Metadata of the chapter that will be visualized in SpringerLink

Book Title	Advances in Cryptology – ASIACRYPT 2025	
Series Title		
Chapter Title	Universally Composable Transaction Order Fairness: Refined Definitions and Adaptive Security	
Copyright Year	2026	
Copyright HolderName	The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd.	
Author	Family Name	**Ciampi**
	Particle	
	Given Name	**Michele**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	University of Edinburgh
	Address	Edinburgh, UK
	Email	michele.ciampi@ed.ac.uk
	ORCID	http://orcid.org/0000-0001-5062-0388
Author	Family Name	**Kiayias**
	Particle	
	Given Name	**Aggelos**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	University of Edinburgh
	Address	Edinburgh, UK
	Division	
	Organization	Input Output
	Address	Edinburgh, UK
	Email	aggelos.kiayias@ed.ac.uk
	ORCID	http://orcid.org/0000-0003-2451-1430
Corresponding Author	Family Name	**Shen**
	Particle	
	Given Name	**Yu**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	University of Edinburgh
	Address	Edinburgh, UK
	Email	yu.shen@ed.ac.uk

ORCID http://orcid.org/0000-0002-4316-8116

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.

Universally Composable Transaction Order Fairness: Refined Definitions and Adaptive Security

Michele Ciampi[1], Aggelos Kiayias[1,2], and Yu Shen[1(✉)]

[1] University of Edinburgh, Edinburgh, UK
{michele.ciampi,aggelos.kiayias,yu.shen}@ed.ac.uk
[2] Input Output, Edinburgh, UK

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.

1 Introduction

Ledger consensus, a special case of state machine replication [28], 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 [23,25] a third property was highlighted, *order fairness* referring

The full version of this paper is available at [16].

G. Hanaoka and B.-Y. Yang (Eds.): ASIACRYPT 2025, LNCS 16246, pp. 1–33, 2025.
https://doi.org/10.1007/978-981-95-5096-8_10

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. [17].

There are two dimensions to order fairness: one is temporal and the other is causal. The temporal aspect, cf. [23,25], at a high level mandates that "earlier" transactions should be serialized first. The causal aspect on the other hand, cf. [7,27], 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. [15].

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 [15,25]. 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 [22–24] 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 [23] 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 [10].

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 [11] 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 Sect. 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 [15]) 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 [19].

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, [15] is the only work that captures and implements transaction order fairness in the UC setting. In [15], 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 [15] 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 [18], 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 [9], 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 [9] 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 [6,19,20]. Moreover, [6,19] 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 [14] 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 [15]. 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 [15] 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 [15] with the following main differences: (i) our protocol is presented in the Proof-of-Stake setting ([15] is Proof-of-Work) and (ii) capturing receiver-order fairness requires building a proper transaction graph as in [24]. 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 [15], 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 [6,9,19,20] 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.

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

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 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].

[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.

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 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,

[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.

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 [13] 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].

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 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 employ-

[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.

ing an account-based protocol inspired by Zerocash [5], 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 [10]. 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 [21]: 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 [4], 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 Sect. 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 the full version [16].

Global Random Oracles. Our constructions and proofs are based on different kinds of global random oracle functionalities (see [16]). 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 [8] 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. As an alternative approach for the NIZK, we could also rely on a global random oracle that cannot be programmed following the approach proposed in [1].

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 [24] and we refer the reader to [24] 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 [25,30] which defines order based on the local timestamps (or indices) they are assigned by protocol participants; and the "local-order" variant [23,24] 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 [26] 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 [29]. 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.* [12] and denote this scheme as $\mathsf{anonPKE}$. We present $\mathsf{anonPKE}$ and its related security properties in [16].

Adaptive UC NIZK. We model the NIZK proof system that we will use as $\mathcal{F}_{\mathsf{NIZK}}$. For our construction we use an adaptively secure NIZK proof system due to Canetti, Sarkar and Wang [13], which provides full adaptive soundness, as well as adaptive zero-knowledge in the CRS model. Refer to the full version [16] for more details.

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 [3], which achieves composable

security with dynamic availability against adaptive corruption. We defer the details of this protocol to Sect. 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 the full version [16].

