Universal Composable Transaction Serialization with Order Fairness*

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

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

Abstract

Order fairness in the context of distributed ledgers has received recently significant attention due to a range of attacks that exploit the reordering and adaptive injection of transactions (violating what is known as "input causality"). To address such concerns an array of definitions for order fairness has been put forth together with impossibility and feasibility results highlighting the difficulty and multifaceted nature of fairness in transaction serialization. Motivated by this we present a comprehensive modeling of order fairness capitalizing on the universal composition (UC) setting. Our results capture the different flavors of sender order fairness and input causality (which is arguably one of the most critical aspects of ledger transaction processing with respect to serialization attacks) and we parametrically illustrate what are the limits of feasibility for realistic constructions via an impossibility result. Our positive result, a novel distributed ledger protocol utilizing trusted enclaves, complements tightly our impossibility result, hence providing an *optimal* sender order fairness ledger construction that is also eminently practical.

*An abridged version of this paper appears in *Proc. CRYPTO 2024*.

Contents

1 Introduction

An important variant of the consensus problem [PSL80] that has recently gained significant attention asks for the maintenance and continuous extension of a ledger of transactions. This "ledger" variation of the classical consensus problem has two fundamental objectives. *Consistency:* the ledgers in the views of any two participants are consistent with each other in the sense that they are either equal, or one is a prefix of the other. *Liveness:* there is a time bound, after which, any valid transaction will be incorporated in the view of the ledger of any honest participant. It is easy to see that these two properties individually can be easily satisfied by trivial protocols, however achieving them in tandem is an interesting protocol design question.

Ledger consensus is covered within the context of state machine replication (SMR) [Sch90]. More recently, the Bitcoin blockchain protocol [Nak08] provided a solution in a "permissionless" setting where participants do not know of each other and only a public setup operation is permitted. The protocol and its properties has been analyzed over a sequence of works [GKL15, PSs17, GKL17] that established its explicit guarantees regarding liveness and consistency.

The main task of a ledger consensus protocol is to continuously serialize the submitted transactions. While such serialization is not necessary for all applications of transaction services (e.g., payments is a notable exception [DM16, GKM$^+$22]), serialization is of critical importance in the context of smart contracts. Indeed, when multiple parties interact with the same contract, the ordering of transactions can be crucial in terms of the effects that each transaction may have on the smart contract state. Observe that the properties of consistency and liveness necessitate that a certain serialization must take place (as the ledger must be well defined as a sequence of transactions) nevertheless do not impose a lot of constraints on that serialization except for the straightforward fact that transactions submitted very far apart from each other (in particular at least one full liveness property window) will be serialized in the order they were submitted.

This state of affairs leaves undetermined how transactions that are submitted more closely to each other should be serialized. This is by far not a theoretical consideration: by exploiting the behavior of Ethereum decentralized finance (DeFi) contracts it was shown how to deploy Miner Extractable Value (MEV) attacks [DGK$^+$20] that exploit the ability of Ethereum miners to influence transaction serialization. By strategically placing suitable DeFi transactions in the serialization order of submitted transactions, miners gain additional profits (beyond what is provided in terms of transaction fees and newly minted coins associated with block production).

Motivated by mitigating such MEV attacks, a series of works attempted to enhance the ledger with a fair-order policy based on the maintainers' local receiving time of transactions. Two main directions emerged in this line of "receiver order fairness" approach. On one hand, batch order fairness [KZGJ20, KDK22] (and later further improvements [CMSZ22, KLS24]) focus on defining valid transaction order based on their local *relative* order. On the other hand, by assuming a system-wide clock, timed order fairness [Kur20, ZSC$^+$20] specifies a suitable order for a pair of transactions, if there is a point in time t such that all honest parties receive one transaction before t and the other transaction after t.

It is worth observing that looking at the receiving end of transaction dissemination, provides at best modest protection against front running attacks as it pits the adversary in a network game against the honest parties: the adversary may still attempt to front run when it becomes aware of a transaction by rushing the network delivery of its own transactions. Such attacks violate what is known as "input causality" a concept studied early by Reiter and Birman [RB94] with the aim to protect against an adversary that settles a transaction that depends in a meaningful way on transactions that are pending. This property has been later studied in the context of atomic broadcast by Cachin *et al.* [CKPS01] and recently in blockchain systems by Malkhi and

Szalachowski [MS23]. It can be seen that input causality complements receiver order fairness as it focuses on the sender side and tries to protect transactions in transit from being meaningfully front-run. Nevertheless input causality cannot entirely substitute receiver order considerations, since, even if input causality holds perfectly, one would still want to mandate that serialization adheres to a fairness criterion that attempts to serialize transactions in a way consistent with the timings that they were generated, disseminated and received by the ledger maintainers.[1]

1.1 Our Results

Motivated by the above considerations we set out to model formally in the universal composition (UC) setting [Can01] the concept of order fair transaction serialization and realize it under plausible assumptions in a proof-of-work (PoW) setting.

Our starting point is the previous UC formalization of [BMTZ17] that focused on abstracting ledger consensus as an ideal functionality and showing how it can be realized in the PoW setting. One of the key features of this functionality is its ledger "extend policy" that allows the ideal world adversary to drive the progress of the ledger while being subject only to the liveness and "chain quality" properties [GKL15]. Note that no order fairness is provided by this functionality as the adversary is free to choose the order of pending transactions at will and even inject transactions adaptively, based on the transactions that honest parties have submitted for inclusion to the ledger.

Modeling. To capture comprehensively all fair ordering considerations, we adapt the ledger functionality of [BMTZ17] so that: (i) It 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. (ii) It offers a parametric extend policy that constraints the adversary to follow an order that belongs to a class of admissible reorderings of the input transactions. Observe that this two-pronged approach combines input causality and receiver fair order considerations. This ideal functionality model results to a class of fair ledgers.

The most stringiest extend policy one can imagine for serializing transactions is the exact order that they were submitted to the ideal functionality. Note that in many ways such a "perfect" order is unreasonable; taking into account the global clock in the model, it could be the case that two transactions are "simultaneously" delivered w.r.t. the global clock. It follows that even though the perfect transaction order is available to the ideal functionality, it would be unreasonable to expect to realize it. Following this, our strictest formalization of sender order fairness is the one that restricts the adversary to conform to the arrival times to the ledger functionality as recorded by the global clock functionality. It is easy to see that one cannot expect to realize that level of fairness either. The reason is that in any setting where transaction delivery is not instantaneous (a setting that covers pretty much all reasonable deployment scenarios), there is no conceivable way that a ledger system can obey it. We formalize this result and in fact prove a much stronger *relativized* version of this impossibility theorem: even given any "private" helper functionality (i.e., one that is just responding to the caller) and a correlated random string, it is infeasible to achieve sender order fairness in a network with bounded delays.

The above results suggest that a relaxation is in order. To capture this, we introduce a natural but more relaxed sender order fairness property, δ-approximate order fairness, where the adversary is allowed to violate sender order fairness but only in a limited way, namely for transactions that are

[1] To see why by an example, consider two DeFi traders that react to an external signal and perform a certain type of transaction (e.g., selling an asset). They are not aware of each other's transaction so input causality is not a consideration; but it is still desirable that the system respects the order with which they submit their transactions.

sent δ time apart. It is easy to show that our impossibility result for sender order fairness extends to the setting where δ is below the network delay Δ, even in the relativized sense as described above. This leaves open the question of whether there exists a plausible helper functionality under which we can prove Δ-approximate order fairness.

