order fairness are defined on different local events, therefore they might concern only relative order or transaction timestamps. In order to facilitate our parametric order policy, we design the ledger so that it bookkeeps the full information of transaction receipt.

Functionality $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$

Initialization: The functionality initializes the party set $\mathcal{P} \leftarrow \emptyset$, functionality set $F \leftarrow \emptyset$ and a list (of messages) $\mathbf{M} \leftarrow \varepsilon$.

Registrations:

- Upon receiving $(\text{REGISTER}, \mathsf{sid})$ from some party $\mathsf{P} = (\mathsf{pid}, \mathsf{sid})$ (or from $\mathcal{A}$ on behalf of a corrupted party), set $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathsf{P}\}$ and return $(\text{REGISTERED}, \mathsf{sid}, \mathsf{P})$ to the caller.
- Upon receiving $(\text{DE-REGISTER}, \mathsf{sid})$ from some party $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted party), set $\mathcal{P} \leftarrow \mathcal{P} \backslash \{\mathsf{P}\}$ and return $(\text{DE-REGISTERED}, \mathsf{sid}, \mathsf{P})$ to the caller.

Network capabilities:

- Upon receiving $(\text{DIFFUSE}, \mathsf{sid}, m)$ from some $\mathsf{P}_s = (\cdot, \mathsf{sid}) \in \mathcal{P}$ (or from $(\mathcal{F}, \mathsf{sid})$ on behalf of P_s), let $\mathcal{P}_{\mathsf{sid}} = \{\mathsf{P}_1 = (\cdot, \mathsf{sid}) \ldots, \mathsf{P}_n = (\cdot, \mathsf{sid})\}$ denote the current party set, do:
  1. Choose n new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$.
  2. Initialize $2n$ new variables $D_{\mathsf{mid}_1} := D^{MAX}_{\mathsf{mid}_1} \ldots := D_{\mathsf{mid}_n} := D^{MAX}_{\mathsf{mid}_n} := 1$ and a per message-delay $\Delta_{\mathsf{mid}_i} = \Delta$ for $i \in [n]$.
  3. Set $\mathbf{M} := \mathbf{M} \parallel (m, \mathsf{mid}_1, D_{\mathsf{mid}_1}, \mathsf{P}_1) \parallel \ldots \parallel (m, \mathsf{mid}_n, D_{\mathsf{mid}_n}, \mathsf{P}_n)$.
  4. Send $(\text{DIFFUSE}, \mathsf{sid}, m, \mathsf{P}_s, (\mathsf{P}_1, \mathsf{mid}_1), \ldots, (\mathsf{P}_n, \mathsf{mid}_n))$ to the adversary.
- Upon receiving $(\text{FETCH}, \mathsf{sid})$ from $\mathsf{P} \in \mathcal{P}$, or from $\mathcal{F}$ that on behalf of an honest party P, or from $\mathcal{A}$ on behalf of a corrupted party P:
  1. For all tuples $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P}) \in \mathbf{M}$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.
  2. Let $\mathbf{M}^{\mathsf{P}}_0$ denote the subvector $\mathbf{M}$ including all tuples of the form $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ with $D_{\mathsf{mid}} \leq 0$ (in the same order as they appear in $\mathbf{M}$). Delete all entries in $\mathbf{M}^{\mathsf{P}}_0$ from $\mathbf{M}$ and in case some $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ is in $\mathbf{M}^{\mathsf{P}}_0$, where P is honest, set $\Delta_{\mathsf{mid}'} = \Delta$ for any $(m, \mathsf{mid}', D_{\mathsf{mid}'}, (\cdot, \mathsf{sid}))$ in $\mathbf{M}$ and replace this record by $(m, \mathsf{mid}', \min\{D_{\mathsf{mid}'}, \Delta\}, \mathsf{P}')$.
  3. Send $\mathbf{M}^{\mathsf{P}}_0$ to P and if P is honest and $\mathcal{F}$ fetches on behalf of P, then send $\mathbf{M}^{\mathsf{P}}_0$ also to $\mathcal{F}$.

Additional adversarial capabilities:

- Upon receiving $(\text{DIFFUSE}, \mathsf{sid}, m)$ from some corrupted $\mathsf{P}_s \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P_s if corrupted), do execute it the same way as an honest-sender diffuse, with the only difference that $\Delta_{\mathsf{mid}_i} = \infty$.
- Upon receiving $(\text{DELAYS}, \mathsf{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\mathsf{mid}_{i_j}}, \mathsf{mid}_{i_j})$: if $D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}} \leq \Delta_{\mathsf{mid}_{i_j}}$ and mid_{i_j} is a message-ID of receiver $\mathsf{P} = (\cdot, \mathsf{sid})$ registered in the current $\mathbf{M}$, set $D_{\mathsf{mid}_{i_j}} := D_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$ and set $D^{MAX}_{\mathsf{mid}_{i_j}} := D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$; otherwise, ignore this pair.
- Upon receiving $(\text{SWAP}, \mathsf{sid}, \mathsf{mid}, \mathsf{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\mathbf{M}$, then swap the triples $(m, \mathsf{mid}, D_{\mathsf{mid}}, (\cdot, \mathsf{sid}))$ and $(m, \mathsf{mid}', D_{\mathsf{mid}'}, (\cdot, \mathsf{sid}))$ in $\mathbf{M}$. Return $(\text{SWAP}, \mathsf{sid})$ to the adversary.
- Upon receiving $(\text{GET-REGISTERED}, \mathsf{sid})$ from $\mathcal{A}$, return the response $(\text{GET-REGISTERED}, \mathsf{sid}, \{\mathsf{P}_1 = (\cdot, \mathsf{sid}) \ldots, \mathsf{P}_n = (\cdot, \mathsf{sid})\})$ to $\mathcal{A}$.

Functionality 1: *The global diffuse functionality.*

We now elaborate on the transaction diffusion mechanism in the ideal world. To start with, we further extend the fair-order ledger functionality $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$ in [CKS24] by introducing two additional internal variables $\mathtt{profiles}$ and $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$. The variable $\mathtt{profiles}$ is an append only list that records tuples of form $(\mathsf{P}, \mathtt{tx}, \tau)$, indicating that a transaction $\mathtt{tx}$ is received by a registered honest party P at time τ. And $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$ is a mapping from $\mathtt{tx}$ in the transaction buffer to a random string m. Upon initialization, the ledger sets $\mathtt{profiles}$ and $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$ as empty and whenever the honest (and synchronized) party set $\mathcal{H} \neq \emptyset$, $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ maintains the invariant that it is registered as a functionality

to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$.

Upon a new transaction $\mathtt{tx}$ is submitted from party P_s to our ledger, $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ adds $\mathtt{tx}$ to $\mathtt{buffer}$ if it is valid and, depending on whether the transaction is sent privately (refer to Section 4.2 for more details), $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$ performs the following operations accordingly. When $\mathtt{tx}$ is sent in the clear, $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ leaks $\mathtt{tx}$ to to $\mathcal{A}$ and if additionally P_s is honest, $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ submits $\mathtt{tx}$ to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ on behalf of P_s ($\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ will accept this request as P_s is an honest party that shares the same session identifier with $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$). When $\mathtt{tx}$ is issued privately, $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ leaks only the size of $\mathtt{tx}$ to $\mathcal{A}$ and waits for a response from $\mathcal{A}$ with a set of random strings $\mathbf{M}$. For each random string $m \in \mathbf{M}$, the ledger first records the mapping from $\mathtt{tx}$ to m in $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$. Moreover, in case that P_s is honest, $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ submits m to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ on behalf of P_s. Note that, in this procedure the ledger learns all random strings that are associated with a transaction $\mathtt{tx}$, including those sent from malicious parties.

Regarding bookkeeping local transaction receiving time, our ledger takes charge of transaction fetching for dummy parties and opens a new interface for receiving messages from $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$. Specifically, when a (MAINTAIN-LEDGER, sid, minerID) command is received by an honest party P for the first time at round τ (which can be checked through the bookeeped honest input sequence $\vec{\mathcal{I}}^{T}_{H}$), our ledger, before forwarding this command to $\mathcal{A}$, sends the FETCH command to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ on behalf of P. Upon receiving as reply (FETCH, sid, $\mathbf{M}$) where $\mathbf{M}$ is a vector of strings, for each string $m \in \mathbf{M}$ with order, $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ first checks if a transaction $\mathtt{tx}$ in buffer maps to m (i.e., $\exists \mathtt{tx}$ s.t. $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}(\mathtt{tx}) = m$). If such transaction exists (or m by itself is a transaction $\mathtt{tx}$), it appends a tuple $(\mathsf{P}, \mathtt{tx}, \tau_{\mathcal{L}})$ to $\mathtt{profiles}$ (recall that the ledger syncs with $\mathcal{G}_{\mathsf{Clock}}$ as the first step of every activation). For the second and later invocations of P on input MAINTAIN-LEDGER in the same round τ, these fetch operations are ignored.

Armed with these modifications, the profile of an honest party P is defined as the sequence $((\mathtt{tx}_1, \tau_1), \ldots, (\mathtt{tx}_k, \tau_k))$ by extracting all tuples $(\mathsf{P}, \mathtt{tx}_i, \tau_i)$ from $\mathtt{profiles}$ with order. And at any time of the execution, ledger $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ keeps track of the transaction profiles and dependency graphs. When the adversary is to commit a block proposal with a list of transactions, the ledger runs $\mathsf{ExtendPolicy}$ to check the validity of transaction order making use of these profiles.

4.2 Transaction Fees for Input Causality

In order to prevent the adversary $\mathcal{A}$ from censoring a transaction $\mathtt{tx}$ on the fly and then adaptively injecting his own transactions, the ledger functionality in [CKS24] leaks only the transaction size to $\mathcal{A}$ before $\mathtt{tx}$ gets settled in the ledger state (a.k.a. input causality). In real world protocols, causality is often achieved via issuing encrypted transactions; nevertheless, while naïvely encrypting all transactions does realize input causality, it incurs a new practical attacking vector that the adversary keeps sending junks to stall the ledger.[8]

To solve this issue, we introduce a transaction fee mechanism with our new ledger functionality $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ in order to model a client's capability to send transactions, based on her past transactions in the current ledger state. Note that to avoid unnecessary complications, our treatment focus on the fee system only while omitting all other details on transaction syntax. We also, without loss of generality, assume a *flat* transaction fee design — i.e., the fee of a transaction $\mathtt{tx}$ is set as a deterministic function fee of $\mathtt{tx}$.

Submitting a transaction. Recall previous ledger functionalities [BMTZ17, CKS24], upon the environment $\mathcal{Z}$ sends a client P a submit transaction command (SUBMIT, sid, $\mathtt{tx}$), the dummy party

[8] An alternative way to mitigate the flooding attack is to accept transactions with fresh PoW only. Yet, we choose not to explore this direction as it imposes more restrictions on a party's capability to send transactions and is less adopted than the fee mechanism.

P forwards this command to the ledger and the functionality processes this transaction and possibly leaks $\mathtt{tx}$ to the adversary. We here extend such mechanism by allowing the environment to specify a private account that it would like to fund or pay transaction fees and introducing a special type of "funding" transactions.

Specifically, now the new transaction submission command $(\text{SUBMIT}, \mathsf{sid}, \mathtt{tx}, \mathsf{acc})$ admits an additional (possibly empty) parameter acc. If $\mathsf{acc} = \bot$, it indicates that the transaction is not sent privately and the ledger simply leaks the whole transaction $\mathtt{tx}$ to the adversary; otherwise, the ledger leaks only the size of $\mathtt{tx}$ (except for the funding transactions that we will detail soon). In other words, input causality is achieved for a subset of transactions that are willing to pay the fees privately (by setting $\mathsf{acc} \neq \bot$ in the input).

To facilitate such fee mechanism, the ledger should allow parties to fund their private accounts. This is achieved via parties issuing a special type of funding transaction, which burns B native tokens controlled by them on the ledger and specifies an account acc to add balance to. We write the command to submit such special funding transaction as

$$(\text{SUBMIT}, \mathsf{sid}, ((0^\kappa, \mathsf{B}, \mathsf{aux}), \mathsf{acc})),$$

where 0^κ indicates that B tokens are to be added to acc, and aux contains a tag showing whether to set-up or update account acc and includes all necessary information to prove that B native tokens are indeed burnt. All funding transactions are sent "in the clear" — i.e., the ledger functionality simply leaks $(0^\kappa, \mathsf{B}, \mathsf{aux})$ to the adversary (but not acc).

On ledger state, blockify and validate. The ledger state in previous works is a list of "blockified" list of transactions — i.e. $\mathtt{state} = \mathsf{Blockify}(\mathtt{tx}_{11}, \ldots \mathtt{tx}_{1n_1}) \parallel \ldots \parallel \mathsf{Blockify}(\mathtt{tx}_{\ell 1}, \ldots \mathtt{tx}_{\ell n_\ell})$ (where $\mathsf{Blockify}$ adds all metadata information about the block). We here extend the structure of $\mathtt{state}$ by replacing the occurrences of all transactions with pairs of a transaction $\mathtt{tx}$ and its associated private account acc (possibly $\bot$).

Any ledger state $\mathtt{state}$ maps to a private account table $\mathsf{PrivAccounts}$. An empty state ε maps to an empty table; and for any state $\mathtt{state}$, appending a fund transaction $((0^\kappa, \mathsf{B}, \mathsf{aux}), \mathsf{acc})$ increases the account balance by B and appending a private transaction $\mathtt{tx}$ using fee account acc decreases its balance by $\mathsf{fee}(\mathtt{tx})$. We write $\mathsf{PrivAccounts}(\mathtt{state}, \mathsf{acc})$ as the balance of an account acc with state $\mathtt{state}$.

$$\mathsf{PrivAccounts}(\mathtt{state} \parallel ((0^\kappa, \mathsf{B}, \mathsf{aux}), \mathsf{acc}), \mathsf{acc}) = \mathsf{PrivAccounts}(\mathtt{state}, \mathsf{acc}) + \mathsf{B}.$$
$$\mathsf{PrivAccounts}(\mathtt{state} \parallel (\mathtt{tx}, \mathsf{acc}), \mathsf{acc}) = \mathsf{PrivAccounts}(\mathtt{state}, \mathsf{acc}) - \mathsf{fee}(\mathtt{tx}).$$

To capture the desiderata that a private transaction always spends a valid private account with sufficient balances, we require that the transaction validation predicate guarantees that no private account can be overspent. I.e., the predicate that we will use in $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ (which we write $\mathsf{Validate}^{\mathsf{ACC}}$) not only checks the validity of all transactions as the ones used in previous works (in case of a funding transaction it checks if there are sufficient native tokens to be burnt), but also rejects all transactions $(\mathtt{tx}, \mathsf{acc})$ that will update the balance of acc to negative. More formally,

$$\mathsf{Validate}^{\mathsf{ACC}}((\mathtt{tx}, \mathsf{acc}), \mathtt{state}, \mathtt{buffer}) = 1 \implies \forall \mathsf{acc}', \mathsf{PrivAccounts}(\mathtt{state} \parallel (\mathtt{tx}, \mathsf{acc}), \mathsf{acc}') \geq 0.$$

On leaking ledger state and transaction buffer. All operations regarding the private accounts, except the amount of balances B funded into a private account and paid for a private transaction, should remain invisible to the adversary (otherwise it hazards input causality). Yet, in order for the ledger to validate incoming transactions, all information needs to be stored in $\mathtt{state}$ transparently

and the previous ledger design yields undesirable consequences in that it leaks the entire $\mathtt{state}$ to the adversary. Hence, our final step is to make part of the ledger state "private" and never leaked to the adversary. Towards this goal, we write $\mathtt{state}^*$ as a "converted" $\mathtt{state}$ such that it removes all occurrences regarding the private accounts — i.e., each block contains only a list of transactions (which is exactly the same structure as in previous works [BMTZ17, CKS24]). Then, in the ledger functionality, whenever the ledger state is leaked to the adversary, it is replaced with $\mathtt{state}^*$; and similarly it applies to honest parties after setting state-slackness.

Regarding the transaction buffer, when it is queried by a READ command by $\mathcal{A}$, we replace all transactions that are not funding transactions and have set a non-empty private account.

$$\mathtt{buffer}^* \triangleq \{(|\mathtt{tx}|, \mathrm{txid}, \tau_{\mathcal{L}}, \mathsf{P}) \in \mathsf{privBTX}\} \cup \{(\mathtt{tx}, \mathrm{txid}, \tau_{\mathcal{L}}, \mathsf{P}) \in \mathtt{buffer} \backslash \mathsf{privBTX}\},$$

where $\mathsf{privBTX} = \{\mathtt{BTX} \mid \mathtt{BTX} = (\mathtt{tx}, \mathsf{acc}, \mathrm{txid}, \tau_{\mathcal{L}}, \mathsf{P}) \in \mathtt{buffer} \wedge \mathsf{acc} \neq \bot \wedge \mathtt{tx} \neq (0^{\kappa}, \cdot, \cdot)\}$.

With our new leakage policies, the adversary $\mathcal{A}$ can never learn any information about private accounts from $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$. The only way that $\mathcal{A}$ can learn the private accounts used by a party P is to corrupt P thus accessing its internal state. Looking ahead, this raises challenges in our simulation as the simulator now has to send (faked) messages indicating that fees have been paid from an appropriately updated account; however later when $\mathcal{A}$ adaptively corrupts a party P and learns all private accounts that P has used so far, the simulator needs to equivocate all previous messages in a consistent way of these unknown accounts. We refer to Section 5.1 for our transaction fee protocol and Section 6 for a detailed analysis.

Remark 2. We note that a private account acc as introduced in this section acts as the pseudonym of a party where she can create and own multiple accounts at the same time. In addition, this pseudonym acc is different from the party identifier P which acts as the "network name" of a machine. The adversary is able to learn the source of a transaction on the network (since $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ leaks P when a transaction is submitted); nevertheless $\mathcal{A}$ cannot learn from $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ the account acc that is used to pay the fees.

We present the full ledger functionality in Functionality 2 and the formal specification of ExtendPolicy in Algorithm 1.

Functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$

Parameters: Integers $\mathtt{windowSize}$, $\mathtt{Delay}$, $\mathtt{waitTime}$. Algorithms $\mathsf{Validate}^{\mathsf{ACC}}$, ExtendPolicy, Blockify, and predict-time.

Clock-time: The functionality maintains a variable $\tau_{\mathcal{L}}$ that is kept in-sync with clock-time: Upon any activation (and thus also initialization), the ledger first sends $(\text{CLOCK-READ}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$ and upon receiving response $(\text{CLOCK-READ}, \mathrm{sid}_C, \tau)$ set $\tau_{\mathcal{L}} \leftarrow \tau$, and then proceeds with remaining actions.

Variables and initialization: The functionality initializes $\mathtt{state}$, profiles, s_{ep}, $\mathtt{NxtBC}$, $\vec{\mathcal{I}}_H^T \leftarrow \varepsilon$, $\mathtt{buffer} \leftarrow \emptyset$ as well as party set $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS} \leftarrow \emptyset$.

Party management:

- Upon receiving $(\text{REGISTER}, \mathrm{sid})$ from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} \leftarrow \mathcal{P} \cup \{\mathsf{P}\}$, initialize $\mathtt{pt}_i \leftarrow 1, \mathtt{state}_i \leftarrow \varepsilon$, and $\tau_{\mathsf{P}}^{\mathsf{reg}} \leftarrow \tau_{\mathcal{L}}$. If P is an honest party and if $\mathcal{H} = \emptyset$ send $(\text{REGISTER}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$ and $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$. If P is honest then update $\vec{\mathcal{I}}_H^T$ and set $\mathcal{H} \leftarrow \mathcal{H} \cup \{\mathsf{P}\}$ and if additionally $\tau_{\mathsf{P}}^{\mathsf{reg}} > 0$ holds, set $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \cup \{\mathsf{P}\}$. Return $(\text{REGISTER}, \mathrm{sid}, \mathsf{P})$ to the caller.
- Upon receiving $(\text{DE-REGISTER}, \mathrm{sid})$ from some party $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $\mathsf{P} \in \mathcal{P}$), set $\mathcal{P} \leftarrow \mathcal{P} \backslash \{\mathsf{P}\}$, $\mathcal{H} \leftarrow \mathcal{H} \backslash \{\mathsf{P}\}$ and $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \backslash \{\mathsf{P}\}$. If $\mathcal{H} = \emptyset$, send $(\text{DE-REGISTER}, \mathrm{sid}_C)$

to $\mathcal{G}_{\mathsf{Clock}}$ and $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$. If P is honest then update $\vec{\mathcal{I}}^T_H$. Return $(\text{DE-REGISTER}, \text{sid}, \mathsf{P})$ to the caller.

Upon receiving any input I from a party $\mathsf{P}_i \in \mathcal{P}$ or from the adversary, the following steps are taken:

1. If $\mathsf{P}_i \in \mathcal{H}$ or if I is a corruption message from $\mathcal{A}$ targeting $\mathsf{P}_i \in \mathcal{H}$, then update $\vec{\mathcal{I}}^T_H \leftarrow \vec{\mathcal{I}}^T_H \| (I, \mathsf{P}_i, \tau_{\mathcal{L}})$. If a party P_i gets corrupted, additionally update $\mathcal{H} \leftarrow \mathcal{H} \backslash \{\mathsf{P}\}$ and $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \backslash \{\mathsf{P}\}$.
2. Let $\hat{\mathcal{P}} \leftarrow \{\mathsf{P} \in \mathcal{P}_{DS} \mid \tau^{\text{reg}}_{\mathsf{P}} < \tau_{\mathcal{L}} - \mathtt{Delay}\}$. Set $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \backslash \hat{\mathcal{P}}$.
3. If $\mathsf{P}_i \in \mathcal{H}$ then additionally take the following steps:
   (a) $(\vec{N}, s') \leftarrow \mathsf{ExtendPolicy}(\vec{\mathcal{I}}^T_H, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}; s_{ep})$. Reset $\mathtt{NxtBC} \leftarrow \varepsilon$ and store $s_{ep} \leftarrow s'$.
   (b) If $\vec{N} \neq \varepsilon$ then parse $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell)$ and update $\mathtt{state} \leftarrow \mathtt{state} \| \mathsf{Blockify}(\vec{N}_1) \| \ldots \| \mathsf{Blockify}(\vec{N}_\ell)$.
   (c) For each $\mathtt{BTX} \in \mathtt{buffer}$: if $\mathsf{Validate}^{\mathsf{ACC}}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) = 0$, then $\mathtt{BTX} \leftarrow \mathtt{BTX} \backslash \{\mathtt{BTX}\}$.
   (d) If $\exists \mathsf{P} \in \mathcal{H} \backslash \mathcal{P}_{DS}$ s.t. $\mathtt{pt}_{\mathsf{P}} \notin [|\mathtt{state}| - \mathtt{windowSize} + 1, |\mathtt{state}|]$, then set $\mathtt{pt}_{\mathsf{P}_k} \leftarrow |\mathtt{state}|$ for all $\mathsf{P}_k \in \mathcal{H} \backslash \mathcal{P}_{DS}$.
4. If the input I is a ledger instruction from a party $\mathsf{P}_i \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted party $\mathsf{P}_i \in \mathcal{P}$), execute the respective code:
   - *Submitting a transaction:*
     If $I = (\text{SUBMIT}, \text{sid}, \mathtt{tx}, \mathtt{acc})$ do the following:
     (a) Choose a unique transaction ID txid and set $\mathtt{BTX} \leftarrow (\mathtt{tx}, \mathtt{acc}, \text{txid}, \tau_{\mathcal{L}}, \mathsf{P}_i)$;
     (b) If $\mathsf{Validate}^{\mathsf{ACC}}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer}) = 1$, then $\mathtt{buffer} \leftarrow \mathtt{buffer} \cup \{\mathtt{BTX}\}$;
     (c) If $\mathtt{BTX.acc} = \perp$ or $\mathtt{tx} = (0^\kappa, \cdot, \cdot)$, send $(\text{SUBMIT}, (\mathtt{tx}, \text{txid}, \tau_{\mathcal{L}}, \mathsf{P}))$ to $\mathcal{A}$ and if additionally, P_i is an honest party then send $(\text{DIFFUSE}, \text{sid}, \mathtt{tx})$ to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$. Otherwise, send $(\text{SUBMIT}, (|\mathtt{tx}|, \text{txid}, \tau_{\mathcal{L}}, \mathsf{P}))$ to $\mathcal{A}$ and receive as response $(\text{DIFFUSE}, \mathbf{M})$. For each $m \in \mathbf{M}$, set $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}(\mathtt{tx}) = m$ and if additionally, P_i is an honest party then send $(\text{DIFFUSE}, \text{sid}, m)$ to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$.
   - *Reading the state:*
     If $I = (\text{READ}, \text{sid})$ then do the following: if $\mathsf{P}_i \in \mathcal{H} \backslash \mathcal{P}_{DS}$ then set $\mathtt{state}_i \leftarrow \mathtt{state}|_{\min\{\mathtt{pt}_i, |\mathtt{state}|\}}$. Return $(\text{READ}, \text{sid}, \mathtt{state}^*_i)$ to the caller.
   - *Maintaining the ledger state:*
     If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ and $\mathsf{P}_i \in \mathcal{H}$ and $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}^T_H) > \tau_{\mathcal{L}}$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$. Else, do the following:
     (a) If $(I, \mathsf{P}_i, \tau_{\mathcal{L}})$ appears only once in $\vec{\mathcal{I}}^T_H$, then send $(\text{FETCH}, \text{sid})$ on behalf of P_i to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ and receive as reply $(\text{FETCH}, \text{sid}, \mathbf{M})$. For each $m \in \mathbf{M}$, if $\exists \mathtt{tx}$ s.t. $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}(\mathtt{tx}) = m$ (or, $m = \mathtt{tx} \in \mathtt{buffer}$), set $\mathtt{profiles} \leftarrow \mathtt{profiles} \| (\mathsf{P}, \mathtt{tx}, \tau_{\mathcal{L}})$.
     (b) Send I to $\mathcal{A}$.
5. If the input I is an additional adversarial capability (received on the backdoor tape from $\mathcal{A}$) execute the respective code:
   - *The adversary reading the state:*
     If $I = (\text{READ}, \text{sid})$, then return $(\mathtt{state}^*, \mathtt{buffer}^*, \vec{\mathcal{I}}^T_H)$ to $\mathcal{A}$.
   - *The adversary proposing the next block:*
     If $I = (\text{NEXT-BLOCK}, (\text{txid}_1, \ldots, \text{txid}_\ell), aux)$, update $\mathtt{NxtBC}$ as follows:
     (a) Set listOfTxid $\leftarrow \varepsilon$;
     (b) For $i = 1, \ldots, \ell$ do: if there exists $\mathtt{BTX} = (\cdot, \text{txid}, \tau_{\mathcal{L}}, \mathsf{P}_i) \in \mathtt{buffer}$ with ID txid $= \text{txid}_i$ then set listOfTxid $\leftarrow$ listOfTxid $\| \text{txid}_i$;
     (c) Finally, set $\mathtt{NxtBC} \leftarrow \mathtt{NxtBC} \| (\text{listOfTxid} \| aux)$ and output $(\text{NEXT-BLOCK}, \mathsf{ok})$ to $\mathcal{A}$.
   - *The adversary setting state-slackness:*
     If $I = (\text{SET-SLACK}, (P_{i_1}, \hat{\mathtt{pt}}_{i_1}), \ldots, (P_{i_\ell}, \hat{\mathtt{pt}}_{i_\ell}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{H}$ then do the following:

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

- *The adversary setting the state for desychronized parties:*
  If $I = (\text{DESYNC-STATE}, (\mathsf{P}_{i_1}, \mathtt{state}'_{i_1}), \ldots, (\mathsf{P}_{i_\ell}, \mathtt{state}'_{i_\ell}))$, with $\{\mathsf{P}_{i_1}, \ldots, \mathsf{P}_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\mathtt{state}_{i_j} \leftarrow \mathtt{state}'_{i_j}$ for each $j \in [\ell]$ and return (DESYNC-STATE, ok) to $\mathcal{A}$.
- *The adversary obtaining the set of registered parties:*
  If $I = (\text{GET-REGISTERED}, \text{sid})$, then return $(\text{GET-REGISTERED}, \text{sid}, \mathsf{P})$ to $\mathcal{A}$.
- *The adversary corrupting a party (additional steps to Item 1.):*
  If $I = (\text{CORRUPT}, \text{sid}, \mathsf{P}_i)$ and $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) > \tau_{\mathcal{L}}$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$. Else return I to $\mathcal{A}$.