4 Extended Ledger Functionality

We extend the fair-order ledger functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ in [15] towards capturing receiver order fairness and a more refined notion of input causality. We denote our new ledger functionality as $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$, and our adaption consists of two aspects. On one hand, in Sect. 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 $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$. 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 Sect. 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 Sect. 4.3, we instantiate two receiver order fairness policies—order linearizability [30] and bounded unfairness [24].

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 $\mathcal{G}_{\mathsf{Ledger}}$ [4] and its extension in $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ [15]. In [4], 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 [15], 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 [15] 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 $\mathtt{tx}$ is submitted in the clear (i.e., causality is not of concern), $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ processes $\mathtt{tx}$ 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 [2] for more details).

[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.

Codewise, $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ maintains a set of registered ledger functionalities with their session id $\{(\mathcal{F}, \mathsf{sid})\}$. 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 $\mathsf{P} = (\cdot, \mathsf{sid})$, $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ leaks them to all registered functionalities of the same session id. We present $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ in the full version [16].

We now elaborate on the transaction diffusion mechanism in the ideal world. To start with, we further extend the fair-order ledger functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ in [15] by introducing two additional internal variables `profiles` and $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$. The variable `profiles` is an append only list that records tuples of form $(\mathsf{P}, \mathtt{tx}, \tau)$, indicating that a transaction `tx` is received by a registered honest party P at time τ. And $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$ is a mapping from `tx` in the transaction buffer to a random string m. Upon initialization, the ledger sets `profiles` and $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$ as empty and whenever the honest (and synchronized) party set $\mathcal{H} \neq \emptyset$, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ maintains the invariant that it is registered as a functionality to $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$.

Upon a new transaction `tx` is submitted from party P_s to our ledger, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ adds `tx` to `buffer` if it is valid and, depending on whether the transaction is sent privately (refer to Sect. 4.2 for more details), $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ performs the following operations accordingly. When `tx` is sent in the clear, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ leaks `tx` to $\mathcal{A}$ and if additionally P_s is honest, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ submits `tx` to $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ on behalf of P_s ($\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ will accept this request as P_s is an honest party that shares the same session identifier with $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$). When `tx` is issued privately, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ leaks only the size of `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 `tx` to m in $\mathsf{F}_{\mathsf{tx}\to\mathsf{s}}$. Moreover, in case that P_s is honest, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ submits m to $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ on behalf of P_s. Note that, in this procedure the ledger learns all random strings that are associated with a transaction `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 $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$. Specifically, when a $(\text{MAINTAIN-LEDGER}, \mathsf{sid}, \mathsf{minerID})$ 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 $\vec{\mathcal{I}}_H^T$), our ledger, before forwarding this command to $\mathcal{A}$, sends the FETCH command to $\mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ on behalf of P. Upon receiving as reply $(\text{FETCH}, \mathsf{sid}, \mathbf{M})$ where $\mathbf{M}$ is a vector of strings, for each string $m \in \mathbf{M}$ with order, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ first checks if a transaction `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 `tx`), it appends a tuple $(\mathsf{P}, \mathtt{tx}, \tau_{\mathcal{L}})$ to `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 `profiles` with order. And at any time of the execution, ledger $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ 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 [15] 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 $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair++}}$ 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 [4,15], upon the environment $\mathcal{Z}$ sends a client P a submit transaction command $(\textsc{submit}, \mathsf{sid}, \mathtt{tx})$, the dummy party 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 $(\textsc{submit}, \mathsf{sid}, \mathtt{tx}, \mathsf{acc})$ 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 $(\textsc{submit}, \mathsf{sid}, ((0^\kappa, \mathsf{B}, \mathsf{aux}), \mathsf{acc}))$, where 0^κ indicates that B tokens are to be added to acc, and aux contains a tag showing

[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.

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. $\mathtt{state} = \mathsf{Blockify}(\mathtt{tx}_{11}, \ldots \mathtt{tx}_{1n_1}) \| \ldots \| \mathsf{Blockify}(\mathtt{tx}_{\ell 1}, \ldots \mathtt{tx}_{\ell n_\ell})$ (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 $\perp$).