It will not take long for the reader to guess a helper ideal functionality under which it is fairly easy to achieve our Δ-approximate sender order fairness formulation: for example, consider a functionality that encrypts all valid transactions and allows parties to disseminate them in an encrypted form so that their validity can be publicly verified and subsequently decrypt them only when they are settled. Given such functionality, it is possible for parties to disseminate transactions in an encrypted form and subsequently open them to reconstruct the state of the ledger. While this approach can work (and indeed it is folklore, see also [MS23] for a specific implementation) it imposes a rather heavy price if it is to be somehow realized from simpler components: it requires a shared private state (in order to facilitate decryption) and an ability to realize that the settlement of the encrypted transactions has taken place (since premature decryption of transactions would break security). Maintaining a shared private state can be expensive in the permissionless setting, e.g., using techniques such as YOSO MPC [BGG+20] it is possible for a committee to keep redistributing a secret-key of a threshold encryption function — with the downside of a quadratic communication complexity (in the security parameter) overhead at each round to facilitate the continuous resharing of the private key between successive committees.

Instead, poised to obtain a construction relevant to practice, we focus on the concept of a private functionality, thinking of it as a trusted execution enclave in the model of [PST17]. We ask whether it is possible to realize our approximate sender anonymous ledger functionality in a hybrid setting where the private functionality is instantiated as an enclave and is seeded only by an initialization string that is drawn from the correlated randomness functionality (this captures the ability of an enclave to be safely initialized by the enclave issuer). In particular this means that our enclave functionality does not have any shared private state across different enclave instances: each enclave has its own independent signing and encryption key. Finally, we also insist on transaction submission being a public operation that does not require access to an enclave.

Protocol overview. We first show how to enforce input-causality, and then we show how to lift the security of our protocol to realize a ledger that has Δ-approximate order fairness. Our protocol is built on top of a Nakamoto-style PoW blockchain $\mathcal{C}$, whose maintainers are equipped with a stateless[2] trusted execution environment[CD16] (enclave for brevity). Each enclave is equipped with its own public-key encryption key pair $(\mathsf{pk}, \mathsf{sk})$[3], where only pk is revealed, while sk is hidden to everyone (including the owner of the enclave). Every output generated by the enclave is signed with a secret key known only to the enclave, but verifiable by anyone holding an *attested* verification key VK.

We require the maintainers to publish the enclave public-keys $(\mathsf{pk}, \mathsf{VK})$ on the blockchain[4], upon registration. When a client wants to issue a transaction $\mathtt{tx}$, it first samples a subset of polylogarithmic size S_{pk} of the registered public-keys (more detail on how S_{pk} is chosen is provided

[2]By stateless here we mean without secure counters, or without any mechanism that prevents the adversary from resetting the enclave.

[3]Note that in the hybrid model where communication takes place strictly via the diffusion functionality $\mathcal{F}_{\mathsf{Diffuse}}$ that we use to model the peer-to-peer network, sharing a secret key sk among n enclaves requires diffusing n distinct messages from a single source — a prohibitive overhead. Contrary, in our protocol, in the worst case, each party has to diffuse polylogarithmic in the number of enclaves sized messages. Combining the capabilities of the enclave with the gossiping protocol over point-to-point channels may improve communication complexity — we leave exploring this interesting direction for future work.

[4]The maintainer will also issue the hash of the code that the enclave will run, which will make sure that every registered enclave will behave accordingly to what our protocol prescribes.

later) and encrypts $(\mathtt{tx}, h)$ using all the selected public keys, where h corresponds to the block-header of $\mathcal{C}$, thus obtaining a set of ciphertexts ct. Then, the client generates a zero-knowledge proof π proving that all the ciphertexts contain the same transaction and that this transaction is not equal to 0^κ. The client then diffuses to the network (ct, π).

Each maintainer verifies the zero-knowledge proof, and if valid, it selects the ciphertext generated using its public key pk (if such a ciphertext does not exist, then (ct, π) is simply ignored). Then the maintainer queries the enclave with the ciphertext along with h', which corresponds to the current blockchain header. Upon receiving the query, the enclave decrypts the ciphertext, thus obtaining the pair $(\mathtt{tx}, h)$ and checks how many blocks apart the chain with header h is compared to h'. If the number of blocks is more than Λ, then the enclave returns $\mathtt{tx}$ to the caller, else it waits to receive a new block header that satisfies the above condition.

This mechanism guarantees that an honest transaction can be decrypted only after a minimum amount of time, which is defined by the number of rounds required to add Λ blocks to the PoW blockchain. Moreover, the selection of S_{pk} guarantees that at least one public-key in the set belongs to an honest maintainer (more detail on this later), hence, the honest transactions will be surely decrypted. Once $\mathtt{tx}$ is obtained, it is diffused to the network, and it is the maintainers' goal now to include $\mathtt{tx}$ in the blockchain. Given that the underlying blockchain protocol we devise provides liveness, we are guaranteed that all honest transactions will appear in the blockchain.

The above approach seems to not cover the following attack. A malicious party could generate a ciphertext, and send it only to corrupted maintainers. The maintainers can decrypt the ciphertext as an honest party would, thus obtaining a signature for $\mathtt{tx}$, and decide whether to diffuse it or not in the network based on honest transactions that have been already decrypted and diffused in the network. This form of *adaptive rejection* is clearly incompatible with our goals.

To solve this problem we design a selection process for the set S_{pk}, which guarantees that the party generating the set (even if it is corrupted) always includes in S_{pk} a public-key of an honest maintainer. In particular, we prove that as long as some constant fraction of the enclave owners is honest, then S_{pk} will contain with overwhelming probability at least one public-key of an honest party. We note that this could be achieved by simply requiring each transaction issuer to encrypt the transaction using all the enclave public keys. We instead propose an approach that yields the set S_{pk} being of polylogarithmic size. We refer the reader to the technical section for more details on this. On a final note, we stress that the enclaves we use are quite simple. In particular, no synchronization is required among them, and the adversary has full control of the internal clock of the enclaves it owns.

Transaction timestamping. We will now argue how to modify the blockchain $\mathcal{C}$, to finally realize our ledger with Δ-approximate order fairness (recall that Δ denotes the network delay). We modify the Nakamoto-style PoW $\mathcal{C}$ by binding the mining procedure of the blocks that form the chain $\mathcal{C}$ with a new type of blocks called *profile blocks*, using 2×1 (pronounced *two-for-one*) PoW [GKL15]. In 2×1 PoW, a hash function output u is checked twice for u and its reversed bit-string $[u]^{\mathsf{R}}$. If $u < T$, a block that extends the chain $\mathcal{C}$ is produced; and if $[u]^{\mathsf{R}} < T$, a new profile block (PB) is mined (which we will detail soon). In such a way, the mining procedure of the chain $\mathcal{C}$ and profile blocks are *bound* together and the adversary cannot gain advantage on either type of block by dropping from mining the other.

The blockchain $\mathcal{C}$ we use for our protocol is a sequence of blocks and all validation algorithms follow the longest-chain rule. The only difference between our chain $\mathcal{C}$ and that in, e.g., Bitcoin, is that blocks in $\mathcal{C}$ do not include transactions directly. The profile blocks instead are those storing the actual transactions, which in our case will correspond to a list of pairs of the form (tag, t) where $\mathsf{tag} = (ct, \pi)$ (which represents a *transaction identifier*) and t denotes the local receiving time. For a

transaction $\mathtt{tx}$ with identifier tag to be declared as *included* in a block $\mathcal{B}$ of the ledger, our protocol requires that the majority of the profile blocks connected to K consecutive blocks report $(\mathsf{tag}, \cdot)$. In a nutshell, we require each party generating a profile block to vote for the transaction and the position this transaction should finally have once it is included in the ledger.

In more detail, we employ two additional parameters for validating profile blocks — the recency parameter R, which is used to guarantee the freshness of profile blocks, and the profile window length parameter K. Specifically, a profile block PB should be allowed to be connected to a valid block $\mathcal{B}$ in the settled part of the blockchain, only if PB has not been mined too long ago. Let ℓ denote the height of block $\mathcal{B}$ that PB attaches to. PB is considered as a valid profile block only when PB is included in blocks with height less than $\ell + R$.