Functionality 2: *The extended ledger facilitating receiver order fairness and input causality.*

Algorithm $\mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}, \vec{\tau}_{\mathtt{state}})$

▷ We assume call-by-value and hence the function has no side effects.
$\vec{N}_{\text{df}} \leftarrow \mathsf{DefaultExtension}(\vec{\mathcal{I}}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}; s_{ep})$
$\tau_{\mathcal{L}} \leftarrow$ current ledger time (computed from $\vec{\mathcal{I}}_H^T$)
Read $\vec{\tau}_{\mathtt{state}}$ and $\vec{\text{hf}}$ from state s_{ep}. If the state is empty, initialize two empty vectors.
Parse $\mathtt{NxtBC}$ as a vector $((\text{hFlag}_1, \mathtt{NxtBC}_1), \ldots, (\text{hFlag}_n, \mathtt{NxtBC}_n))$
$\vec{N} \leftarrow \varepsilon$
▷ Determine time of the block which is windowSize blocks behind the state head
if $|\mathtt{state}| \geq \mathtt{windowSize}$ **then**
    $\tau_{\mathsf{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 1]$
else
    $\tau_{\mathsf{low}} \leftarrow 0$
end if
for each list $\mathtt{NxtBC}_i$ of transaction IDs **do**
    Use the txid contained in $\mathtt{NxtBC}_i$ to determine the list of transactions
    Let $\vec{\mathtt{tx}} = ((\mathtt{tx}_1, \mathtt{acc}_1), \ldots, (\mathtt{tx}_{|\mathtt{NxtBC}_i|}, \mathtt{acc}_{|\mathtt{NxtBC}_i|}))$ denote the transactions of $\mathtt{NxtBC}_i$ and *aux* the (possibly) appended information
    if $\mathtt{tx}_1$ is not a coinbase trasnaction **then**
        $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_{\mathcal{L}} \parallel \ldots \parallel \tau_{\mathcal{L}}, \vec{\text{hf}} \leftarrow \vec{\text{hf}} \parallel 1 \parallel \ldots \parallel 1$ (extended by $|\vec{N}_{\text{df}}|$ elements) and store the vectors in s_{ep}.
        return $\vec{N}_{\text{df}}$ and new state s_{ep}
    else
        $\vec{N}_i \leftarrow \mathtt{tx}_1$
        for $j = 2$ **to** $|\mathtt{NxtBC}_i|$ **do**
            $\mathtt{st}_i \leftarrow \mathsf{Blockify}(\vec{N}_i)$
            ▷ Default Extension if proposal violates fair order
            if $\mathsf{ValidOrder}(\mathtt{tx}_j, \mathtt{state} \parallel \mathtt{st}_i, \mathtt{buffer}, \mathit{aux}) = \mathsf{false}$ **then**
                $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_{\mathcal{L}} \parallel \ldots \parallel \tau_{\mathcal{L}}, \vec{\text{hf}} \leftarrow \vec{\text{hf}} \parallel 1 \parallel \ldots \parallel 1$ (extended by $|\vec{N}_{\text{df}}|$ elements) and store the vectors in s_{ep}.
                return $\vec{N}_{\text{df}}$ and new state s_{ep}
            end if
            ▷ Default Extension if proposal includes invalid transaction

if $\mathsf{ValidTX}((\mathtt{tx}_j, \mathsf{acc}_j), \mathtt{state} \parallel \mathtt{st}_i) = \mathsf{false}$ **then**
$\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_{\mathcal{L}} \parallel \ldots \parallel \tau_{\mathcal{L}}, \vec{\mathrm{hf}} \leftarrow \vec{\mathrm{hf}} \parallel 1 \parallel \ldots \parallel 1$ (extended by $|\vec{N}_{\mathrm{df}}|$ elements) and store the vectors in s_{ep}.
return $\vec{N}_{\mathrm{df}}$ and new state s_{ep}
end if
$\vec{N}_i \leftarrow \vec{N}_i \parallel \mathtt{tx}_j$
end for
$\mathtt{st}_i \leftarrow \mathsf{Blockify}(\vec{N}_i)$
end if
$\mathrm{hFlag}_i \leftarrow 1$
▷ Test that all old transactions are included.
for each $\mathtt{BTX} = ((\mathtt{tx}, \mathsf{acc}), \mathrm{txid}, \tau', \mathsf{P}) \in \mathtt{buffer}$ of an honest party P with time $\tau' < \tau_{\mathcal{L}} - \mathtt{waitTime}$ **do**
if $\mathsf{ValidTX}((\mathtt{tx}, \mathsf{acc}), \mathtt{state} \parallel \mathtt{st}_i) = \mathsf{true}$ **and** $\mathsf{ValidOrder}(\mathtt{tx}, \mathtt{state} \parallel \mathtt{st}_i, \mathtt{buffer}) = \mathsf{true}$ **and** $\mathtt{tx} \notin \vec{N}_i$ **then**
$\mathrm{hFlag}_i \leftarrow 0$
end if
end for
$\vec{N} \leftarrow \vec{N} \parallel \vec{N}_i$
$\mathtt{state} \leftarrow \mathtt{state} \parallel \mathtt{st}_i$
$\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_{\mathcal{L}}, \vec{\mathrm{hf}} \leftarrow \vec{\mathrm{hf}} \parallel \mathrm{hFlag}_i$ and store those vectors in s_{ep}
if $|\mathtt{state}| \geq \mathtt{windowSize}$ **then**
$\tau_{\mathsf{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 1]$
else
$\tau_{\mathsf{low}} \leftarrow 0$
end if
end for
if $\tau_{\mathcal{L}} < \mathtt{maxTime}_{\mathtt{window}} \wedge \mathtt{state} = \varepsilon$ **then return** ε
$\mathsf{inv} \leftarrow 0$ ◁ invalid flag
if $|\mathtt{state}| < \mathtt{windowSize}$ **then** ◁ Ensure a timely startup with enough honest blocks.
$lr \leftarrow \min_{j=0,\ldots,\tau_{\mathcal{L}}-\mathtt{maxTime}_{\mathtt{window}}; i=j+\mathtt{maxTime}_{\mathtt{window}}\ldots\tau_{\mathcal{L}}} : \frac{\mathtt{windowSize}+|\{n\in[|\vec{\tau}_{\mathtt{state}}|]: j\leq\vec{\tau}_{\mathtt{state}}[n]\leq i\}|}{i-j+1}$
$ar \leftarrow \frac{|\{n\in[|\mathtt{state}|]:\vec{\mathrm{hf}}[n]=0\}|}{\mathtt{windowSize}}$
if $lr < \frac{\mathtt{windowSize}}{\mathtt{maxTime}_{\mathtt{window}}} \vee ar > \eta$ **then** $\mathsf{inv} \leftarrow 1$
end if
if $|\mathtt{state}| \geq \mathtt{windowSize}$ **then** ◁ Ensure ledger growth limits and enough honest blocks.
$lr \leftarrow \min_{s=1,\ldots,|\vec{\tau}_{\mathtt{state}}|-\mathtt{windowSize}+1; t=s+\mathtt{windowSize}-1,\ldots,|\vec{\tau}_{\mathtt{state}}|} : \frac{t-s+1}{\vec{\tau}_{\mathtt{state}}[t]-\vec{\tau}_{\mathtt{state}}[s]+1}$
$hr \leftarrow \max_{s=1,\ldots,|\vec{\tau}_{\mathtt{state}}|-\mathtt{windowSize}+1; t=s+\mathtt{windowSize}-1,\ldots,|\vec{\tau}_{\mathtt{state}}|} : \frac{t-s+1}{\vec{\tau}_{\mathtt{state}}[t]-\vec{\tau}_{\mathtt{state}}[s]+1}$
$ar \leftarrow \max_{s=1,\ldots,|\mathtt{state}|-\mathtt{windowSize}+1; t=s+\mathtt{windowSize}-1,\ldots,|\mathtt{state}|} : \frac{|\{n\in[s\ldots t]:\vec{\mathrm{hf}}[n]=0\}|}{t-s+1}$
if $lr < \frac{\mathtt{windowSize}}{\mathtt{maxTime}_{\mathtt{window}}} \vee hr > \frac{\mathtt{windowSize}}{\mathtt{minTime}_{\mathtt{window}}} \vee ar > \eta$ **then** $\mathsf{inv} \leftarrow 1$
end if
if $\mathsf{inv} = 0$ **then**
return $\vec{N}_{\mathrm{df}}$ and new s_{ep}
else
$\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_{\mathcal{L}} \parallel \ldots \parallel \tau_{\mathcal{L}}, \vec{\mathrm{hf}} \leftarrow \vec{\mathrm{hf}} \parallel 1 \parallel \ldots \parallel 1$ (extended by $|\vec{N}_{\mathrm{df}}|$ elements) and store the vectors in s_{ep}.
return $\vec{N}_{\mathrm{df}}$ and new state s_{ep}
end if

Algorithm 1: *The extend policy for* $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$*, with custom* ValidOrder *predicate.*

4.3 Extend Policies with Receiver Order Fairness

In this section we give two examples of ValidOrder predicates in ExtendPolicy that enforces receiver order fairness by using the `profiles` bookkeeped in the ledger functionality.

Order linearizability. Consider one "timed" variant of receiver order fairness — order linearizability [Kur20, ZSC+20]. This serialization policy enforces a transaction `tx` to be ordered before $\mathtt{tx}'$, if there exists a time t such that all parties receives `tx` before t and $\mathtt{tx}'$ after t. And it gives up on the order of any transaction pairs when there is an overlap between their receiving timestamps.

We provide the ValidOrder predicate of order linearizability in Algorithm 2. For a proposed `tx`, ValidOrder first extracts its earliest receiving timestamp $\tau^{\min}$. Then, if there exist any transaction $\mathtt{tx}'$ in `buffer` such that (i) $\mathtt{tx}'$ has not been proposed into the ledger state; (ii) in `profiles`, all parties have received $\mathtt{tx}'$ before $\tau^{\min}$; and (iii) $\mathtt{tx}'$ remains valid with respect to the proposed ledger state, then ValidOrder rejects the proposal of `tx`, since $\mathtt{tx}'$ should be ordered before `tx`.

While such formalization has been unambiguous with fixed number of parties, the transaction order can be ill-defined with dynamic set of registered parties (and recall that we do allow functionality to be registered with a set of changing parties) if they rapidly change over time. To mitigate, we revise the second condition so that only the profiles of parties that stays once synchronized after $\mathtt{tx}'$ will be concerned[9]. Note that, ledger can check if a party is alert by looking at $\vec{\mathcal{I}}_H^T$ to check how long a party has been registered.

Algorithm ValidOrder$(\mathtt{tx}, \vec{N}, \mathtt{buffer})$

1: Parse `tx` as $(\mathtt{tx}, \mathrm{txid}, \tau, \mathsf{P})$
   ▷ Extract the earliest time `tx` is received by an honest party.
2: $\tau^{\min} \leftarrow \min\{\tau' \mid (\cdot, \mathtt{tx}, \tau') \in \mathtt{profiles}\}$
   ▷ Check if valid transaction before time $\tau^{\min}$ is missing
3: **for** $\mathtt{BTX}.\mathtt{tx}' \in \{\mathtt{BTX} \in \mathtt{buffer} \wedge \mathtt{BTX} \notin \vec{N}\}$ **do**
4: Let $\mathsf{P}_1, \ldots, \mathsf{P}_k$ be parties that are once synchronized after $\mathtt{tx}'$
5: Set $\tau^{\max} \leftarrow \max\{\tau' \mid (\mathsf{P}', \mathtt{tx}', \tau') \in \mathtt{profiles} \wedge \mathsf{P}' \in \{\mathsf{P}_1, \ldots, \mathsf{P}_k\}\}$
6: **if** $\tau^{\max} < \tau^{\min} \wedge \mathsf{ValidTX}(\mathtt{tx}', \mathtt{state}) = \mathsf{true}$ **then return** false
7: **end for**
8: **return** true

Algorithm 2: *The* ValidOrder *algorithm for order linearizability.*

We omit discussing the default extension of this policy, as it can be constructed in various ways (arguably, a good simulator should never let the ledger adopt orders from default extension). For example, ordering transactions based on their earliest timestamp can satisfy order linearizability.

[9]Regarding protocols, one can mitigate such problem by adopting the profile-bootstrapping approach in [KLS24, Section 5]. In short, in order for parties to become "synchronized" (i.e., own all resources to run the protocol), they needs to passively listen to the protocol for sufficiently long time and bookkeep all recent transactions before actively participating in the protocol.

Extend policies with bounded unfairness. We then instantiate another type of receiver order fairness based on local profiles, and give a general treatment for what we call (φ, F)-consistent receiver order fairness. Roughly speaking, an (φ, F)-consistent receiver order fairness requires to output an order $\sigma = F(\mathcal{G}_{\mathcal{R},\varphi})$ where $\mathcal{R} = (\mathcal{R}^{\mathcal{H}}, \mathcal{R}^{\mathcal{A}})$ and $\mathcal{R}^{\mathcal{H}}$ is the profile and $\mathcal{R}^{\mathcal{A}}$ is the profiles proposed by the the adversary. This generalizes the (φ, F, B)-consistent serialization in [KLS24, Definition 2], and it captures a large family of order fairness defined based on local relative orders (e.g., [KZGJ20]). Note that a serialization function F on an input graph G returns a vertex orderings, we omit the details on how these orderings are computed but focus on how the fair-order predicate can make use of them.

The ledger functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ at any time keeps track of the latest transaction dependency graph built from honest profiles. Nonetheless, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ shall not naïvely apply F on the honest dependency graph as such view may be different from honest parties in the real world, where corrupted parties can also send their "profiles". Note that corrupted parties may not stick to the transaction order that they received, thus leaking additional local receiving information from $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ cannot help ledger decide the final order.

To capture the adversarial power on influencing profile-based receiver order fairness, we make the following necessary yet minor adaptions on the ledger functionality and its extend policy. Whenever the ideal world adversary proposes new transactions via NEXT-BLOCK, in addition to committing a list of transaction identifiers, the ledger now accepts an additional auxiliary information aux and appends it to the list of transactions and therefore $\mathtt{NxtBC} = (\mathrm{txid}_1, \ldots, \mathrm{txid}_\ell) \| aux$. Then, regarding the extend policy, once it processes $\mathtt{NxtBC}$, ExtendPolicy also passes such aux to the ValidOrder predicate (see Algorithm 1). We note that such modification works generally, as providing auxiliary information is harmless for ordering policies that does not rely on such information (e.g., the one in Algorithm 2 which simply ignores any aux).

Regarding the policy details, the ValidOrder predicate first constructs the transaction dependency graph $G_{\mathcal{R},\varphi}$ from profiles. Then, if the adversary $\mathcal{A}$ is not proposing a graph G in aux as a *spanning super-graph* of $G_{\mathcal{R},\varphi}$ (i.e., all pairwise preferences are preserved, details see [KLS24, Theorem 3]), ValidOrder rejects such proposal immediately. The ideal order σ is then decided by $\sigma = F(G)$ and the predicate checks if any valid transaction in buffer is missed from the proposal. It accepts the proposed $\mathtt{tx}$, if no valid transaction precedes $\mathtt{tx}$ has not been already proposed by the adversary. Refer to Algorithm 3 for the detailed description.

Algorithm ValidOrder$(\mathtt{tx}, \vec{N}, \mathtt{buffer}, aux)$

Parse $\mathtt{tx}$ as $(\mathtt{tx}, \mathrm{txid}, \tau, \mathsf{P})$
▷ Construct the transaction dependency graph
Construct profiles $\{\mathsf{R}_i\}_{i\in[k]}$ from tuples $(\mathsf{P}_i, \mathtt{tx}, \cdot)$ in $\mathtt{profiles}$
Set $\mathcal{R} = \{\mathsf{R}_i\}_{i\in[k]}$ and compute $G_{\mathcal{R},\varphi}$
▷ Reject if the adversary proposes a bad graph
Set G the graph extracted from aux
if G is not a spanning super-graph of $G_{\mathcal{R},\varphi}$ **then return** false
Set $\sigma \leftarrow F(G)$
for $\mathtt{BTX}.\mathtt{tx}' \in \{\mathtt{BTX} \in \mathtt{buffer} \wedge \mathtt{BTX} \notin \vec{N}\}$ **do**
    if $\mathtt{tx}'$ is ordered before $\mathtt{tx}$ in σ **and** ValidTX$(\mathtt{tx}', \mathtt{state}) =$ true **then return** false
end for
return true

Algorithm 3: *The* $\mathsf{ValidOrder}$ *predicate for* (φ, F)*-consistent receiver order fairness.*

5 Protocol Details

We elaborate on our protocol $\Pi_{\mathsf{ledger}}^{\mathsf{fair++}}$ in this section, which is built "on top of" the following ingredients. First, we present an account-based design that allows parties to pay transaction fees privately in Section 5.1. In Section 5.2, we propose a YOSO-style protocol $\Pi_{\mathsf{ss\text{-}tx}}$ that allows a transaction issuer to publicly and verifiably secret-share her transactions to an ephemeral committee, and later all protocol participants will be able to reconstruct those transactions. Next, we adopt $\mathsf{Ouroboros\text{-}Genesis}$ [BGK+18] as the underlying PoS blockchain of our protocol, and show how to cast the protocol $\Pi_{\mathsf{ss\text{-}tx}}$ on chain in Section 5.3. In Section 5.4, we pack all components together and present the full protocol.

5.1 Private Accounts and Transaction Fees

We provide an account-based private transaction fee mechanism. As a preparatory step, we assume that every block $\mathcal{B}$ in the blockchain $\mathcal{C}$ corresponds to a Merkle tree of private account commitments $\mathsf{com}_1, \ldots, \mathsf{com}_n$ (which we detail later) with Merkle root rt^{MT}. I.e., leaves of rt^{MT} are $\mathsf{com}_1, \ldots, \mathsf{com}_n$ that have so far appeared on the blockchain with order[10]. We denote $\langle g \rangle$ as a cyclic group of prime order q, and let h_1, h_2, h_3 be three different random group elements.

Account initialization. Upon receiving a funding transaction command $(\textsc{submit}, \mathsf{sid}, (0^\kappa, \mathsf{B}, \mathsf{aux}), \mathsf{acc})$, party P does the following. If aux contains an account set-up tag and acc has not been seen by P before, it initializes a new account by sampling randomly $\mathsf{sk}, r \in \mathbb{Z}_q$ and creating a commitment $\mathsf{com} = g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}}$. Party P issues a transaction $\mathtt{tx} = (0^\kappa, \mathsf{B}, \mathsf{aux})$ attached with com and a NIZK proof that she knows sk and r used in the commitment. The NP relation $\mathsf{R}_{\mathsf{acc_reg}}$ used in the transaction is defined as in Equation (1).

$$\mathsf{R}_{\mathsf{acc_reg}} = \left\{ \big(\mathsf{B}, \mathsf{com}\big), \big(\mathsf{sk}, r\big) \;\middle|\; \mathsf{com} = g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}} \right\}. \tag{1}$$

P also bookkeeps acc and maps it to a list of accounts initialized with $(\mathsf{com}, \mathsf{B}, \mathsf{sk}, r)$.

Account update. Upon receiving a funding transaction command $(\textsc{submit}, \mathsf{sid}, (0^\kappa, v, \mathsf{aux}), \mathsf{acc})$ such that a party P has already bookkeeped acc (and aux instructs to update acc), P updates the account as follows.

Let $(\mathsf{com}, \mathsf{B}, \mathsf{sk}, r)$ be the latest account state associated with acc, the goal is to update it to $(\mathsf{com}', \mathsf{B} + v, \mathsf{sk}', r')$. First, P extracts the latest rt^{MT} of her local blockchain $\mathcal{C}$, where com appears as a leaf of rt^{MT}. Then, she randomly samples $\mathsf{sk}', r' \in \mathbb{Z}_q$ and creates a new commitment $\mathsf{com}' = g^{r'} \cdot h_1^{\mathsf{B}+v} \cdot h_2^{\mathsf{sk}'}$; in addition, P also reveals a tag $\mathsf{tag} = h_3^{\mathsf{sk}}$ which nullifies the previous account state (to prevent double-spending). Finally, she creates a NIZK proof π that asserts (i) the well-formedness of the previous account state com and the new state com' with balance difference v; (ii) com appears as a leaf in rt^{MT}; (iii) tag is well-formed with respect to the previous account

[10]The blockchain protocol that we will use does not directly include transactions. Yet, we can still map each block in $\mathcal{C}$ to a Merkle root by considering our renewed settlement rule. Refer to Section 5.3 for more details.

key sk; and (iv) the new account balance $\mathsf{B}+v$ is in the valid range $[0, \mathsf{B}_{\mathsf{max}}]$. The formal definition of such NP relation $\mathsf{R}_{\mathsf{acc_upd}}$ is given in Equation (2).

$$\mathsf{R}_{\mathsf{acc_upd}} = \left\{ \begin{pmatrix} rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}' \\ (\mathsf{com}, \mathsf{B}, \mathsf{sk}, r, \mathsf{sk}', r') \end{pmatrix} \middle| \begin{array}{c} \mathsf{com} = g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}} \wedge \mathsf{com} \text{ appears as a leaf of } rt^{\mathsf{MT}} \\ \wedge \mathsf{tag} = h_3^{\mathsf{sk}} \wedge \mathsf{com}' = g^{r'} \cdot h_1^{\mathsf{B}+v} \cdot h_2^{\mathsf{sk}'} \wedge \mathsf{B}+v \in [0, \mathsf{B}_{\mathsf{max}}] \end{array} \right\}. \quad (2)$$

Party P then sends $(0^\kappa, v, \mathsf{aux})$ alongside $\mathsf{com}', \mathsf{tag}$ and proof π to the network; she also locally appends the new account state $(\mathsf{com}', \mathsf{B}+v, \mathsf{sk}', r')$ to the list associated with acc. A code specification of the account update procedure is presented in Algorithm 4.