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 $\mathtt{state}$.

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

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,

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

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 [4,15]). 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.

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

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

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 Sect. 5.1 for our transaction fee protocol.

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 and the formal specification of ExtendPolicy in the full version [16].

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. The formal description of these predicates is presented in the full version [16].

Order Linearizability. Consider one "timed" variant of receiver order fairness—order linearizability [25,30]. This serialization policy enforces a transaction `tx` to be ordered before `tx`$'$, if there exists a time t such that all parties receives `tx` before t and `tx`$'$ after t. And it gives up on the order of any transaction pairs when there is an overlap between their receiving timestamps.

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

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.

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 [24, Definition 2], and it captures a large family of order fairness defined based on local relative orders (e.g., [23]). 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 $\mathtt{NxtBC} = (\mathrm{txid}_1, \ldots, \mathrm{txid}_\ell) \parallel aux$. Then, regarding the extend policy, once it processes $\mathtt{NxtBC}$, ExtendPolicy also passes such aux to the ValidOrder predicate. 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 order linearizability which simply ignores any aux).

[9] Regarding protocols, one can mitigate such problem by adopting the profile-bootstrapping approach in [24, 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.

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 [24, 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 `tx`, if no valid transaction precedes `tx` has not been already proposed by the adversary.

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 Sect. 5.1. In Sect. 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 Ouroboros-Genesis [3] as the underlying PoS blockchain of our protocol, and show how to cast the protocol $\Pi_{\mathsf{ss\text{-}tx}}$ on chain in Sect. 5.3. In Sect. 5.4, we pack all components together and provide an overview of our protocol $\Pi_{\mathsf{ledger}}^{\mathsf{fair++}}$. Due to space limit, we defer the code specification of all protocols to the full version of this paper [16].

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

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

[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 Sect. 5.3 for more details.

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

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

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 the full version [16].

Paying Transaction Fees. To spend a private account and pay transaction fees (i.e., upon receiving $(\text{SUBMIT}, \text{sid}, \mathtt{tx}, \mathsf{acc})$ 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).

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

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 Sect. 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 Sect. 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).

[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.

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):

$$\mathsf{R}_{\mathsf{key\_gen}} = \left\{ \begin{array}{c} \big(\mathsf{epk}_i, \{c_{i,j}\}_{j\in[n]}\big), \\ \big(\mathsf{esk}_i, f, \{r_{i,j}\}_{j\in[n]}\big) \end{array} \middle| \begin{array}{c} \mathsf{esk}_i \text{ is a validsecret key w.r.t. } \mathsf{epk}_i \\ \wedge \{\mathsf{esk}_{i,j}\}_{j\in[n]} = \mathsf{SS}_t(\mathsf{esk}_i, f) \\ \wedge \big(\forall j \in [n], c_{i,j} = \mathsf{commit}(\mathsf{esk}_{i,j}, r_{i,j})\big) \end{array} \right\}. \tag{4}$$

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.

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:

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

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

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

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 $H(s)$.

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 $(\eta \,\|\, \mathtt{sl} \,\|\, \mathtt{TEST})$ 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 $y < T_{\mathsf{P}}^{\mathsf{ep},\mathsf{cmte}} = 2^{\ell_{\mathsf{VRF}}} \cdot \alpha_{\mathsf{P}}^{\mathsf{ep}}$ (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 stage 1 in Sect. 5.2, (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 $\Pi_{\mathsf{ss\text{-}tx}}$. Notice that naïvely running Protocol $\Pi_{\mathsf{ss\text{-}tx}}$ 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 $\Pi_{\mathsf{ss\text{-}tx}}$ 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

[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 $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$ to issue transactions.