Finally, the timestamp of $\mathtt{tx}$ (or, its position in the final ledger) is computed by calculating the median timestamps of tag from all profile blocks included in blocks with height at most $\ell + K$. Note that if a profile block in the K-block window does not report an entry $(\mathsf{tag}, \cdot)$, it is counted as reporting $(\mathsf{tag}, +\infty)$ (i.e., the miner had never received $\mathtt{tx}$ thus cannot decrypt it). Formally, the timestamping of the transaction is computed as follows:

$$\mathsf{TS}(\mathsf{tag}) := \mathsf{med}\{t \mid (\mathsf{tag}, t) \in \mathsf{PB} \in B \wedge \ell \leq \mathsf{height}(B) < \ell + K\}.$$

The median provides Δ-order fairness because we can argue that in the K-block window associated with tag the majority of the timestamps are generated by honest parties.

On the enclave assumption. For our protocol we rely on enclaves, and one may wonder whether the use of enclaves trivializes the problem we are trying to solve. Indeed, it could be that realizing a ledger that achieves δ-order fairness with $\delta < \Delta$ becomes a trivial task with such assumptions. We prove that our result is optimal. In particular, we argue that even under stronger assumptions than the ones we use (e.g., enclaves synchronized with each other) it is impossible to design a protocol with better fairness than the one we provide. The high-level intuition behind this is that order fairness is strongly influenced by network delay. We also conjecture that it is possible to obtain a similarly optimal result by employing out of the box YOSO MPC [BGG+20] with the threshold encryption approach of [MS23] — nevertheless the relevance of such a construction is of less interest in practice and for this reason we do not pursue it here.

We also highlight that realizing sender order fairness and processing transactions in our protocol requires possession of a trusted enclave and hence it is only possible for permissioned nodes; specifically, in our setting such a "permissioned" node refers to a node that has been initialized with such an enclave and can be authenticated as such by other permissioned nodes. Note that our protocol is a PoW-based blockchain and hence it allows non-permissioned nodes to still contribute to the protocol without processing encrypted transactions; in fact such unpermissioned blocks may even contain transactions that are submitted unencrypted for which the clients do not wish to have them protected for sender-order fairness. As a side remark, this simple observation suggests that our protocol can be implemented as a "soft-fork" over bitcoin — we leave the details of this implementation to be explored in future work.

Additionally, the security of our protocol relies on two assumptions — honest majority in terms of computational power and a constant fraction of the enclaves being possessed by honest hosts. While the second assumption is arguably weaker, we comment on how these two assumptions can be unified into one: Suppose all miners are equipped with enclaves and the honest parties account for the majority of computational power yet possess only a minority constant number of enclaves (i.e., the adversary is allowed to possess an arbitrary number of enclaves). Our approach is to extend the 2×1 PoW to 3×1 PoW (by checking non-overlapping 0s at different positions of the RO output), thus additionally mining the enclave public keys (each message mines only one key).

These PoW-ed enclave public keys are included in the blockchain, and the key subset selection procedure is then applied on the set of public keys with *weights* based on their PoW (for instance, a key with two unique PoW-ed messages is weighed two times than another key with one unique message). This adaption, with appropriate protocol parametrization, guarantees that the majority of weighted public keys are controlled by honest parties regardless of the fraction of honest keys.

Finally, one may consider it a compromise to use enclaves in a permissionless setting: we argue that this is not the case in practice. For instance, it is already the case that Bitcoin miners use specialized hardware to engage in mining and there are a handful of providers that offer such hardware in the market. Transitioning to a setting where such specialized hardware is also utilizing a trusted enclave may be a small step — especially if the benefits are substantial — as we demonstrate here: optimal sender order fairness.

2 Preliminaries

UC basics. We provide our protocols and security proofs in Canetti's universal composition (UC) framework [Can00]. We discuss the main components of our real-world model (including the associated hybrids).

We assume that the reader is familiar with simulation-based security and has basic knowledge of the (G)UC framework. We review all the 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.

We now 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.[5] We denote the identities of parties by P_i, i.e. $\mathsf{P}_i = (\mathsf{pid}_i, \mathsf{sid}_i)$, 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. Our protocols are synchronous (G)UC protocols [BMTZ17]: parties have access to a (global) clock setup, denoted by $\mathcal{G}_{\mathsf{Clock}}$. and can communicate over a network diffuse functionality $\mathcal{F}_{\mathsf{Diffuse}}$ (mode details on these functionalities are provided later). 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 deregister, and allow the adversary to learn the current set of registered parties. Additionally, global setups allow any other setup (or functionality) to register and deregister with them[6] and also allow other setups to learn their set of registered parties.

Next we elaborate on the main hybrid functionality used in our paper.

Clock and diffusion functionality. Following the treatment in [BMTZ17], we model the synchronous processors and bounded-delay network as $\mathcal{G}_{\mathsf{Clock}}$ and $\mathcal{F}_{\mathsf{Diffuse}}$ respectively. The global clock $\mathcal{G}_{\mathsf{Clock}}$ maintains a round index for each session, and this index can only be forwarded when

[5] In the latter case the activation goes to the environment by default.

[6] We note that the functionality can learn the code of the registering party (functionality) by considering the *extended identifier* in [BCH+20].

all registered honest parties (in this session) have finished their computation for this round. The diffuse functionality $\mathcal{F}_{\mathsf{Diffuse}}^{\Delta}$ captures Δ-bounded network — i.e., when a message (either honest or adversarial) is reached by at least one honest party at round r, it is guaranteed to be delivered to all honest parties before round $r + \Delta$. A detailed description of these functionalities is presented in Appendix A.

Global random oracles. The hash function H to generate PoW is modeled as a (global) random oracle $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$; by convention, we use a wrapper functionalities $\mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}})$ to restrict the adversarial and *environmental* access to $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$. We assume honest majority in terms of the computational power to solve PoWs. I.e., in every round the RO wrapper allows more accesses from the honest parties than the corrupted (and environmental) ones. We also adopt a restricted programmable and observable global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ [CDG+18] to model a hash function G that is different from the one used to generate PoW. We discuss these two global random oracles in detail in Appendix A.

Non-interactive zero knowledge. For our construction we use a non-interactive, straight-line simulation extractable (NISLE) proof system that relies on $\mathcal{G}_{\mathsf{rpoRO}}$ as the only setup assumption. A protocol that satisfies such a security definition is provided in [LR22]. We assume familiarity with the notion of simulation extractable NIZK, and refer to Appendix A for the formal definition.

Pseudorandom functions, one-way functions and hardcore bits. Let $F : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$ be a keyed function. F is a pseudorandom function (PRF) if for all PPT distinguisher D it holds that $|\mathbf{Pr}_{k \leftarrow \{0,1\}^n}[D^{F_k(\cdot)} = 1] - \mathbf{Pr}_{f \leftarrow \mathcal{F}_n}[D^{f}(\cdot) = 1]| \leq \mathsf{negl}(n)$. A function $f : \{0,1\}^* \to \{0,1\}^*$ is a one-way function if there exists an efficient algorithm for evaluating f, and for all PPT adversary $\mathcal{A}$, we have $\mathbf{Pr}_{x \leftarrow \{0,1\}^n}[\mathcal{A}(f(x)) \in f^{-1}(f(x))] \leq \mathsf{negl}(n)$. A hardcore bit is a function $b : \{0,1\}^* \to \{0,1\}$ such that for all PPT adversary $\mathcal{A}$, it holds that $|\mathbf{Pr}_{x \leftarrow \{0,1\}^n}[\mathcal{A}(f(x)) = b(x)] \leq \frac{1}{2} + \mathsf{negl}(n)$.

3 Ledger Functionality with Parametric Extend Policy

In order to capture fairness in transaction serialization, we extend the ledger functionality $\mathcal{G}_{\mathsf{Ledger}}$ in [BMTZ17] by enhancing the extend policy with parametric order policy and revising the transaction information leakage behavior (which we denote by $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$). We highlight our main adaptions in this section; refer to Appendix B for a detailed description of the ledger functionality.