Algorithm PrivAccountUpdate(v, acc)

1: Let $\mathcal{B} \leftarrow \mathrm{head}(\mathcal{C})$ and set rt^{MT} as the Merkle tree root in $\mathcal{B}$
2: Parse the latest account state w.r.t. acc as $(\mathsf{com}, \mathsf{B}, \mathsf{sk}, r)$
3: Compute $\mathsf{tag} \leftarrow h_3^{\mathsf{sk}}$.
4: Select $r', \mathsf{sk}' \xleftarrow{\$} \mathbb{Z}_q$, and compute $\mathsf{com}' \leftarrow g^{r'} \cdot h_1^{\mathsf{B}+v}, h_2^{\mathsf{sk}'}$.
▷ Prove relation $\mathsf{R}_{\mathsf{acc_upd}}$ in Equation (2)
5: Compute $\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{Prove}((rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}'), (\mathsf{com}, \mathsf{B}, \mathsf{sk}, r, \mathsf{sk}', r'))$
6: **return** $(\mathsf{com}', r', sk', \mathsf{tag}, \pi)$

Algorithm 4: *Transferring v coins into private account* acc.

Paying transaction fees. To spend a private account and pay transaction fees (i.e., upon receiving $(\textsc{submit}, \mathsf{sid}, \mathtt{tx}, \mathsf{acc})$ where $\mathtt{tx}$ is not a funding transaction), the party P first computes the fee of this transaction $v = \mathsf{fee}(\mathtt{tx})$. She then issues an update on acc analogously to adding balances, except now the account balance is deducted by v and instead of proving the updated balance is well-formed she proves that the old balance B is sufficient to pay the fees. The NP relation $\mathsf{R}_{\mathsf{pay_fee}}$ is given in Equation (3).

$$\mathsf{R}_{\mathsf{pay_fee}} = \left\{ \begin{pmatrix} rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}' \\ (\mathsf{com}, \mathsf{B}, \mathsf{sk}, r, \mathsf{sk}', r') \end{pmatrix} \middle| \begin{array}{c} \mathsf{com} = g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}} \wedge \mathsf{com} \text{ appears as a leaf of } rt^{\mathsf{MT}} \\ \wedge \mathsf{tag} = h_3^{\mathsf{sk}} \wedge \mathsf{com}' = g^{r'} \cdot h_1^{\mathsf{B}-v} \cdot h_2^{\mathsf{sk}'} \wedge v \in [0, \mathsf{B}] \end{array} \right\}. \quad (3)$$

Algorithm TxFeePayment$(\mathtt{tx}, \mathsf{acc})$

1: Let $\mathcal{B} \leftarrow \mathrm{head}(\mathcal{C})$ and set rt^{MT} as the Merkle tree root in $\mathcal{B}$
2: Parse the latest account state w.r.t. acc as $(\mathsf{com}, \mathsf{B}, \mathsf{sk}, r)$
3: Compute $\mathsf{tag} \leftarrow h_3^{\mathsf{sk}}$ and $v \leftarrow \mathsf{fee}(\mathtt{tx})$.
4: Select $r', \mathsf{sk}' \xleftarrow{\$} \mathbb{Z}_q$, and compute $\mathsf{com}' \leftarrow g^{r'} \cdot h_1^{\mathsf{B}-v}, h_2^{\mathsf{sk}'}$.
▷ Prove relation $\mathsf{R}_{\mathsf{pay_fee}}$ in Equation (3)
5: Compute $\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{Prove}((rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}'), (\mathsf{com}, \mathsf{B}, \mathsf{sk}, r, \mathsf{sk}', r'))$
6: **return** $(\mathsf{com}', r', sk', \mathsf{tag}, \pi)$

Algorithm 5: *Paying for transaction* $\mathtt{tx}$ *from private account* acc.

5.2 Secret-Sharing Transactions to an Ephemeral Committee

We present a protocol $\Pi_{\mathsf{ss\text{-}tx}}$ that achieves input causality for transactions sent during a time interval of predetermined length. At a high level, our protocol is based on the idea that transaction issuers publicly and verifiably share their transactions to a self-elected ephemeral committee (with size independent of the number of participants) and, at the end of the interval, ephemeral secret keys of the committee members are published so that all protocol participants can locally reconstruct all transactions.

Recall that we assume an adaptive adversary that can corrupt parties during the protocol execution, we borrow ideas from YOSO to elect an ephemeral committee with members that only needs to send one message and have destroyed their states beforehand. Yet, the subtle differences between our approach and fully-fledged YOSO is that, in our protocol, secret states (i.e., secret keys that can decrypt transaction shares) only need to be kept secret for a pre-determined amount of time and eventually they will all become public. Looking ahead, this enables our construction to achieve optimal corruption resilience against adaptive corruption and avoid using heavy cryptographic primitives.

Our protocol $\Pi_{\mathsf{ss\text{-}tx}}$ is based on the following building blocks (as specified in Section 3.3): (i) a statistically-hiding commitment scheme commit; (ii) an adaptively anonymous public-key encryption scheme $\mathsf{anonPKE}$; and (iii) an adaptive NIZK proof system Π_{NIZK}.

For the ease of presentation, we assume in this section that this protocol is executed by a fixed number of N parties that are always online, and there is a PKI setup where all parties generate their long-term key pair $(\mathsf{pk}, \mathsf{sk})$ using $\mathsf{anonPKE.KeyGen}$ and reach agreement over the set of public keys. The protocol is parametrized by the size of the secret-holding committee $n = \omega(\log \kappa)$ where κ is the security parameter (note that $N \gg n$).

Stage 1: Ephemeral key generation and dispersion. At the onset of the protocol execution, all self-elected parties execute the following steps. (We defer presenting how these parties are self-elected to Section 5.3, but emphasize that honest majority is preserved before their sending first message). Each P_i (we add subscript to distinguish them) first generates an ephemeral key pair $(\mathsf{epk}_i, \mathsf{esk}_i)$ of PKE. Next, she secret shares esk_i in a t-out-of-n manner where t is set slightly larger than one half (precisely, $t = n/2 + 1$). More specifically, P_i picks a random polynomial f of degree t such that $f(0) = \mathsf{esk}_i$, and shes sets each share $\mathsf{esk}_{i,j}$ as point $(j, f(j))$.

Then, she randomly chooses a *secret-holding* committee — a subset of n parties[11] among all participants with public keys $\{\mathsf{pk}_j\}_{j\in[n]}$. For each index j, P_i picks a randomness $r_{i,j} \in \{0,1\}^\kappa$ and (i) commits to share $\mathsf{esk}_{i,j}$ with randomness $r_{i,j}$ and get $c_{i,j}$; and (ii) encrypts share $\mathsf{esk}_{i,j}$ and commitment randomness $r_{i,j}$ under pk_j and gets $e_{i,j}$. Note that since long-term public keys are generated using $\mathsf{anonPKE}$, each $e_{i,j}$ does not leak which pk it is encrypted with (thus the adversary does not gain noticeable advantage compared with random corruption).

After generating all shares and commitments, a NIZK proof π_i is generated to prove that P_i has performed the above operations correctly[12]. The statement of π_i is of the form $\mathsf{stmt} = (\mathsf{epk}_i, \{c_{i,j}\}_{j\in[n]})$ and witness $\mathsf{w} = (\mathsf{esk}, f, \{r_{i,j}\}_{j\in[n]})$. The NP relation is defined as follows (recall $\mathsf{SS}_t(s, f)$ denotes shares from secret sharing with a given polynomial f of degree t):

[11] We stress that the dispersion to n parties can be chosen locally at random, since our protocol is secure so long as the majority of roles in Stage 1 is honest and they hand their shares to sufficiently many other honest parties.

[12] We note that, since the $\mathsf{anonPKE}$ scheme that we use involves querying the random oracle [CKR+23], the NIZK proof here only proves the correctness of commitment rather than the correctness of encryption and their consistency in between.

$$\mathsf{R}_{\mathsf{Key_Gen}} = \left\{ \begin{array}{c} \big(\mathsf{epk}_i, \{c_{i,j}\}_{j\in[n]}\big), \\ \big(\mathsf{esk}_i, f, \{r_{i,j}\}_{j\in[n]}\big) \end{array} \middle| \begin{array}{c} \mathsf{esk}_i \text{ is a valid secret key corresponding to } \mathsf{epk}_i \\ \wedge \{\mathsf{esk}_{i,j}\}_{j\in[n]} = \mathsf{SS}_t(\mathsf{esk}_i, f) \\ \wedge \big(\forall j \in [n], c_{i,j} = \mathsf{commit}(\mathsf{esk}_{i,j}, r_{i,j})\big) \end{array} \right\}. \quad (4)$$

P_i now erases the ephemeral secret key esk_i and all its shares (and also all randomness used), and publishes $(\mathsf{epk}_i, \{c_{i,j}\}_{j\in[n]}, \pi_i, \{e_{i,j}\}_{j\in[n]})$.

Note that, at the time that P_i is to send her first message, she has destroyed all secrets generated, and handed them to an anonymous committee. So long as no honest parties take further actions and less than t of the committee members are corrupted, the adversary learns nothing about esk_i from her message.

Protocol $\Pi_{\mathsf{ss\text{-}tx}}$ (Stage 1)

Let $n = \omega(\log \kappa)$ denote the size of committee. An adaptive anonymous PKI $\{\mathsf{pk}_i\}_{i\in[N]}$ and sk_i private to each party P_i. Each self-elected P_i does the following:

1: Run $(\mathsf{epk}_i, \mathsf{esk}_i) \leftarrow \mathsf{PKE.KeyGen}(1^\kappa)$
2: Select a random polynomial f and compute $\{\mathsf{esk}_{i,j}\}_{j\in[n]} \leftarrow \mathsf{SS}_t(\mathsf{esk}_i; f)$.
3: **for** $j \in [n]$ **do**
4: Select $r_{i,j} \xleftarrow{\$} \{0,1\}^\kappa$ and $\mathsf{pk}_j \xleftarrow{\$} \{\mathsf{pk}_i\}_{i\in N}$
5: Compute $c_{i,j} \leftarrow \mathsf{commit}(\mathsf{esk}_{i,j}, r_{i,j})$ and $e_{i,j} \leftarrow \mathsf{anonPKE.Enc}(\mathsf{pk}_j, \mathsf{esk}_{i,j} \,\|\, r_{i,j})$.
6: **end for**
7: Compute $\pi_i \leftarrow \mathsf{NIZK.Prove}((\mathsf{epk}_i, \{c_{i,j}\}_{j\in[n]}), (\mathsf{esk}_i, f, \{r_{i,j}\}_{j\in[n]}))$.
8: Erase esk_i, f and $\mathsf{esk}_{i,j}$ and all randomness used
9: Broadcast $(\mathsf{epk}_i, \{c_{i,j}\}_{j\in[n]}, \pi_i, \{e_{i,j}\}_{j\in[n]})$

Protocol 1: *The key generation sub-protocol.*

Stage 2: Transaction issuing. After all protocol participants share a consistent view of the (valid) key-generation messages sent in Stage 1 and hence an ephemeral PKI, they can start to issue transactions. An ephemeral key is valid if its associated NIZK proof verifies (parties agree on the order of these keys and we use subscript to distinguish different keys).

In order for a client to send a transaction $\mathtt{tx}$, she first collects all valid ephemeral public keys $\mathcal{K} = \{\mathsf{epk}_1, \ldots, \mathsf{epk}_k\}$. Then, the client encrypts the transaction symmetrically by sampling a random secret $s \leftarrow \{0,1\}^\kappa$ and computing e by XOR $\mathtt{tx}$ with the *hashed* secret, i.e., $e \leftarrow \mathtt{tx} \oplus H(s)$. She then secret shares s with a randomly sampled polynomial f_s of degree $t = k/2 + 1$ and gets shares $\{s_i\}_{i\in[k]}$. For each index $j \in [k]$, the client computes the encryption of s_i using randomness r_i with the corresponding ephemeral public key epk_i.

The client then proves that she has performed these computations (except the encryption of $\mathtt{tx}$) correctly, by generating a NIZK proof π. The statement of π is of the form $\mathsf{stmt} = (\{e_j\}_{j\in[k]})$ and witness $\mathsf{w} = (s, f_s, \{r_i\}_{i\in[k]})$. The NP relation is defined as follows:

$$\mathsf{R}_{\mathsf{ss_tx}} = \left\{ \begin{array}{c} \big(\{e_i\}_{i\in[k]}\big), \\ \big(s, f_s, r_{i\in[k]}\big) \end{array} \middle| \begin{array}{c} \{s_i\}_{i\in[k]} = \mathsf{SS}_t(s, f_s) \\ \wedge \big(\forall i \in [k], e_i = \mathsf{PKE.Enc}(\mathsf{epk}_i, s_i; r_i)\big) \end{array} \right\}. \quad (5)$$

Finally, the client releases the encrypted transaction, encrypted shares with the NIZK proof to the network.

Protocol $\Pi_{\mathsf{ss\text{-}tx}}$ (Stage 2)

Let $\mathcal{K} \leftarrow \{\mathsf{epk}_i \mid \Pi_{\mathsf{NIZK}}.\mathsf{Verify}((\mathsf{epk}_i, \{c_{i,j}\}_{j\in[n]})) = \mathsf{true}\}$ and $k = |\mathcal{K}|$. Each transaction issuer of $\mathtt{tx}$ does the following:

Let $t \leftarrow k/2 + 1$ and select $s \xleftarrow{\$} \{0,1\}^\kappa$.
Compute $e \leftarrow \mathtt{tx} \oplus H(s)$ and $\{s_i\}_{i\in[k]} \leftarrow \mathsf{SS}_t(s, f_s)$.
for $i \in [k]$ **do**
    Select $r_i \xleftarrow{\$} \{0,1\}^\kappa$ and compute $e_i \leftarrow \mathsf{PKE.Enc}(\mathsf{epk}_i, s_i; r_i)$.
end for
Compute $\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{Prove}((\{e_i\}_{i\in[k]}), (s, f_s, \{r_i\}_{i\in[k]}))$.
Broadcast $(\pi, e, \{e_i\}_{i\in[k]})$

Protocol 2: *The transaction-sending sub-protocol.*

Stage 3: Key reconstruction. Upon time reaches the end of the predetermined interval, parties can learn the transactions that have been sent so far by reconstructing sufficiently many ephemeral secret keys. Specifically, all protocol participants try to decrypt all encrypted ephemeral secret key shares, for all $\mathsf{epk}_i \in \mathcal{K}$. If a party P with long-term secret key sk successfully decrypts a share $\mathsf{esk}_{i,j} \,\|\, r_{i,j} \leftarrow \mathsf{anonPKE.Dec}(\mathsf{sk}, e_{i,j})$ and meanwhile they reveal the commitment $c_{i,j}$, she posts $\mathsf{esk}_{i,j}$ and related randomness to the network. After collecting sufficiently many valid shares, parties reconstruct esk_i via interpolating those sharing; and after gathering sufficiently many ephemeral secret keys, parties learn a transaction $\mathtt{tx}$ by locally decrypting the corresponding shares of the secret s used to encrypt the transaction, and then reconstructing s via interpolating and finally decrypt $\mathtt{tx}$ using $H(s)$.

Protocol $\Pi_{\mathsf{ss\text{-}tx}}$ (Stage 3)

Each party P with secret key sk does the following:

for $\mathsf{epk}_i \in \mathcal{K}$ **do**
    Parse the key generation message as $(\mathsf{epk}_i\{c_{i,j}\}_{j\in[n]}, \cdot, \{e_{i,j}\}_{j\in[n]})$
    for $j \in [n]$ **do**
        $\mathsf{esk}_{i,j} \,\|\, r_{i,j} \leftarrow \mathsf{anonPKE.Dec}(\mathsf{sk}, e_{i,j})$
        if $\mathsf{esk}_{i,j} \neq \perp$ **and** $c_{i,j} = \mathsf{commit}(\mathsf{esk}_{i,j}, r_{i,j})$ **then** Broadcast $(\mathsf{epk}_i, j, \mathsf{esk}_{i,j}, r_{i,j})$
    end for
end for

To decrypt a transaction $(\pi, e, \{e_i\}_{i\in[k]})$, each protocol participant does the following:

$S \leftarrow \emptyset$
for $i \in [k]$ **do**
    Let $(J_i, K_i) \leftarrow \{(j, \mathsf{esk}_{i,j}) \mid \exists(\mathsf{pk}, \mathsf{esk}_{i,j}, r_{i,j})$ s.t. $c_{i,j} = \mathsf{commit}(\mathsf{esk}_{i,j}, r_{i,j}) \wedge e_{i,j} = \mathsf{anonPKE.Enc}(\mathsf{pk}, \mathsf{esk}_{i,j}, r_{i,j})\}$
    if $|J_i| > n/2 + 1$ **then** $\mathsf{esk}_i \leftarrow \mathsf{interpolate}(J_i, K_i)$ and set $S \leftarrow S \cup \{i\}$
end for
if $|S| \geq k/2 + 1$ **then**
    for $i \in S$ **do**
        Set $s_i \leftarrow \mathsf{PKE.Dec}(\mathsf{esk}_i, e_i)$
    end for
    Compute $s \leftarrow \mathsf{interpolate}(S, \{s_i\})$ and $\mathtt{tx} \leftarrow e \oplus H(s)$

34: **return** $\mathtt{tx}$
35: **end if**

Protocol 3: *The transaction reconstruction sub-protocol.*

5.3 PoS-based Role Assignment and Transaction Serialization

Role assignment in Stage 1 of $\Pi_{\mathsf{ss\text{-}tx}}$. We elaborate on the self-election procedure in the first stage of $\Pi_{\mathsf{ss\text{-}tx}}$— i.e., how parties can self-elect leading to a committee with honest majority, based on a Nakamoto-style PoS blockchain Ouroboros-Genesis.

Recall that in Ouroboros-Genesis, a party P self-elects as the slot-leader for slot $\mathtt{sl}$ in epoch $\mathtt{ep}$, if her VRF evaluation response (y, π) on point $(\eta \parallel \mathtt{sl} \parallel \mathtt{TEST})$ satisfies $y < T_{\mathsf{P}}^{\mathsf{ep}} = 2^{\ell_{\mathsf{VRF}}} \cdot \phi_f(\alpha_{\mathsf{P}}^{\mathsf{ep}})$ where η is the randomness used in epoch $\mathtt{ep}$, $\alpha_{\mathsf{P}}^{\mathsf{ep}} \in [0,1]$ is the relative stake controlled by P (i.e., the fraction of her stake out of all stake), ℓ_{VRF} is the VRF output length and $\phi_f(\alpha) \triangleq (1-f)^{\alpha}$. The function ϕ guarantees that splitting stakes among "virtual" parties does not give the stakeholder any advantage on being elected as the leader for a particular slot. The stake distribution $\mathbb{S}$ that parties use to compute $\alpha_{\mathsf{P}}^{\mathsf{ep}}$ in epoch $\mathtt{ep}$ is the distribution at the end of epoch $\mathtt{ep}-2$ (or, distribution in the genesis block for the first two epochs).

We adopt an analogous procedure for leadership election to generate ephemeral keys in Stage 1 of $\Pi_{\mathsf{ss\text{-}tx}}$, by letting parties self-elect if their VRF evaluation output (y, π) on point $(\eta \| \mathtt{sl} \| \mathtt{COMMITTEE})$ yields $y < T_{\mathsf{P}}^{\mathsf{ep,cmte}} = 2^{\ell_{\mathsf{VRF}}} \cdot \alpha_{\mathsf{P}}^{\mathsf{ep}}$ (which is simply a linear scaling). The stake distribution $\mathbb{S}$ that we use to elect ephemeral keys is the same as that used for slot-leadership to generate the next block. If a party P successfully self-elects as the committee member, she diffuses $(key, \mathtt{sl}, \mathsf{P}, y, \pi, \sigma)$ where key is the ephemeral public key with associated secret-key shares and proof as generated in Protocol 1, (y, π) is the VRF evaluation output and proof, and σ is the signature on the entire message. Note that, in order to share the ephemeral secret key to a set of share-holding parties, P select $n = \omega(\log \kappa)$ keys over the same stake distribution (by considering stakes with weights and apply the selection algorithm per unit of stake).

To gather a set of ephemeral keys with honest majority, we let this procedure last for $R/6$ slots where R is the duration of an epoch. Looking ahead, we show that with overwhelming probability, during $R/6$ slots the ephemeral keys that were honestly created account for roughly the same fraction of honest stakes. And since honest majority holds among the whole stakes at any time during the execution, this translates to the honest majority with respect to the ephemeral keys.

Remark 3. Astute readers may notice that Ouroboros-Genesis operates in an environment with dynamic participation (i.e., the activated fraction of stakes is unknown and can fluctuate over time) which does not cope well with choosing the share-holding committee size n. If n is set too small and the adversary later raises participation level, $\mathcal{A}$ may be able to reconstruct the keys ahead of time; and if n is set too large then the adversary can reduce the participation and honest parties would not be able to eventually reconstruct. We stress that such concern is not particular to our construction, but rather a generic problem when running YOSO protocols on top of a permissionless PoS blockchain where parties may temporarily lose access to her resources. To mitigate, in this work we assume that the ratio of activated stakes is fixed. Note that, even with a restricted adversary, the genesis chain selection rule still offers bootstrapping without checkpoints (as opposed to require parties stay online periodically).

Pipelining $\Pi_{\text{ss-tx}}$. Notice that naïvely running Protocol $\Pi_{\text{ss-tx}}$ sequentially on top of the PoS blockchain leads to intervals where transaction issuers waits for the next committee to publish new ephemeral keys, we thus re-parameterize the duration of each stage and pipeline the protocol so that a set of ephemeral public keys are always available to use, except for the first "warm-up" epoch at the onset (parties can still send unencrypted transactions, e.g., the funding transactions to set up their private account, in the first epoch).

Suppose an invocation of $\Pi_{\text{ss-tx}}$ begins at slot $\mathtt{sl}$, the time interval that parties self-elect as committee members and generate and emit ephemeral keys lasts for the first $R/6$ slots (recall that R denotes the duration of an epoch). The ephemeral public keys (and their associated shares and proofs) are then recorded on the blockchain. Note that, in order for transaction issuers to adopt them, ephemeral public keys should be recorded in a block with timestamp no later than $\mathtt{sl}+2R/3$. Transactions sent during interval $[\mathtt{sl}+R : \mathtt{sl}+7R/6)$ — i.e., an interval of $R/6$ rounds starting at exactly R slots after $\mathtt{sl}$— are encrypted under all ephemeral public keys generated in this invocation and recorded on-chain. As our analysis shall show, at slot $\mathtt{sl}+R$ and afterwards, the underlying PoS blockchain guarantees that parties share a consistent view of blocks with timestamp before $\mathtt{sl}+2R/3$; in other words, all transaction sent during $[\mathtt{sl}+R : \mathtt{sl}+7R/6)$ will use the same set of ephemeral public keys.[13]

Protocol participants wait for the blockchain to order encrypted transactions (which we detail soon) and upon protocol time reaches slot $\mathtt{sl}+2R$, each party will try to decrypt ephemeral secret-key shares, and if a party successfully decrypts the share(s) encrypted under her long-term public key, she emits the share decryption to the network. Hence, all participants can reconstruct sufficiently many ephemeral secret keys and decrypt all transactions sent during time $t \in [\mathtt{sl}+R : \mathtt{sl}+7R/6)$; the blockchain also records all decrypted shares so that parties that later join the protocol can also decrypt and learn these transactions.

See Figure 1 for an illustration of the pipelined invocations. Note that the i-th invocation of $\Pi_{\text{ss-tx}}$ (denoted by Π_i) happens at slot $(i-1)\cdot R/6+1$.

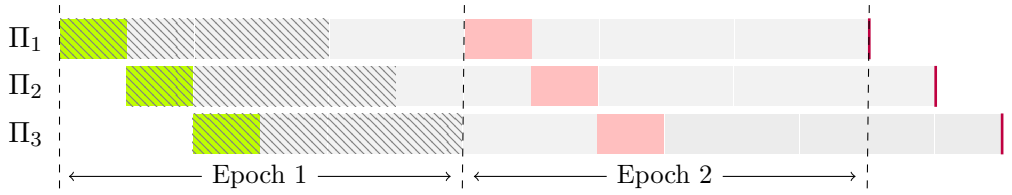

Figure 1: *An illustration of the pipelined invocation of Protocol* $\Pi_{\text{ss-tx}}$. *The lime interval represents the slots where parties self-elect as committee members (Stage 1 of* $\Pi_{\text{ss-tx}}$*); all ephemeral public keys should be recorded in intervals marked with slant lines; the pink interval shows the slots where parties secret-share their transactions towards the committee that is elected in the same invocation of* $\Pi_{\text{ss-tx}}$ *(Stage 2 of* $\Pi_{\text{ss-tx}}$*); and the purple line indicates the time when parties start to decrypt the ephemeral secret key shares.*

Transaction serialization. We now detail the transaction serialization procedure, by revising the Ouroboros-Genesis protocol as follows. Note that, as a PoW-variant of this transaction serialization mechanism has been proposed and analyzed in [KLS24], we here focus on how to migrate the approach to a PoS setting.