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

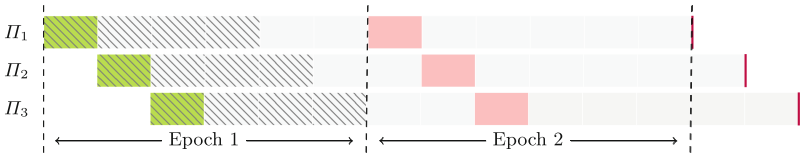

Fig. 1. An illustration of the pipelined invocation of Protocol $\Pi_{\mathsf{ss\text{-}tx}}$. The lime interval represents the slots where parties self-elect as committee members (Stage 1 of $\Pi_{\mathsf{ss\text{-}tx}}$); all ephemeral public keys should be recorded in intervals marked with slant lines; the pink interval shows the slots where parties secret-share their transactions towards the committee that is elected in the same invocation of $\Pi_{\mathsf{ss\text{-}tx}}$ (Stage 2 of $\Pi_{\mathsf{ss\text{-}tx}}$); 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 [24], we here focus on how to migrate the approach to a PoS setting.

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 Sect. 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 Protocol Overview

Finally, we provide an overview of our protocol $\Pi^{\mathsf{fair++}}_{\mathsf{ledger}}$ via the transaction issuing procedure and ledger maintenance actions. A full specification of the protocol is presented in the full version [16].

Issuing a New Transaction. In order for a client P to issue a new transaction upon input (SUBMIT, $\mathtt{tx}, \mathsf{acc}$), it executes the IssueNewTransaction procedure as follows.

1. If acc is empty, send $\mathtt{tx}$ directly to $\mathcal{G}^{\mathsf{tx}}_{\mathsf{Diffuse}}$.
2. If $\mathtt{tx}$ is a funding transaction and acc is fresh to P, it sets up a new private account, and sends $\mathtt{tx}$, the account commitment and proof to $\mathcal{F}_{\mathsf{Diffuse}}$.
3. If $\mathtt{tx}$ is a funding transaction and acc has been recorded before, P updates acc and sends $\mathtt{tx}$ with the updated account to $\mathcal{F}_{\mathsf{Diffuse}}$.
4. Otherwise, $\mathtt{tx}$ is a private transaction paying fees from acc. Let $\mathtt{sl}$ denote the current time (w.l.o.g. we assume $\mathtt{sl} > R$). Then, set $\lceil i = (\mathtt{sl} - R)/(R/6) \rceil$, P extracts all valid ephemeral public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_k$ generated during $[i \cdot R/6, (i+1) \cdot R/6)$ and recorded in a block with timestamp $t \in [i \cdot R/6, (i+4) \cdot R/6)$. P first pays the fee for $\mathtt{tx}$; then she secret shares her transaction to $\mathsf{pk}_1, \ldots, \mathsf{pk}_k$ and diffuses the transaction shares and the fee proof.

Ledger Maintenence. Upon receiving input MAINTAIN-LEDGER, an alert party (i.e., it owns all resources to run the protocol and maintains a synchronized ledger state) performs the following operations.

1. Fetch information from the network over which blocks, transactions, profiles, and ephemeral keys are sent.
2. Filter and update local chain $\mathcal{C}_{\mathsf{loc}}$.
3. Update ledger state: Build transaction dependency graph over the chain, decrypt settled transactions using published ephemeral keys, and filter invalid transactions among them and also remove invalid transactions (if they are sent unencrypted) from the buffer.
4. Try decrypting ephemeral key shares generated before slot $\mathtt{sl} - 2R$ using her long-term private key, and emit the successfully decrypted shares.
5. Run the staking procedure.
   (a) Evaluate slot-leadership for the next block and (when successful) create and emit a new block, containing all ephemeral key related messages and new profiles that has not yet appeared on the blockchain.
   (b) Evaluate slot-leadership for ephemeral-key committee membership and (when successful) generate a new ephemeral key pair, secret share the private key and emit the message to the network.

(c) Evaluate slot-leadership for profiles and (when successful) create and emit a new profile, containing all valid transactions as per the order that they are received (and also remove all transactions whose tag duplicates an existing one).

6 Security Analysis

We provide a high-level overview of the security analysis, and state in Theorem 1 the composable security guarantee of our protocol $\Pi_{\mathsf{ledger}}^{\mathsf{fair++}}$. The full analysis is presented in [16].