Parametric extend policy. The ledger in [BMTZ17] is parametrized with an $\mathsf{ExtendPolicy}$ algorithm such that, for every new block, the adversary is allowed to propose an arbitrary sequence using an arbitrary subset of the transaction buffer, as long as (i) all valid transactions are included timely (liveness); (ii) the proposed ledger grows at a relatively steady rate (chain growth); and (iii) the fraction of honest blocks in any section of the blockchain is lower-bounded (chain quality). This captures exactly the transaction inclusion mechanism in real-world Bitcoin. Meanwhile, subject to the designated chain quality parameter, front-running is possible during the ongoing of a sequence of blocks that is at most the same number of the common prefix parameter.

In order to capture order fairness and towards a more modular design of the core ledger extending mechanism, we propose a new $\mathsf{ExtendPolicy}$ in Algorithm 1 which (i) can be parameterized with different fair-order policies; and (ii) revises the liveness guarantee check (which is necessary since we decouple the consensus and transaction settlement). To improve accessibility, we mark in blue the differences compared with the original $\mathsf{ExtendPolicy}$ in [BMTZ17].

More specifically, upon receiving a proposed block from the adversary, $\mathsf{ExtendPolicy}$ iterates all transactions; for each transaction $\mathtt{tx}$, its *order* and validity with respect to the current state and buffer is evaluated. If $\mathtt{tx}$ turns out to be a valid transaction and extends the ledger following the fair-order policy, the algorithm goes for the next transaction; otherwise, it aborts and returns default

blocks from DefaultExtension. The transaction validation procedure follows that in [BMTZ17]. Regarding the fair-order examination, we let ValidOrder and DefaultExtension be parameters of Algorithm 1. ValidOrder takes the target transaction `tx`, current ledger state `state` and pending transaction buffer `buffer` as input and returns whether `tx` is a valid extension with respect to `state` and `buffer` under a designated fair-order policy; DefaultExtension takes the honest input sequence $\vec{\mathcal{I}}_H^T$, the current ledger state `state` and pending transaction buffer `buffer` and returns the default block under the designated fair-order policy that ledger $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ should specify. The ValidOrder algorithm is called for every transaction (Line 21 in Algorithm 1). Different ValidOrder and DefaultExtension algorithms can implement different fair-order policies, and we discuss them in Section 4.

Alongside with the new ValidOrder examination, we revise the transaction liveness check. Recall that in [BMTZ17], once an honest block is proposed, it should include all relatively recent pending transactions in the buffer to avoid the violation of liveness. Looking ahead, our protocol employs a blockchain scheme that decouples consensus and transaction inclusion (a similar framework has been used in, e.g., [PS17, FGKR20]), hence the liveness parameter is not exactly the same as `windowSize` (which indicates the number of blocks that needs to be pruned to achieve a consistent view on the level of blockchains). We adopt a new parameter `waitTime` for transaction liveness, so the old transaction checks only those sent before time $\tau_{\mathcal{L}} - \mathtt{waitTime}$. Additionally, old transactions will be excluded if they commit to an invalid order with respect to the current ledger. See Line 27 and 28 in Algorithm 1.

Algorithm 1 ExtendPolicy($\vec{\mathcal{I}}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}, \vec{\tau}_{\mathtt{state}}$)

1: $\vec{N}_{\mathrm{df}} \leftarrow$ OrderPolicy.DefaultExtension($\vec{\mathcal{I}}_H^T, \mathtt{state}, \mathtt{NxtBC}, \mathtt{buffer}, \vec{\tau}_{\mathtt{state}}$)
2: $\tau_{\mathcal{L}} \leftarrow$ current ledger time (computed from $\vec{\mathcal{I}}_H^T$)
3: Create local copies of the values `buffer`, `state` and $\vec{\tau}_{\mathtt{state}}$.
4: Parse `NxtBC` as a vector $((\mathrm{hFlag}_1, \mathtt{NxtBC}_1), \ldots, (\mathrm{hFlag}_n, \mathtt{NxtBC}_n))$
5: $\vec{N} \leftarrow \varepsilon$
6: **if** $|\mathtt{state}| \geq \mathtt{windowSize}$ **then**
7: $\quad \tau_{\mathsf{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 1]$
8: **else**
9: $\quad \tau_{\mathsf{low}} \leftarrow 0$
10: **end if**
11: oldValidTxMissing $\leftarrow$ false
12: **for** each list $\mathtt{NxtBC}_i$ of transaction IDs **do**
13: $\quad$ Use the txid contained in $\mathtt{NxtBC}_i$ to determine the list of transactions
14: $\quad$ Let $\vec{\mathtt{tx}} = (\mathtt{tx}_1, \ldots, \mathtt{tx}_{|\mathtt{NxtBC}_i|})$ denote the transactions of $\mathtt{NxtBC}_i$
15: $\quad$ **if** $\mathtt{tx}_1$ is not a coinbase trasnaction **then**
16: $\quad\quad$ **return** $\vec{N}_{\mathrm{df}}$
17: $\quad$ **else**
18: $\quad\quad \vec{N}_i \leftarrow \mathtt{tx}_1$
19: $\quad\quad$ **for** $j = 2$ **to** $|\mathtt{NxtBC}|$ **do**
20: $\quad\quad\quad \mathtt{st}_i \leftarrow \mathsf{Blockify}(\vec{N}_i)$
$\quad\quad\quad$ ▷ Default Extension if proposal violates fair order
21: $\quad\quad\quad$ **if** ValidOrder($\mathtt{tx}_j, \mathtt{state} \,\|\, \mathtt{st}_i, \mathtt{buffer}$) = false **then return** $\vec{N}_{\mathrm{df}}$
$\quad\quad\quad$ ▷ Default Extension if proposal includes invalid transaction

22: **if** $\mathsf{ValidTX}(\mathtt{tx}_j, \mathtt{state} \parallel \mathtt{st}_i) = \mathsf{false}$ then **return** $\vec{N}_{\mathrm{df}}$
23: $\vec{N}_i \leftarrow \vec{N}_i \parallel \mathtt{tx}_j$
24: **end for**
25: $\mathtt{st}_i \leftarrow \mathsf{Blockify}(\vec{N}_i)$
26: **end if**
▷ Test that all old transactions are included.
27: **for** each $\mathtt{BTX} = (\mathtt{tx}, \mathrm{txid}, \tau', \mathsf{P}) \in \mathtt{buffer}$ of an honest party P with time $\tau' < \tau_{\mathcal{L}} - \mathtt{waitTime}$ **do**
28: **if** $\mathsf{ValidTX}(\mathtt{tx}, \mathtt{state} \parallel \mathtt{st}_i) = \mathsf{true}$ **and** $\mathsf{ValidOrder}(\mathtt{tx}, \mathtt{state} \parallel \mathtt{st}_i, \mathtt{buffer}) = \mathsf{true}$ **and** $\mathtt{tx} \notin \vec{N}_i$ **then**
29: $\mathsf{oldValidTxMissing} \leftarrow \mathsf{true}$
30: **end if**
31: **end for**
32: $\vec{N} \leftarrow \vec{N} \parallel \vec{N}_i$
33: $\mathtt{state} \leftarrow \mathtt{state} \parallel \mathtt{st}_i$
34: $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_{\mathcal{L}}$
▷ Must not proceed with too many adversarial blocks
35: $j \leftarrow \max\{\{\mathtt{windowSize}\} \cup \{k \mid \mathtt{st}_k \in \mathtt{state} \wedge$ proposal of $\mathtt{st}_k$ had hFlag $= 1\}\}$
36: **if** $|\mathtt{state}| - j \geq \eta$ **then return** $\vec{N}_{\mathrm{df}}$
37: **if** $|\mathtt{state}| \geq \mathtt{windowSize}$ **then**
38: $\tau_{\mathsf{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 1]$
39: **else**
40: $\tau_{\mathsf{low}} \leftarrow 0$
41: **end if**
42: **end for**
▷ A sequence of blocks cannot take too much time
43: **if** $\tau_{\mathsf{low}} > 0$ **and** $\tau_{\mathcal{L}} - \tau_{\mathsf{low}} > \mathtt{maxTime}_{\mathtt{window}}$ **then return** $\vec{N}_{\mathrm{df}}$
▷ Bootstrapping cannot take too much time
44: **if** $\tau_{\mathsf{low}} = 0$ **and** $\tau_{\mathcal{L}} - \tau_{\mathsf{low}} > 2 \cdot \mathtt{maxTime}_{\mathtt{window}}$ **then return** $\vec{N}_{\mathrm{df}}$
▷ Old enough, valid transactions should be included
45: **if** $\mathsf{oldValidTxMissing}$ **then return** $\vec{N}_{\mathrm{df}}$
46: **return** $\vec{N}$

Transaction leakage after settlement. Recall that while Bitcoin is pseudonymous, all transactions are actually public and transparent (both on-chain and in the mempool). The ledger functionality in [BMTZ17] thus leaks all information to the adversary once it receives a transaction. This gives the adversary the full power to adaptively issue transactions, even within the same round as the victim. In order to capture the goal that the adversary cannot finalize a transaction that depends in a meaningful way of any pending transactions (i.e., input causality, cf. [CKPS01]), the ideal functionality must hide the transaction content and metadata before it gets settled.