[13] Our parametrization takes the advantage that parties start issuing transactions after the blockchain reaches agreement over the set of ephemeral keys, which significantly simplifies the analysis and indicates that clients only need to listen to the diffusion network for blockchains $\mathcal{F}_{\text{Diffuse}}^{\text{bc}}$ to issue transactions.

The blockchain in our protocol does not include transactions "directly." Instead, it includes only valid ephemeral keys and *profiles* which, roughly speaking, is a kind of message that reports its sender's local transaction order. In order for parties to emit their profiles, in every slot they perform an additional VRF evaluation on $(\mathsf{EvalProve}, \eta \,\|\, \mathtt{sl} \,\|\, \mathtt{PROFILE})$; and if the output is smaller than a threshold value $T^{\mathsf{ep,cmte}}$ (i.e., same as that used to elect ephemeral keys) then they emit a profile

$$\mathsf{PROFILE} \triangleq \big((\mathtt{tx}_1, \ldots, \mathtt{tx}_n), \mathtt{sl}, \mathsf{P}, y, \pi, \sigma\big),$$

where $(\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$ is the local transaction order of P received after time slot $\mathtt{sl} - R/6$ (note that we assume the transaction diffuse network $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ is Δ-bounded delay, hence parties do not need to report their whole profile), (y, π) are the VRF output and σ is the signature on the whole message. Due to input causality, some transactions might be in their encrypted form; in addition, since all encrypted transactions have to pay fees as specified in Section 5.1 where an account tag is revealed each time, parties include in profile only the first transaction that they received for those sharing the same tag.

After parties produce and emit profiles and they are included in the blockchain, a transaction dependency graph can be built from the settled chain by adding each transaction as a vertex, and add an edge $(\mathtt{tx}, \mathtt{tx}')$ if majority of the profiles report $\mathtt{tx}$ before $\mathtt{tx}'$. Due to consistency, honest parties share the same view of the transaction dependency graph G extracted from the settled part of the blockchain and they apply a serialization function F on G which yields the final transaction order.

5.4 The Full Protocol Description

The main protocol instance. We introduce our main protocol instance $\Pi^{\mathsf{fair++}}_{\mathsf{ledger}}$ dispatching to the relevant subprocesses. This protocol extends the Ouroboros-Genesis [BGK+18] protocol (which shares the same set of parameters and internal variables for the PoS blockchain), by additionally introducing the transaction fee mechanism and serialization with order fairness. It is thus additionally parametrized with the share-holding committee size n (cf. Table 1). A detailed explanation on internal variables of protocol participants is presented in Table 2.

Protocol $\Pi^{\mathsf{fair++}}_{\mathsf{ledger}}(\mathsf{P}, \mathrm{sid}; \mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}, \mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{RO}}, \mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}, \mathcal{F}_{\mathsf{Diffuse}})$

Global Variables:
- Parameters: R, k, f, s, n, F.
- Local states: τ, $\mathtt{ep}$, $\mathtt{sl}$, $\mathcal{C}_{\mathsf{loc}}$, $\mathtt{isInit}$, t_{on}, t_{work}, $\mathtt{buffer}_{\mathsf{tx}}$, $\mathtt{buffer}_{\mathsf{pb}}$, $\mathtt{buffer}_{\mathsf{cmte}}$, $T^{\mathsf{ep}}_{\mathsf{P}}$, $T^{\mathsf{ep,cmte}}$, $v^{\mathrm{vrf}}_{\mathsf{P}}$, $v^{\mathrm{kes}}_{\mathsf{P}}$, $\mathtt{accounts}$.

Registration/Deregistration:
- Upon receiving $(\textsc{register}, \mathcal{R})$, execute $\mathsf{Registration}(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving $(\textsc{de-register}, \mathcal{R})$, execute $\mathsf{Deregistration}(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving input $(\textsc{is-registered}, \mathrm{sid})$ return $(\textsc{register}, \mathrm{sid}, 1)$ if the local registry $\mathtt{Reg}$ indicates that this party has successfully completed a registration with $\mathcal{R} = \mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ (and did not de-register since then). Otherwise, return $(\textsc{register}, \mathrm{sid}, 0)$.

Interacting with the Ledger: Upon receiving a ledger-specific input $I \in \{(\textsc{submit}, \ldots), (\textsc{read}, \ldots), (\textsc{maintain-ledger}, \ldots)\}$ verify first that all resources are available. If not all resources are available, then ignore the input; else (i.e., the party is operational and time-aware) execute one of the following steps depending on the input I:

- **If** $I = (\text{SUBMIT}, \text{sid}, \mathtt{tx}, \mathsf{acc})$ **then** set $\mathtt{buffer}_{\mathsf{tx}} \leftarrow \mathtt{buffer}_{\mathsf{tx}} \,\|\, (\mathtt{tx}, \mathsf{acc})$ and invoke $\mathsf{IssueNewTransaction}(\mathsf{P}, \text{sid}, \mathtt{tx}, \mathsf{acc})$.
- **If** $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ **then** invoke $\mathsf{LedgerMaintenance}(\mathsf{P})$; if $\mathsf{LedgerMaintenance}$ halts then halt the protocol execution (all future input is ignored).
- **If** $I = (\text{READ}, \text{sid})$ then invoke $\mathsf{ReadState}(\mathsf{P}, \text{sid})$.

Handling external calls:

- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ forward it to $\mathcal{G}_{\mathsf{Clock}}$ and output $\mathcal{G}_{\mathsf{Clock}}$'s response.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$, record that a clock-update was received in the current round. If this instance is currently time-aware but otherwise stalled or offline, then evolve the KES signing key by sending $(\mathsf{USign}, \text{sid}, \mathsf{P}, 0, \tau)$ to $\mathcal{F}_{\mathsf{KES}}$, where τ is the current time, and forward $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$. Furthermore, consider any active interruptible execution as completed.
- Upon receiving $(\text{EVAL}, \text{sid}_{RO}, x)$ forward the query to $\mathcal{G}_{\mathsf{RO}}$ and output $\mathcal{G}_{\mathsf{RO}}$'s response.
- Upon receiving $(\text{FETCH}, \text{sid}_D)$ forward it to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ and output $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$'s response.

Protocol 4: *The main protocol instance of* $\Pi^{\mathsf{fair++}}_{\mathsf{ledger}}$.

Registration and de-registration. In order to participate in the protocol, parties need to register with their resources. Protocol 5 captures the registration procedure. Note that before registering with the network and random oracle, parties should check if they have access to all global functionalities. Parties also register on $\mathcal{F}_{\mathsf{CRS}}$ and learn g, h_1, h_2, h_3 for the private accounts.

Protocol $\mathsf{Registration}(\mathsf{P}, \text{sid}, \mathsf{Reg}, \mathcal{G})$

if $\mathcal{G} \in \{\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}\}$ **then**
  Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}$, set registration status to registered with $\mathcal{G}$, and output the valued received by $\mathcal{G}$.
else if $\mathcal{G} = \mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ **then**
  if the party is not registered with $\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{rpoRO}}$ or $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$, or is already registered with all setup functionalities **then**
    ignore this input
  else
    for each $\mathcal{F} \in \{\mathcal{F}_{\mathsf{INIT}}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}}, \mathcal{F}_{\mathsf{NIZK}}\}$ **do**
      Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}$, set its registration status as registered with $\mathcal{F}$, but do not output the received values.
    end for
    Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$ and receive and store the current time τ.
    Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}$ and set $t_{\mathsf{on}} \leftarrow \tau$.
    Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}_{\mathsf{CRS}}$ and receive and store g, h_1, h_2, h_3.
    Output $(\text{REGISTER}, \text{sid}, \mathsf{P})$ once completing the registration with all the above resources $\mathcal{F}$.
  end if
end if

Protocol 5: *The registration procedure.*

The de-registration procedure (Protocol 6) is analogous to the above, where de-registering from

the ledger is simplified as dropping from the network functionality.

Protocol $\mathsf{Deregistration}(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{G})$

1: **if** $\mathcal{G} \in \{\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}\}$ **then**
2: Send $(\textsc{de-register}, \mathrm{sid})$ to $\mathcal{G}$, set registration status as de-registered with $\mathcal{G}$, and output the valued received by $\mathcal{G}$.
3: **end if**
4: **if** $\mathcal{G} = \mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ **then**
5: Send $(\textsc{de-register}, \mathrm{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}$, set its registration status as de-registered with $\mathcal{F}_{\mathsf{Diffuse}}$ and output $(\textsc{de-register}, \mathrm{sid}, \mathsf{P})$.
6: **end if**

Protocol 6: *The de-registration procedure.*

Party initialization. Upon initialization, parties query $\mathcal{F}_{\mathsf{INIT}}$ to claim their initial stakes (if they are invoked at the onset) or learn the stake distribution in the genesis block (if they are invoked later). We refer to [BGK+18] for more details.

Protocol $\mathsf{Initialization}(\mathsf{P}, \mathrm{sid})$

▷ The following steps are executed in an $(\textsc{maintain-ledger}, \mathrm{sid}, \mathrm{minerID})$-interruptible manner:
1: Send $(\mathsf{KeyGen}, \mathrm{sid}, \mathsf{P})$ to $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$; receiving $(\mathsf{VerificationKey}, \mathrm{sid}, v^{\mathrm{vrf}}_{\mathsf{P}})$ and $(\mathsf{VerificationKey}, \mathrm{sid}, v^{\mathrm{kes}}_{\mathsf{P}})$, respectively.
▷ The following branch is executed on the first maintenance query after registration of this instance at time 0.
2: **if** $\tau = 0$ **then**
3: Send $(\mathsf{ver_keys}, \mathrm{sid}, \mathsf{P}, v^{\mathrm{vrf}}_{\mathsf{P}}, v^{\mathrm{kes}}_{\mathsf{P}})$ to $\mathcal{F}_{\mathsf{INIT}}$ to claim stake from the genesis block.
4: Invoke $\mathsf{FinishRound}(\mathsf{P})$ ◁ Resume below on next maintain-ledger activation
5: Invoke $\mathsf{UpdateTime}(\mathsf{P})$ to update τ, $\mathtt{ep}$, and $\mathtt{sl}$.
6: **while** $\tau = 0$ **do**
7: Call $\mathsf{UpdateTime}(\mathsf{P})$ to update τ, $\mathtt{ep}$, and $\mathtt{sl}$ and give up the activation (set anchor here)
8: **end while**
9: **end if**
▷ The following executed if this is a non-genesis round.
10: **if** $\tau > 0$ **then**
11: **if** $\mathcal{F}_{\mathsf{INIT}}$ signals an error **then**
12: Halt the execution
13: **end if**
14: Send $(\mathsf{genblock_req}, \mathrm{sid}, \mathsf{P})$ to $\mathcal{F}_{\mathsf{INIT}}$
15: Receive from $\mathcal{F}_{\mathsf{INIT}}$ the response $(\mathsf{genblock}, \mathrm{sid}, \mathbf{G} = (\mathbb{S}_1, \eta_1))$, where

$$\mathbb{S}_1 = \big((\mathsf{P}_1, v^{\mathrm{vrf}}_1, v^{\mathrm{kes}}_1, s_1), \ldots, (\mathsf{P}_1, v^{\mathrm{vrf}}_n, v^{\mathrm{kes}}_n, s_n)\big).$$

16: Set $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathbf{G}$
17: Set $T^{\mathsf{ep}}_i \leftarrow 2^{\ell_{\mathsf{VRF}}} \phi_f(\alpha^{\mathsf{ep}}_i)$ as the threshold for stakeholder P_i for epoch $\mathtt{ep}$, where α^{ep}_i is the relative stake of stakeholder P_i in $\mathbb{S}_{\mathsf{ep}}$ and ℓ_{VRF} denotes the output length of $\mathcal{F}_{\mathsf{VRF}}$.
18: Send $(\textsc{hello}, \mathrm{sid}, \mathsf{P}, v^{\mathrm{vrf}}_{\mathsf{P}}, v^{\mathrm{kes}}_{\mathsf{P}})$ to $\mathcal{F}^{\mathsf{new}}_{\mathsf{Diffuse}}$
19: **end if**
20: Set $\mathtt{isInit} \leftarrow \mathsf{true}$, $t_{\mathsf{on}} \leftarrow \tau$, and $t_{\mathsf{work}} \leftarrow 0$.

GLOBAL VARIABLES: The protocol stores the list of variables $\mathsf{P}, v_{\mathsf{P}}^{\text{vrf}}, v_{\mathsf{P}}^{\text{kes}}, \tau$, $\mathtt{ep}$, $\mathtt{sl}$, $\mathcal{C}_{\text{loc}}$, $T_{\mathsf{P}}^{\mathsf{ep}}$, $\mathtt{isInit}$, t_{on} to make each of them accessible by all protocol parts.

Protocol 7: *The PoS initialization procedure.*

Fetch information. Parties fetch information from different diffusion functionalities. Specifically, they learn new chains from $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$, new transactions (via the FETCH command to fetch from $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$), new ephemeral committee keys and shares from $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$ and new profiles from $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$. Additionally, they listen to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{new}}$ to learn if a new party is about to join.

Note that, we denote by $\tilde{\mathtt{tx}}$ as a generalized notion for a transaction, i.e., either in the clear, or encrypted with well-formed proofs as specified in Sections 5.1 and 5.2. Analogously, let cmte denote a generalized notion for an ephemeral committee-related message (that is, a self-election message with well-formed keys and shares, or a message opening a share of previously generated committee ephemeral keys, as specified in Sections 5.2 and 5.3).

Protocol $\mathsf{FetchInformation}(\mathsf{P}, \text{sid})$

1: Define $\mathsf{fetchcount} \leftarrow \tau - t_{\mathsf{on}}$ ◁ Difference of actual time and last time online when fetched
▷ Fetch blocks.
2: Send $\mathsf{fetchcount}$ fetch queries $(\text{FETCH}, \text{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$; denote the i-th response from $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ by $(\text{FETCH}, \text{sid}, b_i)$.
3: Extract chains $\mathcal{C}_1, \ldots, \mathcal{C}_M$ from $b_1, \ldots, b_{\mathsf{fetchcount}}$.
▷ Fetch transactions.
4: Denote the response from $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ in this round by $(\text{FETCH}, \text{sid}, b)$.
5: Extract new transactions $\tilde{\mathtt{tx}}_1, \ldots, \tilde{\mathtt{tx}}_k$ from b.
▷ Fetch committee-related messages.
6: Send $\mathsf{fetchcount}$ fetch queries $(\text{FETCH}, \text{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$; denote the i-th response from $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$ by $(\text{FETCH}, \text{sid}, b_i)$.
7: Extract ephemeral committee related messages $\mathsf{cmte}_1, \ldots, \mathsf{cmte}_i$ from $b_1, \ldots, b_{\mathsf{fetchcount}}$.
▷ Fetch profiles.
8: Send $\mathsf{fetchcount}$ fetch queries $(\text{FETCH}, \text{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$; denote the i-th response from $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$ by $(\text{FETCH}, \text{sid}, b_i)$.
9: Extract profiles $\mathsf{pb}_1, \ldots, \mathsf{pb}_j$ from $b_1, \ldots, b_{\mathsf{fetchcount}}$.
▷ Fetch on $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{new}}$.
10: Send $\mathsf{fetchcount}$ fetch queries $(\text{FETCH}, \text{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{new}}$
11: **if** a message $(\text{HELLO}, \text{sid}, \cdot)$ was received **then**
12: Set WELCOME $\leftarrow 1$
13: **else**
14: Set WELCOME $\leftarrow 1$
15: **end if**

OUTPUT: The protocol outputs $\mathcal{C}_1, \ldots, \mathcal{C}_M$, $\tilde{\mathtt{tx}}_1, \ldots, \tilde{\mathtt{tx}}_k$, $\mathsf{cmte}_1, \ldots, \mathsf{cmte}_i$, $\mathsf{pb}_1, \ldots, \mathsf{pb}_j$, and WELCOME to its caller (but not to $\mathcal{Z}$) if $\mathsf{fetchcount} > 0$ and otherwise provide an empty output.

Protocol 8: *The sub-protocol that fetches incoming blocks, transactions, profiles, ephemeral keys from the network.*

Update time. Parties keep in sync with $\mathcal{G}_{\mathsf{Clock}}$ before they take ledger maintenance operations.

Protocol $\mathsf{UpdateTime}(\mathsf{P}, \mathrm{sid})$

1: Update τ to the current time stamp of the clock (by sending $(\text{CLOCK-READ}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$ and obtaining its answer).
2: Set $\mathtt{ep} \leftarrow \lceil \tau / R \rceil$ and $\mathtt{sl} \leftarrow \tau$

OUTPUT: The protocol outputs τ, $\mathtt{ep}$, $\mathtt{sl}$ to its caller (but not to $\mathcal{Z}$).

Protocol 9: *The time update sub-protocol.*

Chain validation. The validation procedure extends the one used by Ouroboros-Genesis and additionally checks the validity of profiles and committee messages included on-chain.

Protocol $\mathsf{IsValidChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C})$

1: **if** $\mathcal{C}$ contains empty epochs or starts with a block other than $\mathbf{G}$ **then return** false
2: **if** $\exists \mathcal{B} \in \mathcal{C} : \mathsf{TS}(\mathcal{B}) > \mathtt{sl}$ **then return** false
3: **for** each epoch $\mathtt{ep}'$ **do**
4: Let $\mathbb{S}^{\mathcal{C}}_{\mathtt{ep}'}$ be the stakeholder distribution at the end of epoch $\mathtt{ep}' - 2$ in $\mathcal{C}$
5: Set $\alpha^{\mathtt{ep},\mathcal{C}}_{\mathsf{P}'}$ to be the relative stake of any party P' in $\mathbb{S}^{\mathcal{C}}_{\mathtt{ep}'}$
6: Define $T^{\mathtt{ep}'}_{\mathsf{P}'} \leftarrow 2^{\ell_{\mathsf{VRF}}} \cdot \phi_f(\alpha^{\mathtt{ep}'}_{\mathsf{P}'})$ and $T^{\mathtt{ep}',\mathsf{cmte},\mathcal{C}} \leftarrow 2^{\ell_{\mathsf{VRF}}} \cdot \alpha^{\mathtt{ep}'}_{\mathsf{P}'}$
7: **if** $\mathtt{ep}' = 1$ **then**
8: Set $\eta^{\mathcal{C}}_1 \triangleq \eta_1$ from $\mathbf{G}$
9: **else**
10: Set v as the concatenation of the VRF outputs y_ρ from all blocks in $\mathcal{C}$ from the first two-thirds of slots of epoch $\mathtt{ep}' - 1$
11: Send $(\text{EVAL}, \mathrm{sid}_{RO}, \eta^{\mathcal{C}}_{\mathtt{ep}'-1} \,\|\, \mathtt{ep}' \,\|\, v)$ to $\mathcal{G}_{\mathsf{RO}}$ and set response as $\eta^{\mathcal{C}}_{\mathtt{ep}'}$.
12: **end if**
13: **for** each block $\mathcal{B}$ in $\mathcal{C}$ from epoch $\mathtt{ep}'$ **do**
14: Parse $\mathcal{B}$ as $(h, \mathtt{st}', \mathtt{sl}', crt, \rho, \sigma)$.
▷ Check hash
15: Set $\mathsf{badhash} \leftarrow (h \neq \mathcal{G}_{\mathsf{RO}}(\mathcal{B}^{-1}))$, where $\mathcal{B}^{-1}$ is the last block in $\mathcal{C}$ before $\mathcal{B}$
▷ Check VRF
16: Parse crt as (P', y, π) for some P'
17: Send $(\mathsf{Verify}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathtt{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{TEST}, y, \pi, v^{vrf}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$ and denote tis response by $(\mathsf{Verified}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathtt{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{TEST}, y, \pi, b_1)$
18: Send $(\mathsf{Verify}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathtt{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, v^{vrf}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$ and denote tis response by $(\mathsf{Verified}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathtt{ep}'} \,\|\, \mathtt{sl}' \,\|\, \mathtt{NONCE}, y_\rho, \pi_\rho, b_2)$
19: Set $\mathsf{badVRF} \leftarrow b_1 = 0 \vee b_2 = 0 \vee y \geq T^{\mathtt{ep}'}_{\mathsf{P}'}$
▷ Check signature
20: Send $(\mathsf{Verify}, \mathrm{sid}, (h, \mathtt{st}', \mathtt{sl}', crt, \rho), \mathtt{sl}', \sigma, v^{kes}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{KES}}$ and denote tis response by $(\mathsf{Verified}, \mathrm{sid}, (h, \mathtt{st}', \mathtt{sl}', crt, \rho), \mathtt{sl}', b_3)$
21: Set $\mathsf{badSig} \leftarrow b_3 = 0$
▷ Check profiles
22: **if** $\exists \mathsf{pb} \in \mathcal{B} : \mathsf{TS}(\mathsf{pb}) \notin [\mathsf{TS}(\mathcal{B}) - 2R/3, \mathsf{TS}(\mathcal{B}))$ **then** Set $\mathsf{badProfile} \leftarrow \mathsf{true}$
23: **for** each $\mathsf{pb} \in \mathcal{B}$ **do**
24: Parse pb as $(pb, sl, (y_{pb}, \pi_{pb}), \sigma)$
25: **if** pb contains invalid $\mathtt{tx}$ (in the clear) **then** $\mathsf{badProfile} \leftarrow \mathsf{true}$
26: **if** pb contains encrypted transactions with tag $tag \in \mathcal{C}$ **then** $\mathsf{badProfile} \leftarrow \mathsf{true}$

Send $(\mathsf{Verify}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathsf{ep}'} \parallel sl \parallel \texttt{PROFILE}, y_{pb}, \pi_{pb}, v^{vrf}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$ and denote tis response by $(\mathsf{Verified}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathsf{ep}'} \parallel sl \parallel \texttt{PORFILE}, y_{pb}, \pi_{pb}, b_4)$

Send $(\mathsf{Verify}, \mathrm{sid}, (pb, sl, (y_{pb}, \pi_{pb}), sl, \sigma, v^{kes}_{\mathsf{P}'}))$ to $\mathcal{F}_{\mathsf{KES}}$ and denote tis response by $(\mathsf{Verified}, \mathrm{sid}, (pb, sl, (y_{pb}, \pi_{pb}), sl, b_5))$

if $(b_4 = 0) \vee (b_5 = 0) \vee (y_{pb} \geq T^{\mathsf{ep}', \mathsf{cmte}, \mathcal{C}})$ **then** Set $\mathsf{badProfile} \leftarrow \mathsf{true}$

end for

▷ Check committee messages

for each $\mathsf{cmte} \in \mathcal{B}$ **do**

Parse cmte as $(cmte, sl, (y_{cmte}, \pi_{cmte}), \sigma)$

if NIZK in $cmte$ does not verify **then** $\mathsf{badCMTE} \leftarrow \mathsf{true}$

Send $(\mathsf{Verify}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathsf{ep}'} \parallel sl \parallel \texttt{COMMITEE}, y_{cmte}, \pi_{cmte}, v^{vrf}_{\mathsf{P}'})$ to $\mathcal{F}_{\mathsf{VRF}}$ and denote tis response by $(\mathsf{Verified}, \mathrm{sid}, \eta^{\mathcal{C}}_{\mathsf{ep}'} \parallel sl \parallel \texttt{COMMITTEE}, y_{cmte}, \pi_{cmte}, b_6)$

Send $(\mathsf{Verify}, \mathrm{sid}, (cmte, sl, (y_{cmte}, \pi_{cmte}), sl, \sigma, v^{kes}_{\mathsf{P}'}))$ to $\mathcal{F}_{\mathsf{KES}}$ and denote tis response by $(\mathsf{Verified}, \mathrm{sid}, (cmte, sl, (y_{cmte}, \pi_{cmte}), sl, b_7))$

if $(b_6 = 0) \vee (b_7 = 0) \vee (y_{pb} \geq T^{\mathsf{ep}', \mathsf{cmte}, \mathcal{C}})$ **then** Set $\mathsf{badProfile} \leftarrow \mathsf{true}$

end for

if $\mathsf{badHash} \vee \mathsf{badVRF} \vee \mathsf{badSig} \vee \mathsf{badProfile} \vee \mathsf{badCMTE}$ **then return** false

end for

end for

return true

Protocol 10: *The extended chain validation rule checking profiles and committees.*

Chain selection. Our chain selection rule follows that in Ouroboros-Genesis, where if two chains forks recently then the longest chain rule applies; otherwise when the fork happens for more than k blocks then the density rule applies (details refer to [BGK+18]).