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, $\mathcal{S}$ simulates the $\mathsf{IssueNewTransaction}$ 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 $\mathtt{tx}$ and send them to the simulated network. In order to learn $\mathtt{tx}$, $\mathcal{S}$ 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 $\mathtt{tx}$ that is associated with txid. $\mathcal{S}$ then programs the $\mathcal{G}_{\mathsf{rpoRO}}$ on point s (and hides this "isPrograme" 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, once $\mathcal{A}$ decides to corrupt a party P, the simulator learns the whole transaction history of P. For each transaction from input $(\textsc{submit}, \mathtt{tx}, \mathsf{acc})$, $\mathcal{S}$ 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.

Theorem 1. *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 [16]). Assuming that the IND-CPA security of* PKE *and the hardness of computational Diffe-Hellman, the the protocol $\Pi_{\mathsf{ledger}}^{\mathsf{fair}++}$ realizes the ledger functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}++}$ in the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{Diffuse}}^{\mathsf{tx}}, \mathcal{G}_{\mathsf{RO}}, \mathcal{G}_{\mathsf{rpoRO}}, \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{F}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{INIT}}, \mathcal{F}_{\mathsf{VRF}}, \mathcal{F}_{\mathsf{KES}})$-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* $\mathtt{windowSize} = k, \mathtt{Delay} = R/6, \mathtt{waitTime} = R, \mathtt{maxTime}_{\mathtt{window}} = \mathtt{windowSize}/\tau_{\mathsf{CG}}, \mathtt{minTime}_{\mathtt{window}} = (1-\mu)\mathtt{windowSize}$.

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

1. Badertscher, C., Campanelli, M., Ciampi, M., Russo, L., Siniscalchi, L.: Universally composable SNARKs with transparent setup without programmable random oracle. Cryptology ePrint Archive, Report 2024/1549 (2024). https://eprint.iacr.org/2024/1549
2. Badertscher, C., Canetti, R., Hesse, J., Tackmann, B., Zikas, V.: Universal composition with global subroutines: Capturing global setup within plain UC. In: Pass, R., Pietrzak, K. (eds.) TCC 2020: 18th Theory of Cryptography Conference, Part III. Lecture Notes in Computer Science, vol. 12552, pp. 1–30. Springer, Cham, Switzerland, Durham, NC, USA (Nov 16–19, 2020). https://doi.org/10.1007/978-3-030-64381-2_1
3. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018: 25th Conference on Computer and Communications Security, pp. 913–930. ACM Press, Toronto, ON, Canada (Oct 15–19, 2018). https://doi.org/10.1145/3243734.3243848
4. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: a composable treatment. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 324–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_11

5. Ben-Sasson, E., et al.: Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press, Berkeley, CA, USA (May 18–21, 2014). https://doi.org/10.1109/SP.2014.36
6. Benhamouda, F., et al.: Can a public blockchain keep a secret? In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12550, pp. 260–290. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64375-1_10
7. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 524–541. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44647-8_31
8. Camenisch, J., Drijvers, M., Gagliardoni, T., Lehmann, A., Neven, G.: The wonderful world of global random oracles. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10820, pp. 280–312. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78381-9_11
9. Campanelli, M., David, B., Khoshakhlagh, H., Konring, A., Nielsen, J.B.: Encryption to the future - A paradigm for sending secret messages to future (anonymous) committees. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology – ASIACRYPT 2022, Part III. Lecture Notes in Computer Science, vol. 13793, pp. 151–180. Springer, Cham, Switzerland, Taipei, Taiwan (Dec 5–9, 2022). https://doi.org/10.1007/978-3-031-22969-5_6
10. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000). https://doi.org/10.1007/s001459910006
11. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_4
12. Canetti, R., Kolby, S., Ravi, D., Soria-Vazquez, E., Yakoubov, S.: Taming adaptivity in YOSO protocols: The modular way. In: Rothblum, G.N., Wee, H. (eds.) TCC 2023: 21st Theory of Cryptography Conference, Part II. Lecture Notes in Computer Science, vol. 14370, pp. 33–62. Springer, Cham, Switzerland, Taipei, Taiwan (Nov 29 – Dec 2, 2023). https://doi.org/10.1007/978-3-031-48618-0_2
13. Canetti, R., Sarkar, P., Wang, X.: Triply adaptive UC NIZK. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology – ASIACRYPT 2022, Part II. Lecture Notes in Computer Science, vol. 13792, pp. 466–495. Springer, Cham, Switzerland, Taipei, Taiwan (Dec 5–9, 2022). https://doi.org/10.1007/978-3-031-22966-4_16
14. Chiang, J.H.y., David, B., Eyal, I., Gong, T.: FairPoS: input fairness in permissionless consensus. In: Bonneau, J., Weinberg, S.M. (eds.) 5th Conference on Advances in Financial Technologies (AFT 2023). Leibniz International Proceedings in Informatics (LIPIcs), vol. 282, pp. 10:1–10:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023) https://doi.org/10.4230/LIPIcs.AFT.2023.10
15. Ciampi, M., Kiayias, A., Shen, Y.: Universal composable transaction serialization with order fairness. In: Reyzin, L., Stebila, D. (eds.) Advances in Cryptology – CRYPTO 2024, Part II. Lecture Notes in Computer Science, vol. 14921, pp. 147–180. Springer, Cham, Switzerland, Santa Barbara, CA, USA (Aug 18–22, 2024). https://doi.org/10.1007/978-3-031-68379-4_5
16. Ciampi, M., Kiayias, A., Shen, Y.: Universally composable transaction order fairness: Refined definitions and adaptive security. Cryptology ePrint Archive, Paper 2025/1527 (2025). https://eprint.iacr.org/2025/1527
17. Daian, P., et al.: Flash boys 2.0: frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: 2020 IEEE Symposium on Security