We make the following two enhancements to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$. On one hand, upon a transaction is sent to the functionality, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ book-keeps the transaction as $(\mathtt{tx}, \mathrm{txid}, \tau_{\mathcal{L}}, \mathsf{P})$ where $\mathtt{tx}$ denotes the plain transaction (i.e., content and metadata including issuer's signature), txid is a unique random tag generated upon receipt, $\tau_{\mathcal{L}}$ is the time that this transaction is sent and P denotes the sender. Our new functionality $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ immediately leaks transaction length $|\mathtt{tx}|$, identifier txid, receiving

time $\tau_{\mathcal{L}}$ and sender P to the adversary (recall that we assume a network adversary); however, $\mathtt{tx}$ itself remains hidden (note that in [BMTZ17] the entire tuple is leaked immediately). On the other hand, upon the adversary requests to read all transactions so far, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ returns (i) the current state $\mathtt{state}$ where all transactions are transparent (indicating that they have been settled in the immutable ledger); and, (ii) transaction buffer $\mathtt{buffer}$ except that for each transaction tuple $(\mathtt{tx}, \cdot, \cdot, \cdot)$ in $\mathtt{buffer}$, $\mathtt{tx}$ is replaced with its length $|\mathtt{tx}|$ in the response. Precisely, upon receiving the READ command, the functionality returns $(\mathtt{state}, \widehat{\mathtt{buffer}})$ such that

$$\widehat{\mathtt{buffer}} \triangleq \{(|\mathtt{tx}|, \mathrm{txid}, \tau, \mathsf{P}) \mid \mathtt{BTX} = (\mathtt{tx}, \mathrm{txid}, \tau, \mathsf{P}) \in \mathtt{buffer}\}.$$

In such a way, the ideal world adversary is limited to learn only the "existence" of a transaction $\mathtt{tx}$ before its settlement. To learn the full information of $\mathtt{tx}$, the adversary has to either propose $\mathtt{tx}$ in the next block or wait for the default extension policy to handle $\mathtt{tx}$. In other words, the time interval that $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ hides the plain transaction is upper-bounded by liveness; but it is up to the ideal world adversary to decide whether to propose $\mathtt{tx}$ earlier and learn its content, or wait for the default extension. (Arguably, a good simulator never uses the default extension to learn $\mathtt{tx}$; details see the protocol analysis.) Also, notice that the sender P leaks information at the network level and is not necessarily linked with the transaction issuer. Thus $\mathtt{tx}$ remains completely secret before being committed to the ledger state.

4 Extend Policies with Sender Order Fairness

The parametric $\mathsf{ExtendPolicy}$ introduced in Section 3 allows us to insert a customized fair-order policy into $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ such that its output ledger state should follow a legitimate transaction sequence under the designated policy. In this section we consider two policy instances[7]. The first one is the most natural sender-side order fairness $\mathsf{SenderOrder}$ where transactions should be serialized by the order they are received by the ledger $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$. The second policy is a (necessary) relaxation on sender order fairness (namely, δ-$\mathsf{ApproxSenderOrder}$) which asks for the order of two transactions if they reach $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ at times that are δ apart from each other. We discuss impossibility results with respect to both policies in the same setting as Bitcoin (i.e., with global clock and Δ-bounded delay diffusion network). Further, we show that the unfairness stemmed from network delay is impossible to circumvent — we prove that even if we employ additional powerful setups, still no δ-$\mathsf{ApproxSenderOrder}$ can be achieved for any $\Delta < \delta$.

4.1 Sender Order Fairness

Sender-side order fairness captures the natural desire that a ledger $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ serializes transactions based on the time that they are released by their senders. This order is well-defined by looking at the $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ transaction buffer and order transactions based on their arrival time recorded. Hence, every time the adversary tries to propose a transaction $\mathtt{tx}$ that is sent at time τ (i.e., $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ receives $\mathtt{tx}$ and records $(\mathtt{tx}, \mathrm{txid}, \tau, \mathsf{P})$), our $\mathsf{ValidOrder}$ algorithm in $\mathsf{SenderOrder}$ checks whether there exists valid transactions in $\mathtt{buffer}$ with time $\tau' < \tau$. If such transaction exists, the proposed order is invalid and this will force the $\mathsf{ExtendPolicy}$ to return a default extension.

Regarding the $\mathsf{DefaultExtension}$ mechanism, since the original construction in [BMTZ17] has been proposing a default order that follows the one stored in the ledger, we follow that construction with one minor modification. Precisely, in [BMTZ17] a default extension will blockify all pending

[7] Capturing receiver order fairness definitions, e.g., [KZGJ20, CMSZ22], can be an interesting direction for future work.

transactions in the buffer, while ours only include those that are about to violate liveness — again this adaption follows our blockchain scheme that decouples the consensus and transaction inclusion. We note that this adaption is insignificant, as a good simulator shall never allow the ledger to trigger this subroutine.

Algorithm 2 SenderOrder

function ValidOrder($\mathtt{tx}$, $\mathtt{state}$, $\mathtt{buffer}$)
  Parse $\mathtt{tx}$ as $(\mathtt{tx}, \mathrm{txid}, \tau, \mathsf{P})$
  ▷ Extract the most recent transaction time in the ledger state.
  $\tau^* \leftarrow \max\{\tau' \mid \mathtt{BTX} = (\mathtt{tx}, \cdot, \tau', \cdot) \wedge \mathtt{tx} \in \mathtt{state}\}$
  $\vec{\mathtt{tx}} \leftarrow \{\mathtt{tx} \mid \mathtt{BTX} = (\mathtt{tx}, \cdot, \tau', \cdot) \in \mathtt{buffer} \wedge \mathtt{BTX} \notin \vec{N} \wedge \tau^* \leq \tau' < \tau\}$
  for $\mathtt{tx} \in \vec{\mathtt{tx}}$ **do**
    if ValidTX($\mathtt{tx}, \mathtt{state}$) = true **then return** false
  end for
  return true
end function