Protocol $\mathsf{SelectChain}(\mathsf{P}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}}, \mathcal{N} = \{\mathcal{C}_1, \ldots, \mathcal{C}_M\})$

Set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_{\mathsf{loc}}$.

for i **from** 1 **to** M **do**

if $\mathsf{IsValidChain}(\mathcal{C}_i)$ **then**

if $\mathcal{C}_i$ forks from $\mathcal{C}_{\mathsf{max}}$ at most k block **then**

if $|\mathcal{C}_i| > |\mathcal{C}_{\mathsf{max}}|$ **then** set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$

else

Set $j \leftarrow \max\{j' \geq 0 \mid \mathcal{C}_{\mathsf{max}}$ and $\mathcal{C}_i$ have the same block in $\mathtt{sl}_{j'}\}$.

if $|\mathcal{C}_i[0 : j + s]| > |\mathcal{C}_{\mathsf{max}}[0 : j + s]|$ **then** set $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$

end if

end if

end for

Set $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{max}}$.

Protocol 11: *The chain selection rule.*

The transactions issuing procedure. Based on the type of input command, a party P issues different types of transactions. Specifically, with empty acc parameter, `tx` is sent in the clear. When `tx` is a funding transaction, based on whether P has stored acc in `accounts` or not, a corresponding

private account is set-up/updated. In case of sending a private transaction with acc paying the fees, P first pays the transaction fee and then secret shares $\mathtt{tx}$ to the corresponding committee.

Protocol $\mathsf{IssueNewTransaction}(\mathsf{P}, \mathrm{sid}, \mathtt{tx}, \mathsf{acc})$

if $\mathsf{acc} = \bot$ **then**
  Send $(\textsc{diffuse}, \mathrm{sid}, \mathtt{tx})$ to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$
else
  if $\mathtt{tx}$ is of form $(0^\kappa, \mathsf{B}, \cdot)$ **then**
    if $\mathtt{accounts}[\mathsf{acc}] = \varepsilon$ **then**
      Select $\mathsf{sk}, r \in \mathbb{Z}_q$ and compute $\mathsf{com} = g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}}$
      Send $(\textsc{prove}, \mathrm{sid}, (\mathsf{B}, \mathsf{com}), (\mathsf{sk}, r), \mathsf{P})$ to $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{acc_reg}}$ and get response $(\textsc{proof}, \mathrm{sid}, \pi)$ ◁ Prove relation in Equation (1)
      Append $(\mathsf{com}, \mathsf{B}, \mathsf{sk}, r)$ to $\mathtt{accounts}[\mathsf{acc}]$
      Send $(\textsc{diffuse}, \mathrm{sid}, ((\mathsf{com}, \mathsf{B}), \pi))$ to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$
    else
      Parse the last item in $\mathtt{accounts}[\mathsf{acc}]$ as $(\mathsf{com}, v, \mathsf{sk}, r)$
      Let $\mathcal{B} \leftarrow \mathrm{head}(\mathcal{C})$ and set rt^{MT} as the Merkle tree root in $\mathcal{B}$
      Compute $\mathsf{tag} \leftarrow h_3^{\mathsf{sk}}$.
      Select $r', \mathsf{sk}' \xleftarrow{\$} \mathbb{Z}_q$, and compute $\mathsf{com}' \leftarrow g^{r'} \cdot h_1^{\mathsf{B}+v}, h_2^{\mathsf{sk}'}$.
      Send $(\textsc{prove}, \mathrm{sid}, (rt^{\mathsf{MT}}, \mathsf{tag}, \mathsf{B}, \mathsf{com}'), (\mathsf{com}, v, \mathsf{sk}, r, \mathsf{sk}', r'), \mathsf{P})$ to $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{acc_upd}}$ and get response $(\textsc{proof}, \mathrm{sid}, \pi)$ ◁ Prove relation in Equation (2)
      Append $(\mathsf{com}', \mathsf{B} + v, \mathsf{sk}', r')$ to $\mathtt{accounts}[\mathsf{acc}]$
      Send $(\textsc{diffuse}, \mathrm{sid}, ((rt^{\mathsf{MT}}, \mathsf{tag}, \mathsf{B}, \mathsf{com}'), \pi))$ to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$
    end if
  else
    Compute $v \leftarrow \mathsf{fee}(\mathtt{tx})$ ◁ Pay fees
    Parse the last item in $\mathtt{accounts}[\mathsf{acc}]$ as $(\mathsf{com}, v, \mathsf{sk}, r)$
    Let $\mathcal{B} \leftarrow \mathrm{head}(\mathcal{C})$ and set rt^{MT} as the Merkle tree root in $\mathcal{B}$
    Compute $\mathsf{tag} \leftarrow h_3^{\mathsf{sk}}$.
    Select $r', \mathsf{sk}' \xleftarrow{\$} \mathbb{Z}_q$, and compute $\mathsf{com}' \leftarrow g^{r'} \cdot h_1^{\mathsf{B}-v}, h_2^{\mathsf{sk}'}$.
    Send $(\textsc{prove}, \mathrm{sid}, (rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}'), (\mathsf{com}, \mathsf{B}, \mathsf{sk}, r, \mathsf{sk}', r'), \mathsf{P})$ to $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{pay_fee}}$ and get response $(\textsc{proof}, \mathrm{sid}, \pi)$ ◁ Prove relation in Equation (3)
    Append $(\mathsf{com}', \mathsf{B} - v, \mathsf{sk}', r')$ to $\mathtt{accounts}[\mathsf{acc}]$
    ▷ Secret share $\mathtt{tx}$
    Set $i \leftarrow \lceil \max(\mathtt{sl} - R, 0)/(R/6) \rceil + 1$ ◁ Find i-th committee to share
    Parse ephemeral public keys of the i-th committee in $\mathcal{C}_{\mathsf{loc}}[0 : (i+4) \cdot R/6]$ as $(\mathsf{epk}_1, \ldots, \mathsf{epk}_k)$
    Let $t \leftarrow k/2 + 1$ and select $s \xleftarrow{\$} \{0,1\}^\kappa$.
    Compute $e \leftarrow \mathtt{tx} \oplus \mathcal{G}_{\mathsf{rpoRO}}(s)$ and $\{s_i\}_{i \in [k]} \leftarrow \mathsf{SS}_t(s, f_s)$.
    for $i \in [k]$ **do**
      Select $r_i \xleftarrow{\$} \{0,1\}^\kappa$ and compute $e_i \leftarrow \mathsf{PKE.Enc}(\mathsf{epk}_i, s_i; r_i)$.
    end for
    Send $(\textsc{prove}, \mathrm{sid}, (\{e_i\}_{i \in [k]}), (s, f_s, \{r_i\}_{i \in [k]}), \mathsf{P})$ to $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{ss_tx}}$ and get response $(\textsc{proof}, \mathrm{sid}, \pi')$ ◁ Prove relation in Equation (5)
    Send $((e, \{e_i\}_{i \in [k]}, \pi'), ((rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}'), \pi))$ to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$
  end if
end if

Protocol 12: *The procedure to issue a new transaction* $\mathtt{tx}$ *(with private account* acc*).*

The staking procedure. After parties update their stake distribution, they use independent

VRF evaluations to check if they are elected to propose the next block or generate the ephemeral keys.

Protocol $\mathsf{UpdateStakeDist}(\mathsf{P}, \mathrm{sid})$

1: Set $\mathbb{S}_{\mathtt{ep}}$ to be the stakeholder distribution at the end of epoch $\mathtt{ep}-2$ in $\mathcal{C}_{\mathsf{loc}}$ in case $\mathtt{ep} \geq 2$ (and keep the initial stake distribution in case $\mathtt{ep} < 2$).
2: Set $\alpha_{\mathsf{P}}^{\mathtt{ep}}$ to be the relative stake of P in $\mathbb{S}_{\mathtt{ep}}$, set $T_{\mathsf{P}}^{\mathtt{ep}} \leftarrow 2^{\ell_{\mathsf{VRF}}}\phi_f(\alpha_{\mathsf{P}}^{\mathtt{ep}})$ and $T^{\mathtt{ep},\mathsf{cmte}} \leftarrow 2^{\ell_{\mathsf{VRF}}}\alpha_{\mathsf{P}}^{\mathtt{ep}}$.
3: Set $\eta_{\mathtt{ep}}$ as the response after querying $\mathcal{G}_{\mathsf{RO}}$ with $(\textsc{eval}, \mathrm{sid}_{RO}, \eta_{\mathtt{ep}-1} \parallel \mathtt{ep} \parallel v)$ where v is the concatenation of the VRF outputs y_ρ from all blocks in $\mathcal{C}_{\mathsf{loc}}$ from the first $2R/3$ slots of epoch $\mathtt{ep}-1$.

OUTPUT: The protocol outputs τ, $\mathtt{ep}$, $\mathtt{sl}$, $\mathbb{S}_{\mathtt{ep}}$, $\alpha_{\mathsf{P}}^{\mathtt{ep}}$, $T_{\mathsf{P}}^{\mathtt{ep}}$, $T^{\mathtt{ep},\mathsf{cmte}}$ and $\eta_{\mathtt{ep}}$ to its caller (but not to $\mathcal{Z}$).

Protocol 13: *Update stake distribution.*

Protocol $\mathsf{StakingProcedure}(\mathsf{P}, \mathrm{sid})$

▷ The following steps are executed in an (MAINTAIN-LEDGER, sid, minerID)-interruptible manner:
▷ Determine leader status
1: Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \parallel \mathtt{sl} \parallel \mathtt{NONCE})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y_\rho, \pi_\rho)$.
2: Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \| \mathtt{sl} \| \mathtt{TEST})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y, \pi)$.
3: Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \parallel \mathtt{sl} \parallel \mathtt{COMMITEE})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y_{cmte}, \pi_{cmte})$.
4: Send $(\mathsf{EvalProve}, \mathrm{sid}, \eta_j \parallel \mathtt{sl} \parallel \mathtt{PROFILE})$ to $\mathcal{F}_{\mathsf{VRF}}$, denote the response from $\mathcal{F}_{\mathsf{VRF}}$ by $(\mathsf{Evaluated}, \mathrm{sid}, y_{pb}, \pi_{pb})$.
▷ Generate an ephemeral key-pair
5: **if** $y_{cmte} < T^{\mathtt{ep},\mathsf{cmte}}$ **then**
6: Run $(\mathsf{epk}, \mathsf{esk}) \leftarrow \mathsf{PKE.KeyGen}(1^\kappa)$ and compute $\{\mathsf{esk}_i\}_{i\in[n]} \leftarrow \mathsf{SS}_{n/2-1}(\mathsf{esk})$.
7: **for** $i \in [n]$ **do**
8: Select $r_i \xleftarrow{\$} \{0,1\}^\kappa$ and $\mathsf{pk}_i \xleftarrow{\$} \mathbb{S}_{\mathtt{ep}}$
9: Compute $c_i \leftarrow \mathsf{commit}(\mathsf{esk}_i, r_i)$ and $e_i \leftarrow \mathsf{anonPKE.Enc}(\mathsf{pk}_i, \mathsf{esk}_i \parallel r_i)$.
10: **end for**
11: Send $(\textsc{prove}, \mathrm{sid}, (\mathsf{epk}, \{c_i\}_{i\in[n]}), (\mathsf{esk}, f, \{r_i\}_{i\in[n]}), \mathsf{P})$ to $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{Key_Gen}}$ and get response $(\textsc{proof}, \mathrm{sid}, \pi_{\mathsf{Key_Gen}})$ ◁ Prove relation in Equation (4)
12: Erase esk and esk_i and all randomness used
13: Set $\mathsf{K} \leftarrow (\mathsf{epk}_i, \{c_i\}_{i\in[n]}, \pi_{\mathsf{Key_Gen}}, \{e_i\}_{i\in[n]})$.
14: Send $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}, (\mathsf{K}, \mathtt{sl}, (y_{cmte}, \pi_{cmte})), \mathtt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$; denote the response from $\mathcal{F}_{\mathsf{KES}}$ by $(\mathsf{Signature}, \mathrm{sid}, (\mathsf{K}, (y_{cmte}, \pi_{cmte})), \mathtt{sl}, \sigma)$
15: Send $(\textsc{diffuse}, \mathrm{sid}, (\mathsf{K}, \mathtt{sl}, (y_{cmte}, \pi_{cmte})), \sigma)$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$
16: **end if**
▷ Generate a new profile
17: **if** $y_{pb} < T^{\mathtt{ep},\mathsf{cmte}}$ **then**
18: Set $\mathtt{buffer}'_{\mathsf{tx}} \leftarrow \mathtt{buffer}_{\mathsf{tx}}$ and $\mathsf{pb} \leftarrow \varepsilon$
19: **repeat**
20: Parse $\mathtt{buffer}'_{\mathsf{tx}}$ as sequence $(\tilde{\mathtt{tx}}_1, \ldots, \tilde{\mathtt{tx}}_i)$ with order
21: **for** i **from** 1 **to** i **do**
22: **if** $\tilde{\mathtt{tx}}$ is not encrypted and is valid with $\vec{\mathtt{st}} \parallel \mathsf{Blockify}(\mathsf{pb})$ **then**
23: Set $\mathsf{pb} \leftarrow \mathsf{pb} \parallel \tilde{\mathtt{tx}}_i$ and remove $\tilde{\mathtt{tx}}_i$ from $\mathtt{buffer}'_{\mathsf{tx}}$

end if
if $\tilde{\mathtt{tx}}_i$ is an encrypted transaction and tag never appears in $\vec{\mathtt{st}} \parallel \mathsf{Blockify}(\mathsf{pb})$ **then**
Set $\mathsf{pb} \leftarrow \mathsf{pb} \parallel \tilde{\mathtt{tx}}_i$ and remove $\tilde{\mathtt{tx}}_i$ from $\mathtt{buffer}'_{\mathsf{tx}}$
end if
end for
until pb does not increase any more
Send $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}, (\mathsf{pb}, \mathtt{sl}, (y_{pb}, \pi_{pb})), \mathtt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$; denote the response from $\mathcal{F}_{\mathsf{KES}}$ by $(\mathsf{Signature}, \mathrm{sid}, (\mathsf{pb}, \mathtt{sl}, (y_{pb}, \pi_{pb})), \mathtt{sl}, \sigma)$
Send $(\textsc{diffuse}, \mathrm{sid}, (\mathsf{pb}, \mathtt{sl}, (y_{pb}, \pi_{pb}), \sigma))$ to $\mathcal{F}^{\mathsf{profile}}_{\mathsf{Diffuse}}$
end if
▷ Generate a new block
if $y < T^{\mathsf{ep}}_{\mathsf{P}}$ **then**
Set $\vec{N} \leftarrow \varepsilon$
for each $\mathsf{pb} \in \mathtt{buffer}_{\mathsf{pb}}$ **do**
if $\mathsf{pb} \notin \mathcal{C}_{\mathsf{loc}} \wedge \mathsf{TS}(\mathsf{pb}) \geq \max(1, \mathtt{sl} - 2R/3)$ **then** $\vec{N} \leftarrow \vec{N} \parallel \mathsf{pb}$
end for
for each $\mathsf{cmte} \in \mathtt{buffer}_{\mathsf{cmte}}$ **do**
if $\mathsf{cmte} \notin \mathcal{C}_{\mathsf{loc}} \wedge \mathsf{TS}(\mathsf{cmte}) \geq \max(1, \mathtt{sl} - 2R/3)$ **then** $\vec{N} \leftarrow \vec{N} \parallel \mathsf{cmte}$
end for
Set $\mathtt{st} \leftarrow \mathsf{Blockify}(\vec{N})$
Set $crt = (\mathsf{P}, y, \pi)$, $\rho = (y_\rho, \pi_\rho)$ and $h \leftarrow \mathcal{G}_{\mathsf{RO}}(\mathrm{head}(\mathcal{C}_{\mathsf{loc}}))$.
Send $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$; denote the response from $\mathcal{F}_{\mathsf{KES}}$ by $(\mathsf{Signature}, \mathrm{sid}, (h, \mathtt{st}, \mathtt{sl}, crt, \rho), \mathtt{sl}, \sigma)$
Set $\mathcal{B} \leftarrow (h, \mathtt{st}, \mathtt{sl}, crt, \rho, \sigma)$ and update $\mathcal{C}_{\mathsf{loc}} \leftarrow \mathcal{C}_{\mathsf{loc}} \parallel \mathcal{B}$
Send $(\textsc{diffuse}, \mathrm{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$
else
Evolve the KES signing key by sending $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}, 0, \mathtt{sl})$ to $\mathcal{F}_{\mathsf{KES}}$ and set the anchor at end of procedure to resume on next maintenance activation
end if

Protocol 14: *The main staking procedure of generating new blocks, profiles and committees.*

The round finish procedure. Once a party has finished her computation in a round, she advances the synchronous computation by sending the indication to $\mathcal{G}_{\mathsf{Clock}}$ (after receiving an update-request by the environment).

Protocol FinishRound(P)

while A $(\textsc{clock-update}, \mathrm{sid}_C)$ has not been received during the current round **do**
Give up activation (set the anchor here)
end while
Send $(\textsc{clock-update}, \mathrm{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$ ◁ Party will lose its activation here

Protocol 15: *Parties finish all computations in this round and wait for the next.*

Ledger maintenance. We pack here all the steps in the main ledger operation.

Protocol LedgerMaintenance(P)

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

1: **if** $\texttt{isInit} = \mathsf{false}$ **then**
2: Invoke Initialization(P, sid); if the invocation halts then halt (this will abort the execution).
3: **end if**
4: Invoke FetchInformation(P, sid) to receive the newest messages for this round; denote the output by $\mathcal{C}_1, \ldots, \mathcal{C}_M$, $\tilde{\texttt{tx}}_1, \ldots, \tilde{\texttt{tx}}_k$, $\mathsf{cmte}_1, \ldots, \mathsf{cmte}_i$, $\mathsf{pb}_1, \ldots, \mathsf{pb}_j$, and record the flag WELCOME if not yet recorded for this round ◁ Protocol 8
5: Set $\texttt{buffer}_{\mathsf{tx}} \leftarrow \texttt{buffer}_{\mathsf{tx}} \,\|\, (\tilde{\texttt{tx}}_1, \ldots, \tilde{\texttt{tx}}_k)$
6: Set $\texttt{buffer}_{\mathsf{cmte}} \leftarrow \texttt{buffer}_{\mathsf{cmte}} \,\|\, (\mathsf{cmte}_1, \ldots, \mathsf{cmte}_i)$ ◁ Remove messages with bad NIZK
7: Set $\texttt{buffer}_{\mathsf{pb}} \leftarrow \texttt{buffer}_{\mathsf{pb}} \,\|\, (\mathsf{pb}_1, \ldots, \mathsf{pb}_j)$
8: Define $\mathcal{N} \leftarrow \{\mathcal{C}_1, \ldots, \mathcal{C}_M\}$
9: Invoke UpdateTime(P) to update τ, ep and sl ◁ Protocol 9
10: Set $t_{\mathsf{on}} \leftarrow \tau$
11: Invoke SelectChain(P, sid, $\mathcal{C}_{\mathsf{loc}}, \mathcal{N}$) to replace $\mathcal{C}_{\mathsf{loc}}$ ◁ Protocol 11
12: **if** $t_{\mathsf{work}} < \tau$ **then**
13: Invoke UpdateStakeDist(P) to update $\mathbb{S}_{\mathsf{ep}}, \alpha_{\mathsf{P}}^{\mathsf{ep}}, T_{\mathsf{P}}^{\mathsf{ep}}$ and η_{ep} ◁ Protocol 13
14: Call StakingProcedure(P, sid) ◁ Protocol 14
15: Set $t_{\mathsf{work}} \leftarrow \tau$ ◁ React on a joining party
16: **if** WELCOME $= 1$ **then**
17: Send $(\mathsf{diffuse}, \mathsf{sid}, \mathcal{C}_{\mathsf{loc}})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ and $(\mathsf{diffuse}, \mathsf{sid}, \texttt{buffer}_{\mathsf{tx}})$ to $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$
18: **else**
19: Give up activation and resume below upon next maintenance activation
20: **end if**
21: **end if**
22: Invoke FinishRound(P) ◁ Protocol 15

Protocol 16: *The main ledger maintenance operations.*

6 Security Analysis

Overview. The ideal world consists of the dummy parties, the ledger functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$, the clock $\mathcal{G}_{\mathsf{Clock}}$, the random oracles $\mathcal{G}_{\mathsf{RO}}$ and $\mathcal{G}_{\mathsf{rpoRO}}$, transaction diffusion network $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ and the simulator $\mathcal{S}$. $\mathcal{S}$ runs a black-box adversary $\mathcal{A}$, and simulates the network functionality $\mathcal{F}_{\mathsf{Diffuse}}$ (which handles all protocol message diffusion except transactions). $\mathcal{S}$ also runs the simulated NIZK functionality $\mathcal{F}_{\mathsf{NIZK}}$, CRS functionality $\mathcal{F}_{\mathsf{CRS}}$, PoS initialization functionality $\mathcal{F}_{\mathsf{INIT}}$, VRF functionality $\mathcal{F}_{\mathsf{VRF}}$ and key-evolving functionality $\mathcal{F}_{\mathsf{KES}}$.

The simulator in our proof simulates the ledger maintenance procedure (for blocks, ephemeral committees and profiles) for each dummy party, and jointly builds a PoS blockchain with $\mathcal{A}$.

Whenever the ledger $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ notifies $\mathcal{S}$ that a dummy party P submits a new private transaction of length m with txid at time τ, $\mathcal{S}$ (see Simulator - Part 3) simulates the IssueNewTransaction procedure for P by first picking a random secret s and shares s towards the corresponding committee; then $\mathcal{S}$ generates a fake, randomly chosen ciphertext of $\texttt{tx}$ and send them to the simulated network. In order to learn $\texttt{tx}$, $\mathcal{S}$ waits until that in the simulated blockchain, all transactions that might precede the transaction with txid get settled. $\mathcal{S}$ proposes the sequence of transactions to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ (using txid) and thus learns the transaction $\texttt{tx}$ that is associated with txid. $\mathcal{S}$ then programs the $\mathcal{G}_{\mathsf{rpoRO}}$ on point s (and hides this "isPrograMed" information to $\mathcal{A}$) and equivocate the corresponding

encrypted transactions to the real honest transaction. Note that since the time duration that committee secret key shares are kept secret is set slightly longer than the time that $\mathtt{tx}$ will get settled on the blockchain, $\mathcal{S}$ always has enough time to program $\mathcal{G}_{\mathsf{rpoRO}}$ and equivocate the ciphertext to $\mathtt{tx}$. When an encrypted transaction is received from $\mathcal{A}$, $\mathcal{S}$ extracts the plain transaction from the simulated NIZK functionality $\mathcal{F}_{\mathsf{NIZK}}$ (since $\mathcal{A}$ has to prove that he secret shares the one-time pad key to encrypt the transaction).

In order for the simulator to set up private accounts and pay transaction fees, $\mathcal{S}$ gives out faked account commitment and update proof as he fully controls $\mathcal{F}_{\mathsf{NIZK}}$. Regarding adaptive corruption (see Simulator - Part 4), once $\mathcal{A}$ decides to corrupt a party P, the simulator learns the whole transaction history of P. For each transaction from input $(\text{SUBMIT}, \mathtt{tx}, \mathsf{acc})$, $\mathcal{S}$ equivocates on the corresponding account commitment com by solving $r + \log_g h_1 \cdot \mathsf{B} + \log_g h_2 \cdot \mathsf{sk}$. Note that since $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{CRS}}$, he knows $\log_g h_1$, $\log_g h_2$ and sk (as sk is give out as $\mathsf{tag} = h_3^{\mathsf{sk}}$), after corruption he also learns B thus he can compute r such that $g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}} = \mathsf{com}$.

Our simulator terminates when certain bad event happens. We define the following bad events BAD-CP, BAD-CQ, BAD-CG (violation of common prefix, chain quality and chain growth respectively), BAD-PROFILE (violation of honest profile majority), BAD-COMMITTEE (failure of committee that honest majority does not hold) and BAD-Program (failure of equivocating any transaction). We prove that, with overwhelming probability in terms of the security parameter, none of these bad events happen. Hence, the simulator can run the simulation for any polynomial time without abortion.

Simulator $\mathcal{S}_{\text{ledger}}$ (Part 1 - Main structure)

Overview:

- The simulator internally emulates all local UC functionalities by running the code (and keeping the state) of $\mathcal{F}_{\mathsf{Diffuse}}$, $\mathcal{F}_{\mathsf{CRS}}$, $\mathcal{F}_{\mathsf{NIZK}}$, $\mathcal{F}_{\mathsf{VRF}}$, $\mathcal{F}_{\mathsf{KES}}$, $\mathcal{F}_{\mathsf{INIT}}$.
- The simulator mimics the execution of $\Pi_{\mathsf{ledger}}^{\mathsf{fair++}}$ for each honest party P (including their state and the interaction with the hybrids).
- The simulator emulates a view towards the adversary $\mathcal{A}$ in a black-box way, i.e., by internally running adversary $\mathcal{A}$ and simulating his interaction with the protocol (and hybrids) as detailed below for each hybrid. To simplify the description, we assume $\mathcal{A}$ does not violate the theorem assumptions (as they are enforced by a wrapper $\mathcal{W}^{PoS}(\cdot)^a$).