and Privacy. pp. 910–927. IEEE Computer Society Press, San Francisco, CA, USA (May 18–21, 2020). https://doi.org/10.1109/SP40000.2020.00040
18. Garg, S., Gentry, C., Sahai, A., Waters, B.: Witness encryption and its applications. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th Annual ACM Symposium on Theory of Computing, pp. 467–476. ACM Press, Palo Alto, CA, USA (Jun 1–4, 2013). https://doi.org/10.1145/2488608.2488667
19. Gentry, C., et al.: YOSO: you only speak once. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12826, pp. 64–93. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84245-1_3
20. Gentry, C., Halevi, S., Magri, B., Nielsen, J.B., Yakoubov, S.: Random-index PIR and applications. In: Nissim, K., Waters, B. (eds.) TCC 2021. LNCS, vol. 13044, pp. 32–61. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90456-2_2
21. Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 477–498. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_27
22. Kelkar, M., Deb, S., Long, S., Juels, A., Kannan, S.: Themis: Fast, strong order-fairness in byzantine consensus. In: Meng, W., Jensen, C.D., Cremers, C., Kirda, E. (eds.) ACM CCS 2023: 30th Conference on Computer and Communications Security, pp. 475–489. ACM Press, Copenhagen, Denmark (Nov 26–30, 2023). https://doi.org/10.1145/3576915.3616658
23. Kelkar, M., Zhang, F., Goldfeder, S., Juels, A.: Order-fairness for byzantine consensus. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 451–480. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_16
24. Kiayias, A., Leonardos, N., Shen, Y.: Ordering transactions with bounded unfairness: Definitions, complexity and constructions. In: Joye, M., Leander, G. (eds.) Advances in Cryptology – EUROCRYPT 2024, Part III. Lecture Notes in Computer Science, vol. 14653, pp. 34–63. Springer, Cham, Switzerland, Zurich, Switzerland (May 26–30, 2024). https://doi.org/10.1007/978-3-031-58734-4_2
25. Kursawe, K.: Wendy, the good little fairness widget: Achieving order fairness for blockchains. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. p. 25 36. AFT '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3419614.3423263
26. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_9
27. Reiter, M.K., Birman, K.P.: How to securely replicate services. ACM Trans. Program. Lang. Syst. **16**(3), 986–1009 (1994). https://doi.org/10.1145/177492.177745
28. Schneider, F.B.: 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
29. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979). https://doi.org/10.1145/359168.359176
30. Zhang, Y., Setty, S., Chen, Q., Zhou, L., Alvisi, L.: Byzantine ordered consensus without byzantine oligarchy. In: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. USENIX Association, USA (2020)