function DefaultExtension($\vec{\mathcal{I}}_H^T$, $\mathtt{state}$, $\mathtt{NxtBC}$, $\mathtt{buffer}$, $\vec{\tau}_{\mathtt{state}}$)
  $\tau_{\mathcal{L}} \leftarrow$ current ledger time (computed from $\vec{\mathcal{I}}_H^T$)
  $\vec{N}_{\mathrm{df}} \leftarrow \mathtt{tx}_{\mathrm{minerID}}^{\mathrm{base-tx}}$ of an honest miner
  Let $\vec{\mathtt{tx}} = \{\mathtt{tx} \mid \mathtt{BTX} = (\mathtt{tx}, \mathrm{txid}, \tau'_{\mathcal{L}}, \mathsf{P}) \in \mathtt{buffer} \wedge \tau'_{\mathcal{L}} < \tau_{\mathcal{L}} - \mathtt{waitTime}\}$
  Sort $\vec{\mathtt{tx}}$ according to timestamps
  $\mathtt{st} \leftarrow$ Blockify($\vec{N}_{\mathrm{df}}$)
  repeat
    Let $\vec{\mathtt{tx}} = (\mathtt{tx}_1, \ldots, \mathtt{tx}_\ell)$ be the current set of (remaining) transactions
    for $i = 1$ **to** ℓ **do**
      if ValidTX($\mathtt{tx}_i, \mathtt{state} \parallel \mathtt{st}$) = true **and** ValidOrder($\mathtt{tx}_i, \mathtt{state} \parallel \mathtt{st}, \mathtt{buffer}$) = true **then**
        $\vec{N}_{\mathrm{df}} \leftarrow \vec{N}_{\mathrm{df}} \parallel \mathtt{tx}_i$
        Remove $\mathtt{tx}_i$ from $\vec{\mathtt{tx}}$
        $\mathtt{st} \leftarrow$ Blockify($\vec{N}$)
      end if
    end for
  until $\vec{N}_{\mathrm{df}}$ does not increase any more
  if $|\mathtt{state}| + 1 \geq \mathtt{windowSize}$ **then**
    $\tau_{\mathsf{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 2]$
  else
    Set $\tau_{\mathsf{low}} \leftarrow 0$
  end if
  $c \leftarrow 1$
  while $\tau_{\mathcal{L}} - \tau_{\mathsf{low}} > \mathtt{maxTime}_{\mathtt{window}}$ **do**
    Set $\vec{N}_c \leftarrow \mathtt{tx}_{\mathrm{minerID}}^{\mathrm{base-tx}}$ of an honest miner
    $\vec{N}_{\mathrm{df}} \leftarrow \vec{N}_{\mathrm{df}} \parallel \vec{N}_c$
    $c \leftarrow c + 1$
    if $|\mathtt{state}| + c \geq \mathtt{windowSize}$ **then**
      $\tau_{\mathsf{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + c + 1]$

```
        else
            τ_low ← 1
        end if
    end while
    return N⃗_df
end function
```

The sender fairness SenderOrder is our desideratum; unfortunately, SenderOrder is impossible to realize, even in the synchronous network (i.e., $\Delta = 1$ in $\mathcal{F}_{\mathsf{Diffuse}}^{\Delta}$). The high level intuition is that transactions issued by corrupted parties can always reach honest parties in two consecutive rounds by sending to one honest party P at round r and let P diffuse it to others at round $r+1$, where in the case of honest party issuing transactions, the sender can book-keep their receiving time to the next round. The fact that the adversary $\mathcal{A}$ can exploit this one round advantage allows him to make the following two executions identical: in the first execution, $\mathcal{A}$ sends a transaction $\mathtt{tx}$ at round r to only one honest party and in the second execution he sends $\mathtt{tx}$ to all honest parties at round $r+1$. Later, in the second execution $\mathcal{A}$ lets one corrupted party behave like it receive and diffuses the message at round r. In the first execution, $\mathtt{tx}$ in $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ owns timestamp r while in the second execution it owns timestamp $r+1$, however these two executions are indistinguishable.

4.2 Approximate Sender Order Fairness

Given that SenderOrder is unachievable unless under unrealistic network assumptions (that is, the delivery of a transaction is instant and before any transaction that is issued later thus every honest parties see the same sequence of transactions), we next propose a natural alternative that relaxes the sender order fairness, giving up on asking for the order of two transactions if they enter the network at approximately the same time.

We leave the DefaultExtension procedure the same as that in SenderOrder where an order following what $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ receives is proposed (hence we omit it in the algorithm description). Note that such default extension will still yield a legitimate order with respect to δ-ApproxSenderOrder. Regarding the ValidOrder algorithm, when the adversary is to propose a transaction $\mathtt{tx}$ sending the ledger at time τ, all transactions in the buffer which are sent at time before $\tau - \delta$ and are valid with respect to state should be proposed before $\mathtt{tx}$; otherwise, ValidOrder returns false to instruct the ledger to use the DefaultExtension.

Algorithm 3 δ-ApproxSenderOrder

```
function ValidOrder(tx, N⃗, buffer)
    Parse tx as (tx, txid, τ, P)
    ▷ Extract the most recent transaction time in the ledger state.
    τ* ← max{τ' | BTX = (tx, ·, τ', ·) ∧ tx ∈ state}
    t⃗x ← {BTX = (·, ·, τ', ·) | BTX ∈ buffer ∧ BTX ∉ N⃗ ∧ τ* ≤ τ' < τ − δ}
    for tx ∈ t⃗x do
        if ValidTX(tx, state) = true then return false
    end for
    return true
end function
```

Impossibility of δ-Approximate Order with Δ network latency for $\delta < \Delta$. We show that in the same setting as Bitcoin (i.e., with $\mathcal{G}_{\mathsf{Clock}}$, $\mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}})$ and $\mathcal{F}_{\mathsf{Diffuse}}$), it is impossible to guarantee fairness for two transactions $\mathtt{tx}, \mathtt{tx}'$ that enters the system less than δ rounds apart from each other for any $\delta < \Delta$.

Theorem 1. *In the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}), \mathcal{F}_{\mathsf{Diffuse}}^{\Delta})$-hybrid environment, there exist no protocol Π that securely realizes $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ parameterized with δ-ApproxSenderOrder for any $\delta < \Delta$.*

Proof. Suppose towards a contradiction, there exists a protocol Π that securely realizes $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ parameterized with δ-ApproxSenderOrder in the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}), \mathcal{F}_{\mathsf{Diffuse}}^{\Delta})$-hybrid environment.

Consider two transactions $\mathtt{tx}, \mathtt{tx}'$. Let τ, τ' denote the timestamp that $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ assigns to them in the transaction buffer respectively. Consider a world W_1 where $\mathtt{tx}$ is sent more than δ yet less than Δ rounds before $\mathtt{tx}'$ (i.e., $\tau + \delta < \tau' < \tau + \Delta$). I.e., δ-ApproxSenderOrder would force $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ to output $\mathtt{tx} \prec \mathtt{tx}'$. Regarding the transaction diffusion pattern in W_1, assume all but the sender of $\mathtt{tx}$ receives $\mathtt{tx}$ at time $\tilde{\tau}$ such that $\tau' < \tilde{\tau} < \tau + \Delta$; and all but the sender of $\mathtt{tx}'$ receives $\mathtt{tx}'$ at time $\tilde{\tau}'$ such that $\tau' \leq \tilde{\tau}' < \tilde{\tau}$. I.e., for all parties except the senders, they saw $\mathtt{tx}' \prec \mathtt{tx}$ unanimously. Then, we consider the following three scenarios.

First, neither the sender of $\mathtt{tx}$ nor the sender of $\mathtt{tx}'$ has the chance to send a message. We prove that for all other protocol participants in W_1 this is indistinguishable from the world W_2 where $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ shall output $\mathtt{tx}' \prec \mathtt{tx}$. Consider the following scenario in W_2: two transactions $\mathtt{tx}, \mathtt{tx}'$ are sent at time τ', τ respectively and are delivered to all other parties at time $\tilde{\tau}$, $\tilde{\tau}'$ respectively. Due to δ-ApproxSenderOrder, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ outputs $\mathtt{tx}' \prec \mathtt{tx}$. The local views of all other parties are identical to the ones in W_1 hence the messages they send are also identical.

Second, at least one of the two senders of $\mathtt{tx}, \mathtt{tx}'$ successfully sends at least one message. Note that this implies Π is a protocol such that the pairwise order of any two transactions could be dominated by a single party (or, a tiny fraction of resources to run the protocol). Again we show that W_1 is indistinguishable from the world W_3 where $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ shall output $\mathtt{tx}' \prec \mathtt{tx}$. Suppose, with out loss of generality, $\mathtt{tx}$ is generated by a corrupted party. Consider the following scenario in W_3: the transaction timestamp in $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ and diffusion pattern are exactly the same as those in W_2 except that now the corrupted issuer of $\mathtt{tx}$ sends a message claiming that $\mathtt{tx}$ is sent at time τ (this is a valid message even if Δ is known and parties use it to filter old messages). If in W_1 the issuer of $\mathtt{tx}$ sends the same message that $\mathtt{tx}$ is generated at time τ, then in all honest parties' local views, W_3 is identical to the scenario in W_1 and they send the identical messages.