Party sets:

- We denote $\mathcal{S}_{\text{alert}}$ the alert parties (synchronized and executing the protocol) and use $\mathcal{S}_{\text{syncStalled}}$ shorthand for parties that are synchronized (and hence time aware and online) but stalled. We denote by $\mathcal{P}_{DS}$ all honest but de-synchronized parties (both operational or stalled).
- For each registered honest party, the simulator maintains the local state containing in particular the local chain $\mathcal{C}_{\mathsf{loc}}^{\mathsf{P}}$, the time t_{on} it remembers when last being online. For each party P and clock time τ, the simulator stores a flag $\mathsf{update}_{\mathsf{P},\tau}$ (initially false) to remember whether this party has updated its state already in this round. Note that an registered party is registered with all its local hybrid.
- Upon any activation, the simulator will query the current party set from the ledger, the clock, and the random oracle to evaluate in which category an honest party belongs to. If a new honest party is registered to the ledger, it internally runs the initialization procedure of $\Pi_{\mathsf{ledger}}^{\mathsf{fair++}}$.
- We assume that the simulator queries upon any activation for the sequence $\vec{\mathcal{I}}_H^T$, and the current time τ from the clock. We note that the simulator is capable of determining $\mathsf{predict\text{-}time}(\cdot)$ of $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$.

Messages from the Clock: Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ from $\mathcal{G}_{\mathsf{Clock}}$, if P is an honest

registered party, then remember that this party has received such a clock update (and the environment gets an activation). Otherwise, send $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$.

Messages from the global random oracle: $\mathcal{S}$ acts as a dummy adversary and forward all responses from $\mathcal{G}_{\mathsf{rpoRO}}$ to its requestor except that upon receiving a $(\text{IS-PROGRAMMED}, m)$ request to $\mathcal{G}_{\mathsf{rpoRO}}$, return $(\text{IS-PROGRAMMED}, \mathsf{false})$ if m is previously programed by $\mathcal{S}$.

Messages from the ledger:

- Upon receiving $(\text{SUBMIT}, m, \text{txid}, \tau, \mathsf{P})$ from $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$, do the following:
  1. **If** m is a message sent by $\mathcal{S}$ upon receiving $(\text{DIFFUSE}, \text{sid}, m')$ from $\mathcal{A}$ to $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ (refer to the interaction with global transaction diffusion network), **then** return m' to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$.
  2. **Else**, invoke $\text{IssueNewTransaction}(P, \tau, m)$ and return the response to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$.
- Upon receiving $(\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ from $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$, extract from $\vec{\mathcal{I}}_H^T$ the party P that issued this query. If P has already completed its round-task, then ignore this request. Otherwise, execute $\text{SimulateStaking}(\mathsf{P}, \tau_{\mathcal{L}})$.

Messages from the global transaction diffusion network: $\mathcal{S}$ acts as a dummy adversary and forward all responses from $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ to its requestor except that upon receiving a $(\text{DIFFUSE}, \text{sid}, m)$ request to $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ from $\mathcal{A}$, before forwarding the response to $\mathcal{A}$, depending on the type of transaction $\mathcal{A}$ is about to send, $\mathcal{S}$ does the following.

- Message m is a plain transaction: send $(\text{SUBMIT}, \text{sid}, m, \bot)$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$.
- Message $m = ((\mathsf{B}, \mathsf{com}), \pi)$ sets up a new account:
  If $((\mathsf{B}, \mathsf{com}), (\mathsf{sk}, r), \pi)$ is not stored in the simulated $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{acc_reg}}$, **then** ignore the message; **else** sample a uniformly random string acc and send $(\text{SUBMIT}, \text{sid}, (0^\kappa, \mathsf{B}, \cdot), \mathsf{acc})$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$.
- Message $m = ((rt^{\mathsf{MT}}, \mathsf{tag}, \mathsf{B}, \mathsf{com}'), \pi)$ updates an existing account:
  If $((rt^{\mathsf{MT}}, \mathsf{tag}, \mathsf{B}, \mathsf{com}'), (\mathsf{com}, v, \mathsf{sk}, r, \mathsf{sk}', r'), \pi)$ is not stored in the simulated $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{acc_upd}}$ **or** , **then** ignore the message; **else** let acc denote the string chosen with respect to sk and send $(\text{SUBMIT}, \text{sid}, (0^\kappa, \mathsf{B}, \cdot), \mathsf{acc})$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$.
- Message $m = ((e, \{e_i\}_{i\in[k]}, \pi'), ((rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}'), \pi))$ sends a private transaction:
  If $((rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}'), (\mathsf{com}, \mathsf{B}, \mathsf{sk}, r, \mathsf{sk}', r'), \pi')$ is not stored in the simulated $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{pay_fee}}$, **then** ignore the message. **If** $((\{e_i\}_{i\in[k]}), (s, f_s, \{r_i\}_{i\in[k]}), \pi)$ is not stored in the simulated $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{ss_tx}}$, **then** ignore the message. **Otherwise**, let acc denote the string chosen with respect to sk and $\mathtt{tx}$ denote the transaction such that $\mathtt{tx} = \mathcal{G}_{\mathsf{rpoRO}}(s) \oplus e$, send $(\text{SUBMIT}, \text{sid}, \mathtt{tx}, \mathsf{acc})$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$.

[a] Refer to [BGK+18] for detailed discussions on the PoS wrapper.

Simulator $\mathcal{S}_{\text{ledger}}$ (Part 2 - Black-Box Interaction)

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

- The simulator relays back and forth the communication between the (internally emulated) $\mathcal{F}_{\mathsf{INIT}}$ functionality and the adversary $\mathcal{A}$ acting on behalf of a corrupted party.
- If at time $\tau = 0$, a corrupted party $\mathsf{P} \in \mathcal{S}_{\text{initStake}}$ registers via $(\mathsf{ver_keys}, \text{sid}, \mathsf{P}, v_{\mathsf{P}}^{\text{vrf}}, v_{\mathsf{P}}^{\text{kes}})$ to $\mathcal{F}_{\mathsf{INIT}}$, then input $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ on behalf of P.

Simulation of the Functionalities $\mathcal{F}_{\mathsf{KES}}$ and $\mathcal{F}_{\mathsf{VRF}}$ towards $\mathcal{A}$:

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

Simulation of the Network $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ (over which chains are sent) towards $\mathcal{A}$:

- Upon receiving $(\textsc{diffuse}, \mathrm{sid}, (\mathcal{C}_{i_1}, U_{i_1}), \ldots, (\mathcal{C}_{i_\ell}, U_{i_\ell}))$ with a list of chains and corresponding parties from $\mathcal{A}$ (or on behalf some corrupted $\mathsf{P} \in \mathcal{P}$), then do the following:
  1. Relay this input to the simulate network functionality and record its response to $\mathcal{A}$.
  2. Execute $\textsc{ExtendLedgerState}(\tau_{\mathcal{L}})$.
  3. Provide $\mathcal{A}$ with the recorded output of the simulated network.
- Upon receiving $(\textsc{diffuse}, \mathrm{sid}, \mathcal{C})$ from $\mathcal{A}$ on behalf of some *corrupted* party P, then do the following:
  1. Relay this input to the simulate network functionality and record its response to $\mathcal{A}$.
  2. Execute $\textsc{ExtendLedgerState}(\tau_{\mathcal{L}})$.
  3. Provide $\mathcal{A}$ with the recorded output of the simulated network.
- Upon receiving $(\textsc{fetch}, \mathrm{sid})$ from $\mathcal{A}$ on behalf some corrupted $\mathsf{P} \in \mathcal{P}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\textsc{delays}, \mathrm{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ and record the answer to $\mathcal{A}$. Before giving this answer to $\mathcal{A}$, query the ledger state $\mathtt{state}$ and execute $\textsc{AdjustView}(\mathtt{state}, \tau_{\mathcal{L}})$.
- Upon receiving $(\textsc{swap}, \mathrm{sid}, \mathsf{mid}, \mathsf{mid}')$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ and record the answer to $\mathcal{A}$. Before giving this answer to $\mathcal{A}$, query the ledger state $\mathtt{state}$ and execute $\textsc{AdjustView}(\mathtt{state}, \tau_{\mathcal{L}})$.

Simulation of the Network $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$ resp.) (over which input blocks (enclave public keys resp.) are sent) towards $\mathcal{A}$:

- Upon receiving $(\textsc{diffuse}, \mathrm{sid}, m)$ from $\mathcal{A}$ with an input block m on behalf some corrupted $\mathsf{P} \in \mathcal{P}$, then do the following:
  1. Forward the request to the internally simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$ resp.), which replies for each message with a message-ID mid.
  2. Remember the association between mid and the corresponding input block.
  3. Provide $\mathcal{A}$ with whatever the network outputs.
- Upon receiving $(\textsc{fetch}, \mathrm{sid})$ from $\mathcal{A}$ on behalf some corrupted $\mathsf{P} \in \mathcal{P}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$ resp.) and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\textsc{delays}, \mathrm{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$ resp.) and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\textsc{swap}, \mathrm{sid}, \mathsf{mid}, \mathsf{mid}')$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{profile}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{cmte}}$ resp.) and return whatever is returned to $\mathcal{A}$.

Simulation of the Functionalities $\mathcal{F}_{\mathsf{NIZK}}$ towards $\mathcal{A}$:

- The simulator relays back and forth the communication between the (internally emulated) $\mathcal{F}_{\mathsf{NIZK}}$ functionality and the adversary $\mathcal{A}$ acting on behalf of a corrupted party.

Simulation of the Functionalities $\mathcal{F}_{\mathsf{CRS}}$ towards $\mathcal{A}$:

- Upon first query, $\mathcal{S}$ picks random g, x_1, x_2, x_3 and sets $h_1 = g^{x_1}, h_2 = g^{x_2}$ and $h_3 = g^{x_3}$ and returns (g, h_1, h_2, h_3) to the requestor.
- For the second and later queries, return (g, h_1, h_2, h_3).

Simulator $\mathcal{S}_{\text{ledger}}$ (Part 3 - Internal Procedures)

1: **procedure** $\textsc{SimulateStaking}(\mathsf{P}, \tau)$ ◁ Simulate the mining procedure of P in round τ
2: Execute $\mathsf{FetchInformation}$ of party P and skip the transaction fetch subroutines
3: Execute $\mathsf{UpdateStakeDist}$ of party P
4: Extract the list of transactions P shall recevie (from interactions with $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ and $\mathcal{A}$'s instruc-

tions to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$)
if $\mathsf{Update}_{\mathsf{P},\tau}$ **then**
Send (CLOCK-UPDATE, sid_C, P) to $\mathcal{A}$ if $\mathcal{S}_{\mathrm{ledger}}$ has received such an input in round τ
else
Execute $\mathsf{StakingProcedure}(\mathsf{P}, \mathrm{sid})$ and set $\mathsf{Update}_{\mathsf{P},\tau} \leftarrow \mathsf{true}$
Before the activation goes to $\mathcal{A}$, execute ExtendLedgerState(τ)
end if
Remember that party P has completed for this round τ
end procedure

procedure ExtendLedgerState(τ)
for party $U_p \in \{\mathcal{S}_{\mathrm{alert}} \cup \mathcal{S}_{\mathrm{syncStalled}}\}$ of round τ **do**
Let $\mathcal{C}_{U_p}$ be the party's currently stored local chain.
Determine the number of rounds $\rho^{(Up)}$ this party legs behind τ, i.e., $\rho^{(Up)} = \tau - t^{(Up)}_{\mathsf{on}}$
Let $\mathcal{C}^{(Up)}_1, \ldots, \mathcal{C}^{(Up)}_k$ be the chains contained in the receiver buffer of $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$ with delay at most $\rho^{(Up)}$.
Execute $\mathsf{SelectChain}(\mathcal{C}_{U_p}, \{\mathcal{C}^{(Up)}_1, \ldots, \mathcal{C}^{(Up)}_k\})$ and $\mathsf{ReadState}(U_p, \mathrm{sid})$ and let the chain's decoded state be $\vec{\mathtt{st}}_{U_p}$
end for
Let $\vec{\mathtt{st}}$ be the longest state among all such states $\vec{\mathtt{st}}_{U_p}$ where $U_p \in \{\mathcal{S}_{\mathrm{alert}} \cup \mathcal{S}_{\mathrm{syncStalled}}\}$
if $|\mathtt{state}| > |\vec{\mathtt{st}}^{\lceil k}|$ **then** Execute AdjustView($\mathtt{state}$)
if $\mathtt{state}$ is not a prefix of $\vec{\mathtt{st}}^{\lceil k}$ **then**
Abort simulation: consistency violation. ◁ Event BAD-CP
end if
Define the difference diff to be the block sequence s.t. $\mathtt{state} \parallel \mathsf{diff} = \vec{\mathtt{st}}^{\lceil k}$.
Parse $\mathsf{diff} = \mathsf{diff}_1 \parallel \ldots \parallel \mathsf{diff}_n$.
for j **from** 1 **to** n **do**
Map each transaction $\mathtt{tx}$ in this block to its unique transaction ID txid.
Let $\mathsf{list}_j = (\mathrm{txid}_{j,1}, \ldots, \mathrm{txid}_{j,\ell_j})$ be the corresponding list for this block diff_j
if coinbase $\mathrm{txid}_{j,1}$ specifies a party honest at block creation time **then**
hFlag $\leftarrow 1$
else
hFlag $\leftarrow 0$
end if
Output (NEXT-BLOCK, list_j) to $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ and receive (NEXT-BLOCK, ok) as an immediate answer.
end for
if Fraction of blocks with hFlag $= 0$ in the recent k blocks $> 1 - \mu_{\mathsf{CQ}}$ **then**
Abort simulation: chain quality violation ◁ Event BAD-CQ
else if State increases less than k blocks during the last k/τ_{CG} rounds **then**
Abort simulation: chain growth violation ◁ Event BAD-CG
else if Honest profiles account for less than half for any interval of length $R/6$ staring before $\tau - 2R/3$ **then**
Abort simulation: profile block violation ◁ Event BAD-PROFILE
end if
▷ Equivocate honest transactions.
Send (READ, sid) to $\mathcal{G}^{\mathsf{Fair++}}_{\mathsf{Ledger}}$ and receive $\mathtt{state}$
for $\mathtt{tx} \in \mathtt{state}$ **and** txid corresponds to a faked transaction **and** $\mathsf{equivocate}_{\mathtt{tx}} = \mathsf{false}$ **do**
Let s denote the secret used in faking $\mathtt{tx}$ and e the faked encryption
Send (PROGRAM, s, h) such that $h \oplus e = \mathtt{tx}$ to $\mathcal{G}_{\mathsf{rpoRO}}$

if programming $\mathcal{G}_{\mathsf{rpoRO}}$ fails **then**
Abort simulation: failure of equivocation ◁ Event BAD-Program
end if
Set $\mathsf{equivocate}_{\mathtt{tx}} \leftarrow \mathsf{true}$
end for
Execute AdjustView($\mathtt{state} \parallel \mathsf{diff}$)
end procedure

▷ Adjust the view of synchronized parties.
procedure AdjustView($\mathtt{state}, \tau$)
$\mathsf{pointers} \leftarrow \varepsilon$
for party $U_p \in \mathcal{H}$ of round τ **do**
Let $\mathcal{C}_{U_p}$ be the party's currently stored local chain.
Determine the number of rounds $\rho^{(Up)}$ this party legs behind τ, i.e., $\rho^{(Up)} = \tau - t_{\mathsf{on}}^{(Up)}$
Let $\mathcal{C}_1^{(Up)}, \ldots, \mathcal{C}_k^{(Up)}$ be the chains contained in the receiver buffer of $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ with delay at most $\rho^{(Up)}$.
Execute $\mathsf{SelectChain}(\mathcal{C}_{U_p}, \{\mathcal{C}_1^{(Up)}, \ldots, \mathcal{C}_k^{(Up)}\})$ and $\mathsf{ReadState}(U_p, \mathrm{sid})$ and let the chain's decoded state be $\vec{\mathtt{st}}_{U_p}$
end for
for each synchronized party $U_p \in (\mathcal{H} \backslash \mathcal{P}_{DS})$ of round τ **do**
Determine the pointers $\mathtt{pt}_{U_p}$ s.t. $\vec{\mathtt{st}}_{U_p}^{\lceil k} = \mathtt{state}|_{\mathtt{pt}_{U_p}}$
if such a point does not exist **then**
return ◁ Call on invalid input or event BAD-CP occurred
end if
if $\mathsf{Update}_{U_p,\tau}$ **then**
$\mathsf{pointers} \leftarrow \mathsf{pointers} \parallel (U_p, \mathtt{pt}_{U_p})$
end if
end for
Output (SET-SLACK, pointers) to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$
▷ Adjust the view of desynchronized parties
$\mathsf{pointers} \leftarrow \varepsilon$
$\mathsf{desyncStates} \leftarrow \varepsilon$
for each desynchronized party $U_p \in \mathcal{P}_{DS}$ of round τ **do**
if $\mathsf{Update}_{U_p,\tau} = \mathsf{false}$ **then**
Set $\mathtt{pt}_{U_p}$ to be $|\vec{\mathtt{st}}_{U_p}^{\lceil k}|$
$\mathsf{pointers} \leftarrow \mathsf{pointers} \parallel (U_p, \mathtt{pt}_{U_p})$
$\mathsf{desyncStates} \leftarrow \mathsf{desyncStates} \parallel (U_p, \vec{\mathtt{st}}_{U_p}^{\lceil k})$
end if
Output (SET-SLACK, pointers) to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$
Output (DESYNC-STATE, desyncStates) to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$
end for
end procedure

procedure IssueNewTransaction(P, τ, m) ◁ Issue dummy transactions.
if m is neither a funding transaction nor private one **then return** m
if m is of form $(0^\kappa, \mathsf{B}, \cdot)$ that sets up an account **then**
Pick $\mathsf{com} \xleftarrow{\$} \mathbb{Z}_q$ and $\pi \leftarrow \{0,1\}^\kappa$
Remember that on $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{acc_reg}}$ return 1 for query $(\mathrm{VERIFY}, \mathrm{sid}, (\mathsf{B}, \mathsf{com}), \pi)$

return $((\mathsf{B}, \mathsf{com}), \pi)$
end if
Let $\mathcal{C}$ denote the simulated blockchain for party P
Set rt^{MT} as the latest Merkle tree root in $\mathcal{C}$
if m is of form $(0^\kappa, \mathsf{B}, \cdot)$ that updates an account **then**
Pick $\mathsf{com}, \mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ and $\pi \leftarrow \{0,1\}^\kappa$
Compute $\mathsf{tag} \leftarrow h_3^{\mathsf{sk}}$ and record sk for tag.
Remember that on $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{acc_upd}}$ return 1 for query $(\text{VERIFY}, \text{sid}, (rt^{\mathsf{MT}}, \mathsf{tag}, \mathsf{B}, \mathsf{com}), \pi)$
return $((rt^{\mathsf{MT}}, \mathsf{tag}, \mathsf{B}, \mathsf{com}), \pi)$
end if
▷ Issue faked private transaction
if m denotes the transaction size **then**
Compute $v \leftarrow \mathsf{fee}(m)$
Pick $\mathsf{com}, \mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ and $\pi \leftarrow \{0,1\}^\kappa$
Compute $\mathsf{tag} \leftarrow h_3^{\mathsf{sk}}$ and record sk for tag.
Remember that on $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{pay_fee}}$ return 1 for query $(\text{VERIFY}, \text{sid}, (rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}), \pi)$
Set $i \leftarrow \lceil \max(\mathtt{sl} - R, 0)/(R/6) \rceil + 1$ ◁ Find i-th committee to share
Parse ephemeral public keys on $\mathcal{C}$ of the i-th committee in $\mathcal{C}_{\mathsf{loc}}[0 : (i+4) \cdot R/6]$ as $(\mathsf{epk}_1, \ldots, \mathsf{epk}_k)$
Let $t \leftarrow k/2 + 1$ and select $e, s \xleftarrow{\$} \{0,1\}^\kappa$.
Compute $\{s_i\}_{i \in [k]} \leftarrow \mathsf{SS}_t(s, f_s)$.
for $i \in [k]$ **do**
Select $r_i \xleftarrow{\$} \{0,1\}^\kappa$ and compute $e_i \leftarrow \mathsf{PKE.Enc}(\mathsf{epk}_i, s_i; r_i)$.
end for
Send $(\text{PROVE}, \text{sid}, (\{e_i\}_{i \in [k]}), (s, f_s, \{r_i\}_{i \in [k]}), \mathsf{P})$ to $\mathcal{F}_{\mathsf{NIZK}}$ for NP-relation $\mathsf{R}_{\mathsf{ss_tx}}$ and get response $(\text{PROOF}, \text{sid}, \pi')$
return $((e, \{e_i\}_{i \in [k]}, \pi'), ((rt^{\mathsf{MT}}, \mathsf{tag}, v, \mathsf{com}), \pi))$
end if
end procedure

Simulator $\mathcal{S}_{\text{ledger}}$ (Part 4 - Adaptive Corruptions)

Upon $\mathcal{A}$ corrupts a party P during the protocol execution, $\mathcal{S}$ learns all submit inputs $(\text{SUBMIT}, \text{sid}, \mathtt{tx}_1, \mathsf{acc}_1), \ldots, (\text{SUBMIT}, \text{sid}, \mathtt{tx}_k, \mathsf{acc}_k)$ from P (which is ordered consistently as they are received) and executes the following procedure for each $(\text{SUBMIT}, \text{sid}, \mathtt{tx}_i, \mathsf{acc}_i)$:

if $\mathsf{acc} \neq \perp$ **then**
if acc is later updated/spent with tag **then**
Parse sk related with tag
else
$\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$
end if
if $\mathtt{tx}$ is of form $(0^\kappa, \mathsf{B}, \cdot)$ and acc is new to P **then**
Parse the message sent for $\mathtt{tx}$ as $(\mathtt{tx}, \mathsf{com}, \cdot)$
Compute r such that $g^r \cdot h_1^{\mathsf{B}} \cdot h_2^{\mathsf{sk}} = \mathsf{com}$ ◁ $\mathcal{S}$ knows $\log_g h_2$
Append $(\mathsf{com}, \mathsf{B}, \mathsf{sk}, r)$ to $\mathtt{accounts}[\mathsf{acc}]$

126: **else if** $\mathtt{tx}$ is of form $(0^\kappa, \mathsf{B}, \cdot)$ and acc has been seen by P **then**
127: Parse the message sent for $\mathtt{tx}$ as $(\mathtt{tx}, \mathsf{com}, \cdot, \cdot)$
128: Parse the last item in $\mathtt{accounts}[\mathsf{acc}]$ as $(\mathsf{com}', v, \mathsf{sk}', r')$
129: Compute r such that $g^r \cdot h_1^{\mathsf{B}+v} \cdot h_2^{\mathsf{sk}} = \mathsf{com}$
130: Append $(\mathsf{com}, \mathsf{B}+v, \mathsf{sk}, r)$ to $\mathtt{accounts}[\mathsf{acc}]$
131: **else**
132: Compute $v' \leftarrow \mathsf{fee}(\mathtt{tx})$
133: Parse the message sent for $\mathtt{tx}$ as $(\mathtt{tx}, \mathsf{com}, \cdot, \cdot)$
134: Parse the last item in $\mathtt{accounts}[\mathsf{acc}]$ as $(\mathsf{com}', v, \mathsf{sk}', r')$
135: Compute r such that $g^r \cdot h_1^{v-v'} \cdot h_2^{\mathsf{sk}} = \mathsf{com}$
136: Append $(\mathsf{com}, v-v', \mathsf{sk}, r)$ to $\mathtt{accounts}[\mathsf{acc}]$
137: **end if**
138: **end if**
139: **if** $\mathtt{tx}_i$ is sent privately and $\mathsf{equivocate}_{\mathtt{tx}_i} = \mathsf{false}$ **then**
140: Let s denote the secret used in faking $\mathtt{tx}_i$ and e the faked encryption
141: Send $(\textsc{program}, s, h)$ such that $h \oplus e = \mathtt{tx}_i$ to $\mathcal{G}_{\mathsf{rpoRO}}$
142: **if** programming $\mathcal{G}_{\mathsf{rpoRO}}$ fails **then**
143: **Abort** simulation: failure of equivocation ◁ Event BAD-Program
144: **end if**
145: Set $\mathsf{equivocate}_{\mathtt{tx}_i} \leftarrow \mathsf{true}$
146: **end if**

6.1 Composable Guarantees

Blockchain security. Recall the following blockchain security properties in [BGK+18] which can be useful for proving our composable security.

- **Common Prefix (CP); parameterized with $k \in \mathbb{N}^+$.** The chains $\mathcal{C}_1, \mathcal{C}_2$ possessed by two alert parties at the onset of the slots $\mathtt{sl}_1 \leq \mathtt{sl}_2$ are such that $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$, where $\mathcal{C}_1^{\lceil k}$ denotes the chain obtained by removing the last k blocks from $\mathcal{C}_1$, and $\preceq$ denotes the prefix relation.
- **Chain Growth (CG); parameterized with $\tau \in (0,1]$ and $s \in \mathbb{N}$.** Consider a chain $\mathcal{C}$ possessed by an alert party at the onset of a slot $\mathtt{sl}$. Let $\mathtt{sl}_1$ and $\mathtt{sl}_2$ be two previous rounds for which $\mathtt{sl}_1 + s \leq \mathtt{sl}_2 \leq \mathtt{sl}$. Then $|\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]| \geq \tau \cdot s$ where τ is the speed coefficient.
- **Chain Quality (CQ); parameterized with $\mu \in (0,1]$ and $s \in \mathbb{N}$.** Consider any portion of length at least s of the chain possessed by an alert party at the onset of a round; the ratio of blocks originating from alert parties is at least μ. We call μ the chain quality coefficient.
- **Existential Chain Quality (∃CQ); with parameter $s \in \mathbb{N}$.** Consider a chain $\mathcal{C}$ possessed by an alert party at the onset of a slot $\mathtt{sl}$. Let $\mathtt{sl}_1$ and $\mathtt{sl}_2$ be two previous slots for which $\mathtt{sl}_1 + s \leq \mathtt{sl}_2 \leq \mathtt{sl}$. Then $\mathcal{C}[\mathtt{sl}_1 : \mathtt{sl}_2]$ contains at least one alertly generated block.

Theorem 1 ([BGK+18], restated). *Consider the execution of Ouroboros-Genesis with adversary $\mathcal{A}$ and environment $\mathcal{Z}$. Let f be the active-slot coefficient, let Δ be the upper bound on the network delay and let Q be an upper bound on the total number of queries issued to $\mathcal{G}_{\mathsf{RO}}$. Let $\alpha, \beta \in [0,1]$ denote a lower bound on the alert ratio and participating ratio throughout the whole execution, respectively. Let R and L denote the epoch length and the total lifetime of the system (in slots) and let k, s be the* maxvalid-bg *parameter. If for some $\epsilon \in (0,1)$ we have $\alpha \cdot (1-f)^{\Delta+1} \geq (1+\epsilon)/2, R \geq 144\Delta/\epsilon\beta f, k > 192\Delta/(\epsilon\beta)$ and $R/6 \geq s = k/(4f) \geq 48\Delta/(\epsilon\beta f)$, then Ouroboros-Genesis achieves the following guarantees:*

- ***Common prefix.*** *The probability that* Ouroboros-Genesis *violates the common prefix property with parameter k is no more than* $\exp(-\Omega(k))$.
- ***Chain growth.*** *The probability that* Ouroboros-Genesis *violates the chain growth property with parameters $s \geq 48\Delta/(\epsilon\beta f)$ and $\tau_{\mathsf{CG}} = \beta f/16$ is no more than* $\exp(-\Omega(s))$.
- ***Existential chain quality.*** *The probability that* Ouroboros-Genesis *violates the existential chain quality property with parameter $s \geq 12\Delta/(\epsilon\beta f)$ is no more than* $\exp(-\Omega(s))$.
- ***Chain quality.*** *The probability that* Ouroboros-Genesis *violates the chain quality property with parameters $k \geq 48\Delta/(\epsilon\beta f)$ and $\mu = \epsilon\beta f/16$ is no more than* $\exp(-\Omega(k))$.

Given these properties, we show that profiles that honest parties produced in any interval of $R/6$ consecutive slots account for the majority of all profiles with timestamps in this interval, and they are well recorded in the blockchain.

Lemma 2. *Consider an execution of $\Pi_{\mathsf{ledger}}^{\mathsf{fair}++}$ over a lifetime of $L = \mathsf{poly}(\kappa)$ slots and any sequence of $R/6$ consecutive slots $S = (i, \ldots, i + R/6 - 1)$. If the properties in Theorem 1 are not violated during the execution, then for any honest party P at time $t > i + R$ holding a chain $\mathcal{C}$, the majority of the profiles with timestamp in S and recorded in $\mathcal{C}[i : i + 2R/3]$ are honestly generated.*

Proof. Due to the lower bound on alert stake ratio as in Theorem 1 and the fact that $R = \omega(\log \kappa)$ (thus we can apply a large convergence bound), at slot $i+R$, majority of the profiles of timestamp in S are generated by honest parties. In light of existential chain quality, all honest profiles generated in S are included in a block of the adopted chain $\mathcal{C}$ up to slot $[i, i + 2R/3]$. Then, at slot $i + R$, based on the chain growth property, for any chain $\mathcal{C}$ held by an honest party P, it contains as least $\tau_{\mathsf{CG}} \cdot R/3$ blocks in the last $R/3$ slots, and by the common prefix property, the chains adopted by honest parties are identical up to slot $i + 2R/3$, which also holds for all slots afterwards. □

Ephemeral committee security. In terms of the security of ephemeral committee for the i-th invocation of $\Pi_{\mathsf{ss\text{-}tx}}$, our simulation fails under the following circumstances: (i) $\mathcal{A}$ corrupts more than half of the ephemeral committee members before sending their key generation message; (ii) for an honestly generated ephemeral key, $\mathcal{A}$ (blindly) corrupts more than half of the secret-holding committee members before the designated time; (iii) by looking at the key generation message (which contains commitment and encryption of the secret shares), $\mathcal{A}$ learns the corresponding ephemeral secret key; and (iv) by looking at the transaction issuing message which consists of the encryption of transaction under another random secret s and the encryption of shares under the corresponding ephemeral public keys, $\mathcal{A}$ learns s and thus the transaction.

Consider item (i) in the abovementioned situations. Since committee membership election and profile production share the same VRF (except they evaluate different points), we have the following corollary from Lemma 2.

Corollary 3. *Consider an execution of $\Pi_{\mathsf{ledger}}^{\mathsf{fair}++}$ over a lifetime of $L = \mathsf{poly}(\kappa)$ slots and the i-th invocation of $\Pi_{\mathsf{ss\text{-}tx}}$ where parties self-elect as committee member during slots $((i-1) \cdot R/6, i \cdot R/6]$. If the properties in Theorem 1 are not violated during the execution, then for any honest party P at time $t > (i + 5) \cdot R/6$ holding a chain $\mathcal{C}$, the majority of the ephemeral public keys of the i-th committee recorded in $\mathcal{C}[(i-1) \cdot R/6 : (i + 3) \cdot R/6]$ are honestly generated.*

Regarding item (ii), since honest parties shares the ephemeral secret key to a randomly selected subset of parties of size $n = \omega(\log \kappa)$ and that majority of the shares are honest throughout the execution, large convergence bounds again applies so that honest majority is achieved (at the time of choosing these public keys) among the secret-holding committee members.

Now, consider item (iii), we rely on the security of anonPKE (refer to Theorem 7 in Appendix A.1).

Lemma 4. *Suppose that* anonPKE *is* KM-NCE-CCA *and unique recipient secure, then the adversary* $\mathcal{A}$ *can learn majority of the ephemeral secret keys with probability no more than* $\mathsf{negl}(\kappa)$.

For item (iv), since the first three items has ensured that $\mathcal{A}$ cannot learn more than half of the ephemeral secret keys before they are actually published, we prove the secrecy of a transaction via a reduction to the IND-CPA security game of the underlying PKE scheme. In more detail, we consider the following game between the challenger and the adversary.

Game 1 $\mathsf{Exp}_{\Pi_{\mathsf{ss\text{-}tx}},\mathcal{A}}(1^\kappa)$

1. The challenger maintains a set of parties and generates $(\mathsf{epk}_1, \ldots, \mathsf{epk}_k)$ by running the self-election and ephemeral key generation protocol for each party; when a party P is corrupted, this procedure on P is controlled by the adversary but subject to the constrain that he can corrupt up to a minority of the parties.
2. The adversary is given $(\mathsf{epk}_1, \ldots, \mathsf{epk}_k)$ and is allowed to adaptively corrupt up to a minority of the parties.
3. The adversary generates two equal-length transactions $\mathtt{tx}_0$ and $\mathtt{tx}_1$ and transmits them to the challenger.
4. The challenger randomly selects $b \xleftarrow{\$} \{0,1\}$ and encrypts $\mathtt{tx}_b$ using $(\mathsf{epk}_1, \ldots, \mathsf{epk}_k)$ as per Protocol 2 and returns the output to the adversary.
5. The adversary outputs a bit b' and wins if $b = b'$.

Lemma 5. *Suppose that* PKE *is IND-CPA secure, then the adversary cannot win Game 1 with probability more than* $1/2 + \mathsf{negl}(\kappa)$ *where* κ *is the security parameter.*

Proof. We show that if $\mathcal{A}$ in Game 1 wins with probability more than $1/2 + \mathsf{negl}(\kappa)$, then the challenger $\mathcal{C}$, acting as the adversary in the IND-CPA game of PKE and using $\mathcal{A}$ as a subroutine, wins with the same probability.

The reduction works as follows. After receiving $\{\mathsf{pk}_1, \ldots, \mathsf{pk}_N\}$ from the IND-CPA security game of PKE, $\mathcal{C}$ (using the code of simulator) runs the setup stage in Game 1 with $\mathcal{A}$. Importantly, when the i-th time that $\mathcal{C}$ is about to generate epk_i on behalf of an honest party as per Protocol 1, $\mathcal{C}$ directly gives out pk_i, and commits and encrypts to n randomly selected shares ($\mathcal{C}$ also fakes the NIZK proof). $\mathcal{C}$ then performs all other operations honestly. Note that since $\mathcal{A}$ will not be able to corrupt more than half of the parties due to Lemma 4, $\mathcal{A}$ cannot tell if epk_i is malformed or not.

Without loss of generality, we assume that $\mathsf{epk}_1, \ldots \mathsf{epk}_j$ is replaced with $\mathsf{pk}_1, \ldots \mathsf{pk}_j$ and epk_{j+1}, $\ldots, \mathsf{epk}_k$ are generated by $\mathcal{A}$.

Upon receiving two transactions $\mathtt{tx}_0$ and $\mathtt{tx}_1$ from $\mathcal{A}$, $\mathcal{C}$ picks uniformly random points $y_{0,0}, y_{0,1}$, $y_{k-t+1}, \ldots, y_k$ (it holds that $k-t+1 < j$ due to honest majority of committee members). Note that, $y_{0,0}, y_{k-t+1}, \ldots, y_k$ defines a unique t-degree polynomial f; and $y_{0,1}, y_{k-t+1}, \ldots, y_k$ defines another unique t-degree polynomial f'. $\mathcal{C}$ then computes $m_{0,i} = f(i)$ and $m_{1,i} = f'(i)$ for $i \in [1, k-t]$, and hands $(m_{0,1}, \ldots, m_{0,k-t}, y_{k-t+1}, \ldots, y_j), (m_{1,1}, \ldots, m_{1,k-t}, y_{k-t+1}, \ldots, y_j)$ to the challenger in the IND-CPA security game of PKE and ask for encrypting one vector under key $\mathsf{pk}_1, \ldots pk_j$ respectively.

Upon receiving response $(e_1, \ldots, e_j)$, $\mathcal{C}$ randomly samples $e \xleftarrow{\$} \{0,1\}^\kappa$ and publishes a transaction of form $(e, (e_1, e_2, \ldots, e_j, e_{j+1}, \ldots, e_k))$ where and $e_{j+1}, \ldots, e_k$ are encryptions of $y_{j+1}, \ldots, y_k$ under adversarially generated keys $\mathsf{epk}_{j+1}, \ldots, \mathsf{epk}_k$ respectively. $\mathcal{C}$ gives $(e, (e_1, e_2, \ldots, e_k))$ to $\mathcal{A}$ and outputs whatever $\mathcal{A}$ outputs. □

Theorem 6. *Let* k *be the common-prefix parameter,* R *the epoch-length parameter and* Δ *the network delay. let* τ_{CG} *and* μ *be the speed and chain-quality coefficients, respectively (both defined*

as in Theorem 1). Assuming that the IND-CPA security of PKE *and the hardness of computational Diffe-Hellman, the the protocol* $\Pi^{\mathsf{fair}++}_{\mathsf{ledger}}$ *realizes the ledger functionality* $\mathcal{G}^{\mathsf{Fair}++}_{\mathsf{Ledger}}$ *in the* $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}, \mathcal{G}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{F}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{INIT}}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}})$*-hybrid world. I.e., there exists a simulator that simulates the protocol execution in the ideal world perfectly except with negligible probability in the parameter* κ *when setting* $R \geq \omega(\log \kappa)$*, for the ledger parameters* $\mathtt{windowSize} = k, \mathtt{Delay} = R/6$, $\mathtt{waitTime} = R, \mathtt{maxTime}_{\mathtt{window}} = \mathtt{windowSize}/\tau_{\mathsf{CG}}, \mathtt{minTime}_{\mathtt{window}} = (1-\mu)\mathtt{windowSize}$.

Proof. We prove the theorem by providing a simulator $\mathcal{S}_{\text{ledger}}$ in the ideal world such that the protocol execution in the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}, \mathcal{G}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{F}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{INIT}}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}})$-hybrid world is indistinguishable from the ideal-world execution with the ledger functionality and the simulator. This simulation can be done perfectly, as the only events that prevent a successful simulation are those defined by $\mathsf{BAD\text{-}CP}$, $\mathsf{BAD\text{-}CQ}$, $\mathsf{BAD\text{-}CG}$, $\mathsf{BAD\text{-}PROFILE}$, $\mathsf{BAD\text{-}COMMITTEE}$ and $\mathsf{BAD\text{-}Program}$, which we have been proving in Theorem 1, Lemmas 2 and 5, and Corollary 3 that happens with negligible probability. □

Acknowledgements

Michele Ciampi was partially supported by the Input Output Research Hub (IORH) of the University of Edinburgh and by Sunday Group Inc. Yu Shen's research has been supported by Input Output (iohk.io) through their funding of the University of Edinburgh Blockchain Technology Lab.

References

[BCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, Berkeley, CA, USA, May 18–21, 2014. IEEE Computer Society Press. https://doi.org/10.1109/SP.2014.36.

[BCH+20] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part III*, volume 12552 of *Lecture Notes in Computer Science*, pages 1–30, Durham, NC, USA, November 16–19, 2020. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-64381-2_1.

[BGG+20] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 260–290, Durham, NC, USA, November 16–19, 2020. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-64375-1_10.

[BGK+18] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 913–930, Toronto, ON, Canada, October 15–19, 2018. ACM Press. https://doi.org/10.1145/3243734.3243848.

[BMTZ17] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Bitcoin as a transaction ledger: A composable treatment. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-63688-7_11.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000. https://doi.org/10.1007/s001459910006.

[CDEG23] James Hsin-yu Chiang, Bernardo David, Ittay Eyal, and Tiantian Gong. FairPoS: Input Fairness in Permissionless Consensus. In Joseph Bonneau and S. Matthew Weinberg, editors, *5th Conference on Advances in Financial Technologies (AFT 2023)*, volume 282 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:23, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.AFT.2023.10.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 280–312, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-78381-9_11.

[CDK+22] Matteo Campanelli, Bernardo David, Hamidreza Khoshakhlagh, Anders Konring, and Jesper Buus Nielsen. Encryption to the future - A paradigm for sending secret messages to future (anonymous) committees. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part III*, volume 13793 of *Lecture Notes in Computer Science*, pages 151–180, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-031-22969-5_6.

[CDPW07] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 61–85, Amsterdam, The Netherlands, February 21–24, 2007. Springer Berlin Heidelberg, Germany. https://doi.org/10.1007/978-3-540-70936-7_4.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541, Santa Barbara, CA, USA, August 19–23, 2001. Springer Berlin Heidelberg, Germany. https://doi.org/10.1007/3-540-44647-8_31.

[CKR+23] Ran Canetti, Sebastian Kolby, Divya Ravi, Eduardo Soria-Vazquez, and Sophia Yakoubov. Taming adaptivity in YOSO protocols: The modular way. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023: 21st Theory of Cryptography Conference, Part II*, volume 14370 of *Lecture Notes in Computer Science*, pages 33–62, Taipei, Taiwan, November 29 – December 2, 2023. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-031-48618-0_2.

[CKS24] Michele Ciampi, Aggelos Kiayias, and Yu Shen. Universal composable transaction serialization with order fairness. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024, Part II*, volume 14921 of *Lecture Notes in Computer Science*, pages 147–180, Santa Barbara, CA, USA, August 18–22, 2024. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-031-68379-4_5.

[CSW22] Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive UC NIZK. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 466–495, Taipei, Taiwan, December 5–9, 2022. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-031-22966-4_16.

[DGK+20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy*, pages 910–927, San Francisco, CA, USA, May 18–21, 2020. IEEE Computer Society Press. https://doi.org/10.1109/SP40000.2020.00040.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM*

Symposium on Theory of Computing, pages 467–476, Palo Alto, CA, USA, June 1–4, 2013. ACM Press. https://doi.org/10.1145/2488608.2488667.

[GHK+21] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part II*, volume 12826 of *Lecture Notes in Computer Science*, pages 64–93, Virtual Event, August 16–20, 2021. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-84245-1_3.

[GHM+21] Craig Gentry, Shai Halevi, Bernardo Magri, Jesper Buus Nielsen, and Sophia Yakoubov. Random-index PIR and applications. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 32–61, Raleigh, NC, USA, November 8–11, 2021. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-90456-2_2.

[KDL+23] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 475–489, Copenhagen, Denmark, November 26–30, 2023. ACM Press. https://doi.org/10.1145/3576915.3616658.

[KLS24] Aggelos Kiayias, Nikos Leonardos, and Yu Shen. Ordering transactions with bounded unfairness: Definitions, complexity and constructions. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part III*, volume 14653 of *Lecture Notes in Computer Science*, pages 34–63, Zurich, Switzerland, May 26–30, 2024. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-031-58734-4_2.

[KMTZ13] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013: 10th Theory of Cryptography Conference*, volume 7785 of *Lecture Notes in Computer Science*, pages 477–498, Tokyo, Japan, March 3–6, 2013. Springer Berlin Heidelberg, Germany. https://doi.org/10.1007/978-3-642-36594-2_27.

[Kur20] Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, pages 25–36, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3419614.3423263.

[KZGJ20] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part III*, volume 12172 of *Lecture Notes in Computer Science*, pages 451–480, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-030-56877-1_16.

[Ped92] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140, Santa Barbara, CA, USA, August 11–15, 1992. Springer Berlin Heidelberg, Germany. https://doi.org/10.1007/3-540-46766-1_9.

[RB94] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Trans. Program. Lang. Syst.*, 16(3):986–1009, 1994. https://doi.org/10.1145/177492.177745.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990. https://doi.org/10.1145/98163.98167.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979. https://doi.org/10.1145/359168.359176.

[ZSC+20] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without byzantine oligarchy. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, USA, 2020. USENIX Association.

A Preliminaries (Cont'd)

Global clock. We model synchronous parties using $\mathcal{G}_{\mathsf{Clock}}$ (cf. [KMTZ13]). At a high level, $\mathcal{G}_{\mathsf{Clock}}$ maintains a per-session round variable τ, which is forwarded only when all registered parties in that session has sent $\mathcal{G}_{\mathsf{Clock}}$ the CLOCK-UPDATE command. The current time can be checked by sending a CLOCK-READ command. Whenever a party who has finished the computation in a round is activated, she checks if the time from $\mathcal{G}_{\mathsf{Clock}}$ has been forwarded; if not, she does nothing and wait for the next activation.

Functionality $\mathcal{G}_{\mathsf{Clock}}$

This functionality maintains state variables as follows.

State Variable	Description
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties $\mathsf{P} = (\text{pid}, \text{sid})$.
$F \leftarrow \emptyset$	The set of registered functionalities (together with their session identifier).
$\tau_{\text{sid}} \leftarrow 0$	The clock variable for session sid.
$d_{\mathsf{P}} \leftarrow 0$	The clock-update variable for $\mathsf{P} = (\text{pid}, \text{sid}) \in \mathcal{P}$.
$d(\mathcal{F}, \text{sid}) \leftarrow 0$	The clock-update variable for $(\mathcal{F}, \text{sid}) \in F$.

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

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

Functionality 3: *The global clock.*

(Local) Diffuse functionalities. We model the Δ-bounded delay network with $\mathcal{F}_{\mathsf{Diffuse}}$ for all protocol-related messages like blocks, ephemeral public keys and private key shares (except transactions that are handled by $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ in Section 4.1). Note that once a corrupted message reaches at least one honest party at round r, $\mathcal{F}_{\mathsf{Diffuse}}$ guarantees to deliver this message to all honest parties before round $r + \Delta$ (i.e., honest parties keep "echoing" messages). By convention, different types of messages are diffused by different functionalities, and we write $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{cmte}}_{\mathsf{Diffuse}}$ to denote the network for chains, and ephemeral keys as stated in Section 5.2.

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

The functionality is parameterized by the network delay Δ.

State Variable	Description
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties.
$\vec{M} \leftarrow []$	A dynamically updatable list of quadruples $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ where D_{mid} denotes the fetch counter.

Network capabilities:

- Upon receiving $(\text{DIFFUSE}, \text{sid}, m)$ from some $\mathsf{P}_s \in \mathcal{P}$, where $\mathcal{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ denotes the current party set, do:
  1. Choose n new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$.
  2. Initialize $2n$ new variables $D_{\mathsf{mid}_1} := D^{MAX}_{\mathsf{mid}_1} \ldots := D_{\mathsf{mid}_n} := D^{MAX}_{\mathsf{mid}_n} := 1$ and a per message-delay $\Delta_{\mathsf{mid}_i} = \Delta$ for $i \in [n]$.
  3. Set $\vec{M} := \vec{M} \parallel (m, \mathsf{mid}_1, D_{\mathsf{mid}_1}, \mathsf{P}_1) \parallel \ldots \parallel (m, \mathsf{mid}_n, D_{\mathsf{mid}_n}, \mathsf{P}_n)$.
  4. Send $(\text{DIFFUSE}, \text{sid}, m, \mathsf{P}_s, (\mathsf{P}_1, \mathsf{mid}_1), \ldots, (\mathsf{P}_n, \mathsf{mid}_n))$ to the adversary.
- Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathsf{P} \in \mathcal{P}$, or from $\mathcal{A}$ on behalf of a corrupted party P:
  1. For all tuples $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P}) \in \vec{M}$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.
  2. Let $\vec{M}^{\mathsf{P}}_0$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ with $D_{\mathsf{mid}} \leq 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}^{\mathsf{P}}_0$ from $\vec{M}$ and in case some $(m, \mathsf{mid}, D_{\mathsf{mid}}, \mathsf{P})$ is in $\vec{M}^{\mathsf{P}}_0$, where P is honest, set $\Delta_{\mathsf{mid}'} = \Delta$ for any $(m, \mathsf{mid}', D_{\mathsf{mid}'}, (\cdot, \text{sid}))$ in $\vec{M}$ and replace this record by $(m, \mathsf{mid}', \min\{D_{\mathsf{mid}'}, \Delta\}, \mathsf{P}')$.
  3. Output $\vec{M}^{\mathsf{P}}_0$ to P (if P is corrupted, send $\vec{M}^{\mathsf{P}}_0$ to $\mathcal{A}$).

Additional adversarial capabilities:

- Upon receiving $(\text{DIFFUSE}, \text{sid}, m)$ from some corrupted $\mathsf{P}_s \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P_s if corrupted), do execute it the same way as an honest-sender diffuse, with the only difference that $\Delta_{\mathsf{mid}_i} = \infty$.
- Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\mathsf{mid}_{i_j}}, \mathsf{mid}_{i_j})$: if $D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}} \leq \Delta_{\mathsf{mid}_{i_j}}$ and mid_{i_j} is a message-ID of receiver $\mathsf{P} = (\cdot, \text{sid})$ registered in the current $\vec{M}$, set $D_{\mathsf{mid}_{i_j}} := D_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$ and set $D^{MAX}_{\mathsf{mid}_{i_j}} := D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$; otherwise, ignore this pair.
- Upon receiving $(\text{SWAP}, \text{sid}, \mathsf{mid}, \mathsf{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \mathsf{mid}, D_{\mathsf{mid}}, (\cdot, \text{sid}))$ and $(m, \mathsf{mid}', D_{\mathsf{mid}'}, (\cdot, \text{sid}))$ in $\vec{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.
- Upon receiving $(\text{GET-REG}, \text{sid})$ from $\mathcal{A}$, return the response $(\text{GET-REG}, \text{sid}, \mathcal{P})$ to $\mathcal{A}$.

Functionality 4: *The (local) diffusion network.*

Non-interactive zero knowledge scheme. The NIZK scheme that we use in this paper are from [CSW22], secure against an adaptive adversary in the CRS model.

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

This functionality is parametrized by an NP relation R and running with parties $\mathsf{P}_1, \ldots, \mathsf{P}$ and adversary $\mathcal{S}$.