Finally, consider the case where both transaction issuers send messages claiming their transaction time. We show W_1 is indistinguishable from the world W_4 where $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ shall output $\mathtt{tx}' \prec \mathtt{tx}$. Consider the following scenario in W_4: the transaction issuer of $\mathtt{tx}$ is corrupted and $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ assigns timestamp $\tau', \tau - \epsilon$ to transaction $\mathtt{tx}$ and $\mathtt{tx}'$ respectively. In addition, $\mathtt{tx}, \mathtt{tx}'$ are diffused to all parties except their senders at at the same time as those in W_1; the (honest) issuer of $\mathtt{tx}'$ sends messages saying that $\mathtt{tx}'$ is sent at time $\tau - \epsilon$ and the corrupted issuer of $\mathtt{tx}$ sends messages claiming $\mathtt{tx}$ is sent at time τ. Assume in W_1, the issuer of $\mathtt{tx}'$ is corrupted; then, the honest issuer of $\mathtt{tx}$ sends messages claiming that $\mathtt{tx}$ is sent at time τ and the corrupted issuer of $\mathtt{tx}'$ sends messages claiming that $\mathtt{tx}'$ is sent at time $\tau - \epsilon$. In such a case, the diffusion pattern and messages sent in both W_1 and W_4 are identical. □

Approximate order with any private $\mathcal{F}$. Trusted hardware may be considered as a promising solution to transaction order fairness, as they are promised to help "authenticate and timestamp" a transaction that is generated even by a malicious host. Despite the fact that trusted execution environments are hard to synchronize, we show that when network delay exists, it leads to the

imprecision on order fairness that cannot be circumvented by any powerful time-aware private functionalities.

We consider the model where there exists a functionality $\mathcal{F}_i$ for each party P_i which, can only communicate with P_i, global clock $\mathcal{G}_{\mathsf{Clock}}$ and a correlated randomness functionality $\mathcal{G}_{\mathsf{CR}}^{\mathcal{D}}$; however $\mathcal{F}_i$ can function arbitrarily. Specifically, the correlated randomness setup $\mathcal{G}_{\mathsf{CR}}^{\mathcal{D}}$ returns a random string s from some distribution $\mathcal{D}$ to a functionality $\mathcal{F}$ only when $\mathcal{F}$ owns the code as specified in code in $\mathcal{G}_{\mathsf{CR}}^{\mathcal{D}}$; in such a way only the designated $\mathcal{F}$ can acquire the randomness. This can be achieved, by considering the request message of format (cf. UC with global subroutine in [BCH+20]) $(\mu, (\mathrm{sid}, \mathrm{pid}))$ where μ specifies the sender's code.

Functionality $\mathcal{G}_{\mathsf{CR}}^{\mathcal{D}}$

Upon receiving $\mathsf{eid} = (\mu, (\mathrm{sid}, \mathrm{pid}))$ from P:
- If $\mu \neq \mathsf{code}$ ignore this message.
- If there is no tuple of the form $(\mathrm{sid}, \ldots)$, generate $(s_1, \ldots, s_n) \leftarrow \mathcal{D}(1^\lambda))$ and store $(\mathrm{sid}, s_1, \ldots, s_n)$. Otherwise, retrieve $(\mathrm{sid}, s_1, \ldots, s_n)$.
- Send Send (sid, s_i) to P_i.

Given $\mathcal{G}_{\mathsf{CR}}^{\mathcal{D}}$, all responses parties acquired from $\mathcal{F}$ can be verified by other parties (in other words, they can be "convinced" by the messages from $\mathcal{F}$). We showcase that even if $\mathcal{F}$ can implement an arbitrary function and is time-aware (i.e., $\mathcal{F}$ learns the global time from $\mathcal{G}_{\mathsf{Clock}}$), this does not help circumvent the impossibility result from the network delay. The subtlety in our proof, at a high level, is that corrupted parties can keep querying his local $\mathcal{F}$ for the same message in every round via different corrupted parties. We formalize this result in Theorem 2.

Theorem 2 (Impossibility of δ-approximate order with any private $\mathcal{F}$). *If $\Delta > 0$, then for any $\delta < \Delta$ there exist no protocol Π that securely realizes $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ parameterized with δ-ApproxSenderOrder in the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}), \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{G}_{\mathsf{CR}}^{\mathcal{D}}, \mathcal{F})$-hybrid environment.*

Proof. Suppose towards a contradiction, there exists a protocol Π that securely realizes $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ parameterized with δ-ApproxSenderOrder in the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}), \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{G}_{\mathsf{CR}}^{\mathcal{D}}, \mathcal{F})$-hybrid environment.

Consider two transactions $\mathtt{tx}, \mathtt{tx}'$ in scenario W_1. Let τ, τ' denote the timestamp that $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ assigns to them in the transaction buffer respectively. With out loss of generality, assume $\tau + \delta < \tau' < \tau + \Delta$ and $\mathtt{tx}'$ is generated by the adversary. Due to δ-ApproxSenderOrder, $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ should output $\mathtt{tx} \prec \mathtt{tx}'$. We show this is indistinguishable from the scenario W_2 where $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ shall output $\mathtt{tx}' \prec \mathtt{tx}$. Before we go to W_2, consider the responses from $\mathcal{F}$ in W_1. The issuer of $\mathtt{tx}$, when sending $\mathtt{tx}$, can attach the message that he queries his own $\mathcal{F}$ at round τ; the (corrupted) issuer of $\mathtt{tx}'$, when sending $\mathtt{tx}$', however can attach the message that he queries his own $\mathcal{F}$ at round τ (but not τ'). This is because the adversary can keep querying $\mathtt{tx}'$ to $\mathcal{F}$ in every round up to τ' (which includes round $\tau < \tau'$). Since in each round, the adversary can query $\mathcal{F}$ via different corrupted parties (and these parties behave honestly for all other operations) the message that he acquired from $\mathcal{F}$ in round τ is admissible by all honest parties.

Now consider the following scenario W_2: $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ assigns timestamp τ', τ to transaction $\mathtt{tx}$ and $\mathtt{tx}'$ respectively, and the transaction issuer of $\mathtt{tx}$ is corrupted. In addition, $\mathtt{tx}, \mathtt{tx}'$ are diffused to all parties except their senders at at the same time in W_1 and W_2 (this diffusion pattern is possible, see the proof of Theorem 1). In W_2, the (honest) issuer of $\mathtt{tx}'$ queries its private $\mathcal{F}$ the same time

as it was sent messages hence get a response with respect to time τ; meanwhile the corrupted issuer of $\mathtt{tx}$ keeps querying $\mathtt{tx}$ and diffuses $\mathtt{tx}$ with a response with respect to time τ at round τ'.

We learn that the diffusion pattern and the responses from $\mathcal{F}$ sent in both scenarios are identical, thus parties must decide the same order, which contradicts the fact that the protocol realizes δ-$\mathsf{ApproxSenderOrder}$. □

5 Protocol Details

We describe our protocol $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ in this section. The technical roadmap mainly consists two parts — transaction encryption and decryption using enclaves (in Section 5.1) and encrypted-transaction serialization (in Section 5.2). A full code specification of our protocol is presented in Appendix C. For the ease of presentation, we describe our protocol in the PoW and static setting (where the computational power is fixed yet known), we discuss how our protocol can be translated to different settings (PoS and dynamic participation) in Section 5.3.

5.1 Transaction Encryption from Trusted Hardware

Our protocol follows the classical setting of state machine replication, where two types of parties — maintainers (miners/validators) and clients — are considered as protocol participants. We assume that all protocol maintainers are equipped with their own trusted hardware[8]. In contrary, a client, who sends transactions to the maintainers to use the ledger, is not equipped with an enclave.