- **Proof:** On input $(\text{PROVE}, \text{sid}, x, w, \mathsf{P})$ from party P: If $\mathsf{r}(x, w) \neq 1$ then ignore the input. Send $(\text{PROVE}, \text{sid}, x)$ to $\mathcal{S}$. Upon receiving the answer $(\text{PROOF}, \text{sid}, \pi)$ from $\mathcal{S}$, store (sid, x, w, π) and send $(\text{PROOF}, \text{sid}, \pi)$ to P.
- **Verification:** On input $(\text{VERIFY}, \text{sid}, x, \pi)$ from a party V: If (sid, x, w, π) is stored, then return $(\text{VERIFICATION}, \text{sid}, x, \pi, \mathsf{R}(x, w))$ to V. Else, send $(\text{VERIFY}, V, \text{sid}, x, \pi)$ to $\mathcal{S}$. Upon receiving $(\text{WITNESS}, \text{sid}, w)$ from $\mathcal{S}$, store (sid, x, w, π), and return $(\text{VERIFICATION}, \text{sid}, x, \pi, \mathsf{R}(x, w))$ to V.

Functionality 5: *The adaptively-secure non-interactive zero-knowledge proof.*

The common reference string. Our commitment scheme relies on a common reference string which we model as $\mathcal{F}_{\mathsf{CRS}}$.

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

- **Retrieve.** Upon receiving $(\text{RETRIEVE}, \text{sid})$ from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), do the following:
  1. If activated for the first time, choose a value $g, h_1, h_2, h_3 \xleftarrow{\$} \mathbb{Z}_q$.
  2. Return $(\text{RETRIEVE}, (g, h_1, h_2, h_3))$ to P.

Functionality 6: *The common reference string.*

A.1 Adaptively Anonymous Public-Key Encryption

Global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$. The restricted programmable and observable global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ follows the modelling in [CDG+18]. We allow the adversary to observe and program the GRO; meantime, every party (in the same session) can check if a point is programmed by calling the "IsProgrammed" interface.

Functionality $\mathcal{G}_{\mathsf{rpoRO}}$

The functionality is parameterized by the security parameter κ. It maintains a dynamically updatable list H and prog Initially, $\mathsf{H} = \mathsf{prog} = \emptyset$.

- **Eval.** Upon receiving (EVAL, m) from a party (P, sid) or from the adversary, do the following:
  1. If $\mathsf{H}[m] = \perp$, sample a value h uniformly at random from $\{0,1\}^\kappa$ and set $\mathsf{H}[m] \leftarrow h$.
  2. Parse m as (s, m').
  3. If the query is made by the adversary, or if $s \neq \text{sid}$, then add (s, m', h) to the (initailly empty) list of illegitimate queries $\mathcal{Q}_s$.
  4. Output (EVAL, m, h) to the requestor.
- **Observe.** Upon receiving $(\text{OBSERVE}, \text{sid})$ from the adversary: If $\mathcal{Q}_{\text{sid}}$ does not exist, set $\mathcal{Q}_{\text{sid}} = \emptyset$. Output $(\text{OBSERVE}, \mathcal{Q}_{\text{sid}})$ to the adversary.
- **Program.** Upon receiving $(\text{PROGRAM}, m, h)$ with $h \in \{0,1\}^\kappa$ from the adversary, do the following:
  1. If $\exists h' \in \{0,1\}^\kappa$ such that $\mathsf{H}[m] = h'$ and $h \neq h'$, ignore this input.
  2. Set $\mathsf{H}[m] \leftarrow h$ and $\mathsf{prog} \leftarrow \mathsf{prog} \cup \{m\}$.
  3. Output $(\text{PROGRAM}, \mathsf{ok})$ to the adversary.
- **IsProgrammed.** Upon receiving $(\text{IS-PROGRAMMED}, m)$ from a party (P, sid) or from the adver-

sary, do the following:
1. If the input was given by (P, sid), parse m as (s, m'). If $s \neq \text{sid}$, ignore this input.
2. Set $b \leftarrow m \in \mathsf{prog}$ and output (IS-PROGRAMMED, b) to the requestor.

Functionality 7: *The restricted global random oracle with programmability and observability.*

Adaptively anonymous public-key encryption scheme. We will use the unique-recipient key-and-message non-committing encryption from [CKR+23] to achieve anonymity of public keys used to encrypt against selective opening attacks.

Algorithm anonPKE

Setup: A cyclic group $\mathbb{G}$ of order q, where q is a κ-bit prime, and let g be a generator of $\mathbb{G}$. Let the message space of the encryption scheme be $\{0,1\}^\kappa$.

$\underline{\mathsf{KeyGen}(1^\kappa)}$
1: Sample $\mathsf{sk} \xleftarrow{\$} \mathbb{Z}_q$ and compute $\mathsf{pk} \leftarrow g^{\mathsf{sk}}$
2: **return** $(\mathsf{pk}, \mathsf{sk})$

$\underline{\mathsf{Enc}(\mathsf{pk}, m)}$
3: Sample $r \xleftarrow{\$} \mathbb{Z}_q$ and compute $\beta \leftarrow g^r$
4: Compute $k \leftarrow \mathcal{G}_{\mathsf{rpoRO}}(\mathsf{pk}^r)$
5: Compute a MAC $d \leftarrow \mathcal{G}_{\mathsf{rpoRO}}(r, m)$
6: Set $e \leftarrow k \oplus (r, m)$
7: **return** (β, e, d)

$\underline{\mathsf{Dec}(\mathsf{sk}, c)}$
8: Parse $c = (\beta, e, d)$
9: Set $k' \leftarrow \mathcal{G}_{\mathsf{rpoRO}}(\beta^{\mathsf{sk}})$
10: Compute $(r', m') \leftarrow e \oplus k'$
11: **if** $g^{r'} = \beta \wedge d = \mathcal{G}_{\mathsf{rpoRO}}(r', m')$ **then return** m' **else return** $\perp$

$\underline{\mathsf{Fake}}$
12: Sample $r \xleftarrow{\$} \mathbb{Z}_q$ and compute $\beta \leftarrow g^r$
13: Set $\tau \leftarrow r$ and sample $c, d \xleftarrow{\$} \{0,1\}^{2\kappa}$
14: **return** $(c = (\beta, e, d), \tau)$

$\underline{\mathsf{Open}_k(\mathsf{pk}, \mathsf{sk}, (c^*_\gamma, \tau^*_\gamma, m^*_\gamma)_{\gamma \in [k]})}$
15: **for each** $(c^*_\gamma = (\beta, e, d), \tau^*_\gamma, m^*_\gamma)$ **do**
16: Set $r \leftarrow \tau^*_\gamma$
17: Program $\mathcal{G}_{\mathsf{rpoRO}}$ s.t. $\mathcal{G}_{\mathsf{rpoRO}}(r, m^*_\gamma) = d$ and $\mathcal{G}_{\mathsf{rpoRO}}(\mathsf{pk}^r) = e \oplus (r, m^*_\gamma)$
18: **end for**
19: **return** sk

Algorithm 6: *The adaptively anonymous public-key encryption scheme.*

Recall two properties, namely security and unique recipient, of the key-and-message non-committing encryption scheme from [CKR+23].

Experiment KM-NCE-CCA

$\underline{\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}CCA\text{-}real}}_{\mathsf{KM\text{-}NCE},\mathcal{A},k}(\kappa)}$

$(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\kappa)$
$((m^*_\gamma)_{\gamma\in[k]},\mathsf{state}_1) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{Dec}}}(\mathsf{pk})$
$(c^*_\gamma \xleftarrow{\$} \mathsf{Enc}(\mathsf{pk},m^*_\gamma))_{\gamma\in[k]}$
$\mathsf{state}_2 \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{Dec}}}((c^*_\gamma)_{\gamma\in[k]},\mathsf{state}_1)$
$\mathsf{state}_3 \leftarrow \mathcal{A}_3(\mathsf{sk},\mathsf{state}_2)$
return b'

$\underline{\mathcal{O}_{\mathsf{Dec}}(c)}$
if $c \in \{c^*_\gamma : \gamma \in [k]\}$ **then return** $\perp$
return $\mathsf{Dec}(\mathsf{sk},c)$

$\underline{\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}CCA\text{-}ideal}}_{\mathsf{KM\text{-}NCE},\mathcal{A},k}(\kappa)}$

$(\mathsf{pk},\mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen}(1^\kappa)$
$((m^*_\gamma)_{\gamma\in[k]},\mathsf{state}_1) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{Dec}}}(\mathsf{pk})$
$((c^*_\gamma,\tau^*_\gamma) \xleftarrow{\$} \mathsf{Fake})_{\gamma\in[k]}$
$\mathsf{state}_2 \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{Dec}}}((c^*_\gamma)_{\gamma\in[k]},\mathsf{state}_1)$
$\mathsf{sk}' \leftarrow \mathsf{Open}_k(\mathsf{pk},\mathsf{sk},(c^*_\gamma,\tau^*_\gamma,m^*_\gamma)_{\gamma\in[k]})$
$\mathsf{state}_3 \leftarrow \mathcal{A}_3(\mathsf{sk}',\mathsf{state}_2)$
return b'

Experiment 1: *The experiments for* KM-NCE-CCA *security.*

Definition 1 (Security). *A KM-NCE scheme* $\mathsf{KM\text{-}NCE} = (\mathsf{Setup},\mathsf{KeyGen},\mathsf{Enc},\mathsf{Dec},\mathsf{Fake},\mathsf{Open}_k)$ *in the k-challenge setting is (*CCA-*)*secure *if for any PPT adversary* $\mathcal{A} = (\mathcal{A}_1,\mathcal{A}_2,\mathcal{A}_3)$*, the advantage*

$$\mathbf{Adv}^{\mathsf{AC\text{-}RSO}_{\mathrm{k,Dist}}\&\mathsf{C\text{-}CPA}}_{\mathsf{PKE},\mathcal{A},\mathcal{S},\mathcal{D},n,t,k}(\kappa) \triangleq \left|\mathbf{Pr}[\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}CCA\text{-}real}}_{\mathsf{KM\text{-}NCE},\mathcal{A},k}(\kappa)=1] - \mathbf{Pr}[\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}CCA\text{-}ideal}}_{\mathsf{KM\text{-}NCE},\mathcal{A},k}(\kappa)=1]\right|$$

is negligible, where $\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}CCA\text{-}real}}_{\mathsf{KM\text{-}NCE},\mathcal{A},k}$ *and* $\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}CCA\text{-}ideal}}_{\mathsf{KM\text{-}NCE},\mathcal{A},k}$ *are defined in Experiment 1.*

Experiment KM-NCE-UR

$\underline{\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}UR}}_{\mathsf{KM\text{-}NCE},\mathcal{A}}(\kappa)}$

$((\mathsf{pk}_i,\mathsf{sk}_i) \xleftarrow{\$} \mathsf{KeyGen}(1^\kappa))_{i\in[h]}$
$c \xleftarrow{\$} \mathcal{A}_1^{\mathcal{O}_{\mathsf{Dec}},\mathcal{G}_{\mathsf{rpoRO}}}(\{\mathsf{pk}_i\}_{i\in[h]})$
Set $b \leftarrow (\exists i,j \in [h] : i \neq j \wedge \mathsf{Dec}(\mathsf{sk}_i,c) \neq \perp \wedge \mathsf{Dec}(\mathsf{sk}_j,c) \neq \perp)$
return b

$\underline{\mathcal{O}_{\mathsf{Dec}}(c)}$
return $\mathsf{Dec}(\mathsf{sk},c)$ ◁ possibly $\perp$

Experiment 2: *The experiments for* KM-NCE-UR *security.*

Definition 2 (Unique Recepient). *A KM-NCE scheme* $\mathsf{KM\text{-}NCE} = (\mathsf{Setup},\mathsf{KeyGen},\mathsf{Enc},\mathsf{Dec},\mathsf{Fake},\mathsf{Open}_k)$ *is* unique recipient *if for any PPT adversary* $\mathcal{A}$*,* $\mathbf{Pr}[\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}UR}}_{\mathsf{KM\text{-}NCE},\mathcal{A}}(\kappa)=1]$ *is negligible, where* $\mathbf{Exp}^{\mathsf{KM\text{-}NCE\text{-}UR}}_{\mathsf{KM\text{-}NCE},\mathcal{A}}$ *is defined in Experiment 2.*

Theorem 7. [CKR+23, Theorem 1] *The construction of* anonPKE *in Algorithm 6 is* $\mathsf{KM\text{-}NCE}_k$*-CCA and unique recipient secure, in the pROM under the CDH assumption in group* $\mathbb{G}$*.*

A.2 Hybrid Functionalities for PoS

We present hybrid functionalities for a PoS blockchain (refer to [BGK+18] for more detailed discussions). Specifically, this includes the global random oracle $\mathcal{G}_{\mathsf{RO}}$, the genesis block creation and distribution procedure $\mathcal{F}_{\mathsf{INIT}}$, the verifiable random functions $\mathcal{F}_{\mathsf{VRF}}$, and the key-evolving signatures scheme $\mathcal{F}_{\mathsf{KES}}$.

Functionality $\mathcal{G}_{\mathsf{RO}}$

The functionality is parameterized by a security parameter κ.

State Variable	Description
$\mathcal{P} \leftarrow \emptyset$	The set of registered parties.
$H \leftarrow \emptyset$	A dynamically updatable function table where $H[x] = \perp$ denotes the fact that no pair of the form $(x, \cdot)$ is in H.

- **Eval.** Upon receiving $(\text{EVAL}, \text{sid}_{RO}, x)$ from some party $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted P), do the following:
  1. If $H[x] = \perp$ sample a value y uniformly at random from $\{0,1\}^\kappa$ and set $H[x] \leftarrow y$.
  2. Return $(\text{EVAL}, \text{sid}, x, H[x])$ to the requestor.

Functionality 8: *The global random oracle.*

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

The functionality $\mathcal{F}_{\mathsf{INIT}}$ is parameterized by the set $\mathsf{P}_1, \ldots, \mathsf{P}_n$ of initial stakeholders n and their respective stakes $s_1, \ldots, s_n$. It maintains the set of registered parties $\mathcal{P}$.
Upon receiving any message from a party, the functionality first sends $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ to the clock to receive the current round. Subsequently:

1. If this is the first (genesis) round and the message is request from some initial stakeholder P_i of the form $(\mathsf{ver_keys}, \text{sid}, \mathsf{P}_1, v_i^{\text{vrf}}, v_i^{\text{kes}})$, then $\mathcal{F}_{\mathsf{INIT}}$ stores the verification keys tuple $(\mathsf{P}_i, v_i^{\text{vrf}}, v_i^{\text{kes}})$ and acknowledges its receipt. If some of the registered public keys are equal, it outputs an error and halts. Otherwise, it samples and stores a random value $\eta_1 \xleftarrow{\$} \{0,1\}^\kappa$ and constructs a genesis block $(\mathbb{S}_1, \eta_1)$, where $\mathbb{S}_1 = \big((\mathsf{P}_1, v_1^{\text{vrf}}, v_1^{\text{kes}}, s_1), \ldots, (\mathsf{P}_1, v_n^{\text{vrf}}, v_n^{\text{kes}}, s_n)\big)$
2. If this is not the first round, then do the following. If any of the n initial stakeholders has not send a request of the above form, i.e., a $(\mathsf{ver_keys}, \text{sid}, \mathsf{P}_1, v_i^{\text{vrf}}, v_i^{\text{kes}})$-message, to $\mathcal{F}_{\mathsf{INIT}}$ in the genesis round then $\mathcal{F}_{\mathsf{INIT}}$ outputs an error and halts. Otherwise, if the currently received input is a request of the form $(\mathsf{genblock_req}, \text{sid}, \mathsf{P}_i)$ from any (initial or not) stakeholder P, $\mathcal{F}_{\mathsf{INIT}}$ sends $(\mathsf{genblock}, \text{sid}, (\mathbb{S}_1, \eta_1))$ to the requester.

Functionality 9: *The genesis block creattion and stake distribution.*

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

$\mathcal{F}_{\mathsf{VRF}}$ interacts with its set of registered parties $\mathcal{P}$ as follows.

- **Key generation.** Upon receiving a message $(\mathsf{KeyGen}, \text{sid})$ from a stakeholder P_i, hand $(\mathsf{KeyGen}, \text{sid}, \mathsf{P}_i)$ to the adversary. Upon receiving $(\mathsf{VerificationKey}, \text{sid}, \mathsf{P}_i, v)$ from the adversary, if P_i is honest, verify that v is unique, record the pair (P_i, v) and return $(\mathsf{VerificationKey}, \text{sid}, v)$ to P_i. Initialize the table $T(v, \cdot)$ to empty.
- **Malicious key generation.** Upon receiving a message $(\mathsf{KeyGen}, \text{sid}, v)$ from $\mathcal{S}$, verify that v has not being recorded before; in this case initialize table $T(v, \cdot)$ to empty and record the pair $(\mathcal{S}, v)$.
- **VRF evaluation.** Upon receiving a message $(\mathsf{Eval}, \text{sid}, m)$ from P_i, verify that some pair (P_i, v) is recorded. If not, then ignore the request. Then, if the value $T(v, m)$ is undefined, pick a random

value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v,m) = (y, \emptyset)$. Then output $(\mathsf{Evaluated}, \mathrm{sid}, y)$ to P_i, where y is such that $T(v,m) = (y,S)$ for some S.

- **VRF evaluation and proof.** Upon receiving a message $(\mathsf{EvalProve}, \mathrm{sid}, m)$ from P_i, verify that some pair (P_i, v) is recorded. If not, then ignore the request. Else, send $(\mathsf{EvalProve}, \mathrm{sid}, \mathsf{P}_i, m)$ to the adversary. Upon receiving $(\mathsf{EvalProve}, \mathrm{sid}, m, \pi)$ from the adversary, if value $T(v,m)$ is undefined, verify that π is unique, pick a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v,m) = (y, \{\pi\})$. Else, if $T(v,m) = (y,S)$, set $T(v,m) = (y, S \cup \{\pi\})$. In any case, output $(\mathsf{Evaluated}, \mathrm{sid}, y, \pi)$ to P_i.
- **Malicious VRF evaluation.** Upon receiving a message $(\mathsf{Eval}, \mathrm{sid}, v, m, \pi)$ from $\mathcal{S}$ for some v, do the following. First, if (S, v) is recorded and $T(v,m)$ is undefined, then choose a random value y from $\{0,1\}^{\ell_{\mathsf{VRF}}}$ and set $T(v,m) = (y,S)$ and output $(\mathsf{Evaluated}, \mathrm{sid}, y)$ to $\mathcal{S}$. The same is performed in case (P_i, v) is recorded and P_i corrupted. Else, if $T(v,m) = (y,S')$ for some $S' \neq \emptyset$, union S to S' and output $(\mathsf{Evaluated}, \mathrm{sid}, y)$ to $\mathcal{S}$, else ignore the request.
- **Verification.** Upon receiving a message $(\mathsf{Verify}, \mathrm{sid}, m, y, \pi, v')$ from some party P, send $(\mathsf{Verify}, \mathrm{sid}, m, y, \pi, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, \mathrm{sid}, m, y, \pi, v')$ from the adversary do:
  1. If $v' = v$ for some $(\cdot, v)$ and the entry $T(v,m)$ equals (y,S) with $\pi \in S$, then set $f = 1$.
  2. Else, if $v' = v$ for some recorded pair of the form $(\cdot, v)$, but no entry $T(v,m)$ of the form $(y, \{\ldots, \pi, \ldots\})$ is recorded, then set $f = 0$.
  3. Else, initialize the table $T(v', \cdot)$ to empty, and set $f = 0$. Output $(\mathsf{Verified}, \mathrm{sid}, m, y, \pi, f)$ to P.

Functionality 10: *The verifiable random functions.*

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

$\mathcal{F}_{\mathsf{KES}}$ is parameterized by the total number of signature updates T, interacting with a signer P_S and registered parties in $\mathcal{P}$ as follows:

- **Key generation.** Upon receiving a message $(\mathsf{KeyGen}, \mathrm{sid}, \mathsf{P}_S)$ from a stakeholder P_S, send $(\mathsf{KeyGen}, \mathrm{sid}, \mathsf{P}_S)$ to the adversary. Upon receiving $(\mathsf{VerificationKey}, \mathrm{sid}, \mathsf{P}_S, v)$ from the adversary, send $(\mathsf{VerificationKey}, \mathrm{sid}, v)$ to P_S, record the triple $(\mathrm{sid}, \mathsf{P}_S, v)$ and set counter $k_{\mathrm{ctr}} = 1$.
- **Sign and Update.** Upon receiving a message $(\mathsf{USign}, \mathrm{sid}, \mathsf{P}_S, m, j)$ from US, verify that $(\mathrm{sid}, \mathsf{P}_S, v)$ is recorded for some sid and that $k_{\mathrm{ctr}} \leq j \leq T$. If not, then ignore the request. Else, set $k_{\mathrm{ctr}} = j + 1$ and send $(\mathsf{Respond}, (\mathsf{Sign}, \mathrm{sid}, \mathsf{P}_S, m, j))$ to the adversary. Upon receiving $(\mathsf{Signature}, \mathrm{sid}, \mathsf{P}_S, m, j, \sigma)$ from the adversary, verify that no entry $(m, j, \sigma, v, 0)$ is recorded. If it is, then output an error message to P_S and halt. Else, send $(\mathsf{Signature}, \mathrm{sid}, m, j, \sigma)$ to P_S, and record the entry $(m, j, \sigma, v, 1)$.
- **Signature verification.** Upon receiving a message $(\mathsf{Verify}, \mathrm{sid}, m, j, \sigma, v')$ from some stakeholder P_i do:
  1. If $v' = v$ and the entry $(m, j, \sigma, v, 1)$ is recorded, then set $f = 1$. (This condition guarantees completeness: If the verification key v' is the registered one and σ is a legitimately generated signature for m, then the verification succeeds.)
  2. Else, if $v' = v$, the signer is not corrupted, and no entry $(m, j, \sigma', v, 1)$ for any σ' is recorded, then set $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. (This condition guarantees unforgeability: If v' is the registered one, the signer is not corrupted, and never signed m, then the verification fails.)
  3. Else, if there is an entry (m, j, σ, v', f') recorded, then let $f = f'$. (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
  4. Else, if $j < k_{\mathrm{ctr}}$, let $f = 0$ and record the entry $(m, j, \sigma, v, 0)$. Otherwise, if $j = k_{\mathrm{ctr}}$, hand $(\mathsf{Verify}, \mathrm{sid}, m, j, \sigma, v')$ to the adversary. Upon receiving $(\mathsf{Verified}, \mathrm{sid}, m, j, \phi)$ from the adversary

let $f = \phi$ and record the entry (m, j, σ, v', ϕ). (This condition guarantees that the adversary is only able to forge signatures under keys belonging to corrupted parties for time periods corresponding to the current or future slots.)

Output $(\mathsf{Verified}, \mathrm{sid}, m, j, f)$ to P_i.

Functionality 11: *The key-evolving signatures.*

B Glossary

Protocol parameters. In Table 1, we list all the protocol parameters and their explanation.

Variable	Description
κ	Security parameter; length of the random oracle output.
Δ	Upper bound on message delay in slots.
f	Active slots coefficient.
$\phi(\cdot)$	Slot-leader probability function.
R	Epoch length in slots.
n	Size of secret-holding committee; $n = \omega(\log \kappa)$.
F	The transaction serialization function.

Table 1: *Overview of protocol* $\Pi^{\mathsf{fair++}}_{\mathsf{ledger}}$ *parameters.*

Main state variables of $\Pi^{\mathsf{fair++}}_{\mathsf{ledger}}$ participants. We summarize the internal state variables of a protocol participant as follows.

Variable	Description
$\tau, \mathtt{sl}$	The party's current time-stamp (in time-slots).
$\mathtt{ep}$	The epoch that $\mathtt{sl}$ belongs to.
$\mathcal{C}_{\mathsf{loc}}$	The local chain the party adopts for staking and exporting ledger state.
$\mathtt{isInit}$	A variable indicating whether party has completed the initialization.
$t_{\mathsf{on}}, t_{\mathsf{work}}$	Values to steer when the party executes the fetching or staking procedures respectively.
$\mathtt{buffer}_{\mathsf{tx}}$	The buffer of transactions.
$\mathtt{buffer}_{\mathsf{pb}}$	The buffer of profiles.
$\mathtt{buffer}_{\mathsf{cmte}}$	The buffer of ephemeral keys.
$T^{\mathsf{ep}}_{\mathsf{P}}$	The threshold of party P to evaluate slot leadership in (current) epoch $\mathtt{ep}$.

$T^{\mathsf{ep,cmte}}$	The threshold of party P to evaluate ephemeral committeee membership as well as profile production in (current) epoch $\mathtt{ep}$.
$v_{\mathsf{P}}^{\mathrm{vrf}}$, $v_{\mathsf{P}}^{\mathrm{kes}}$	The public keys of this party to interact with $\mathcal{F}_{\mathsf{VRF}}$ and $\mathcal{F}_{\mathsf{KES}}$.
`accounts`	A mapping from account name acc to a list of accounts of private keys and randomness used.

Table 2: *Overview of internal variables of protocol* $\Pi_{\mathsf{ledger}}^{\mathsf{fair++}}$ *participants.*