The trusted hardware functionality $\mathcal{G}_{\mathsf{att}}$. Following the model and treatment by Pass *et al.* [PST17], we capture the maintainers' enclaves as a shared setup $\mathcal{G}_{\mathsf{att}}$. This enclave functionality $\mathcal{G}_{\mathsf{att}}$ is parameterized with a signature scheme SIG and a registered set of parties $\mathcal{P}$. Upon initialization, a master signing/verification key pair $(\mathsf{msk}, \mathsf{mvk})$ is generated, and parties can query the verification key via the GET-VK interface (i.e., it is assumed that in the real world such master verification key is securely distributed).

$\mathcal{G}_{\mathsf{att}}$ has two "local" interfaces INSTALL and RESUME, and maintains a table T to record the installed program and their associated parties and internal states. Specifically, every registered party can install a program that can later be resumed by herself. Each program has its own memory which can only be reached and modified by the program itself. After resuming a program, the enclave signs the output and the executed program (with session id) using the master signing key msk; later, any party can verify the output by using the master verification key queried from $\mathcal{G}_{\mathsf{att}}$.

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

The functionality is parameterized with a signature scheme $\mathsf{SIG} = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and a registered party set $\mathcal{P}$. On initialization, $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{SIG.KeyGen}(1^\kappa)$ and $T \leftarrow \emptyset$.

Master key query. Upon receiving (GET-VK) from P, send (GET-VK, mpk) to P.

Install an enclave. Upon receiving (INSTALL, idx, prog) from some $\mathsf{P} \in \mathcal{P}$:
1. If P is honest, assert $idx = sid$.
2. Generate nonce $eid \in \{0,1\}^\kappa$, store $T[eid, \mathsf{P}] = (idx, \mathsf{prog}, \vec{0})$, send eid to P.

[8]Note that owning an enclave has no implication on the maintainer's computational power. Our protocol works as long as the majority of the computational power is honest and a constant fraction of the enclaves are controlled by honest maintainers.

Resume an enclave. Upon receiving (RESUME, eid, inp) from some $\mathsf{P} \in \mathcal{P}$:
1. Let $(idx, \mathsf{prog}, \mathsf{mem}) = T[eid, \mathsf{P}]$, abort if not found.
2. Let $(\mathsf{outp}, \mathsf{mem}) = \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$, update $T[eid, \mathsf{P}] = (idx, \mathsf{prog}, \mathsf{mem})$.
3. Let $\sigma = \mathsf{SIG.Sign}(\mathsf{msk}, idx, eid, \mathsf{prog}, \mathsf{outp})$, and send (outp, σ) to P.

Transaction encryption scheme. We detail the code that $\mathcal{G}_{\mathsf{att}}$ is expected to execute in TxDec (Program 1). This program is parameterized with a public key encryption scheme PKE and $\Lambda \in \mathbb{N}^+$ which indicates the time interval for hiding the transaction information and is counted by the number of blocks. TxDec has two interfaces, both can be executed for multiple times. The first interface KEY-GEN returns the public key pk in storage. When called for the first time, it runs the key generation algorithm of PKE and stores the key pair $(\mathsf{pk}, \mathsf{sk})$. I.e., for each session, one enclave can only hold one public key pair. The second interface TX-DEC, taking input a ciphertext ct and a chain $\mathcal{C}$, decrypts ct to a plain transaction tx when all the following conditions are satisfied: (i) ct successfully decrypts to $\mathtt{tx} \,\|\, h$ where h is a valid block state in $\mathcal{C}$; and (ii) the length of $\mathcal{C}$, starting from the block with hash h has progressed for at least Λ blocks.

Importantly, in TX-DEC there is a backdoor that can be triggered only when the decrypted transaction is an *all-zero* string. This backdoor, when triggered, re-decrypts the ciphertext using an alternate decryption algorithm Dec^* (which we detail very soon below). Looking ahead, this backdoor enables the simulator to equivocate fake transactions, as Dec^* allows for the adversary to program the random oracle and decrypt ct to an arbitrary string he chooses. Note that while every party can exploit this backdoor and equivocate her transaction, this is indeed harmless in the real world, because in order for a transaction to be considered legitimate, it should be proved by NIZKPoK as a valid transaction; though, an all-zero string can never be proved. In the ideal world, the simulator can generate a fake NIZK proof by hiding the information that random oracle is programmed from the black-box adversary, thus convincing him that an all-zero string is a valid transaction.

Program 1 $\mathsf{TxDec}(\mathsf{PKE}, \Lambda)$

Key generation. Upon receiving (KEY-GEN):
1. If $(\mathsf{sk}, \mathsf{pk})$ is not stored, generate a key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.KeyGen}(1^\kappa)$ and store $(\mathsf{sk}, \mathsf{pk})$.
2. Return (KEY-GEN, pk).

Transaction Decryption. Upon receiving (TX-DEC, ct, $\mathcal{C}$):
1. Compute $\mathtt{tx} \,\|\, h \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}, ct)$; abort if decryption fails.
2. If $\nexists \mathcal{B} \in \mathcal{C}$ such that $H(\mathcal{B}) = h$, abort.
3. Let ℓ denote the block height of $\mathcal{B}$, abort if $\ell + \Lambda < \mathrm{len}(\mathcal{C})$.
4. If $\mathtt{tx} = 0^{|\mathtt{tx}|}$, $\mathtt{tx} \,\|\, h \leftarrow \mathsf{PKE.Dec}^*(\mathsf{sk}, ct)$
5. Return (TX-DEC, tx, ct).

Next we introduce the public-key encryption scheme PKE to be used in the TxDec program, consisting of four algorithms KeyGen, Enc, Dec and Dec^*. It is parameterized with a collection of one-way trapdoor functions and the restricted programmable and observable global random oracle $\mathcal{G}_{\mathsf{rpoRO}}$ introduced by Camenish *et al.* [CDG+18]. Specifically, a trapdoor one-way function in $\mathcal{F}_{\mathsf{owf}}$ consists of three functions $\mathsf{f}, \mathsf{f}^{-1}$ and b which is the hardcore predicate of f.

– The key generation KeyGen algorithm randomly selects a trapdoor one-way function from $\mathcal{F}_{\mathsf{owf}}$,

and returns (f, b) as the public key and $(\mathsf{f}, \mathsf{f}^{-1}, \mathsf{b})$ as the secret key.

- The encryption algorithm Enc takes a k-bit message as input. For the i-th bit, it samples a random element x_i in the input domain of f and computes ciphertext ct_i by concatenating $\mathsf{f}(x_i)$ and $\mathsf{b}(x_i) \oplus m_i$. After computing all ct_i it outputs the ciphertext $ct = ct_1, \ldots, ct_k$.
- The decryption algorithm Dec to compute a k-bit plaintext works as follows. For the i-th bit and its corresponding ciphertext piece ct_i, m_i is computed by first extracting $x_i = \mathsf{f}^{-1}(ct_i')$ (ct_i' denotes all but the last bit), and then $\mathsf{b}(x_i)$ is XORed with b_i' which is the last bit of ct_i to get m_i'. The plaintext is decrypted by concatenating all m_i'.
- The alternate decryption algorithm Dec^* works exactly the same as Dec, except that it replaces the computation of hardcore bit $\mathsf{b}(x_i)$ with the random oracle query, and the response is XORed with b_i' (where last bit is used as result).

Algorithm 4 $\mathsf{PKE}(\mathcal{F}_{\mathsf{owf}}, \mathcal{G}_{\mathsf{rpoRO}})$

Let $\mathcal{F}_{\mathsf{owf}} = \{\mathsf{f}_i : \mathcal{D}_i \to \mathcal{C}_i\}_{i \in I}$ be a collection of one-way trapdoor functions, where I and all $\mathcal{D}_i, \mathcal{C}_i$ have cardinality at least 2^κ. Let $\mathsf{f}_i^{-1} : \mathcal{C}_i \to \mathcal{D}_i$ denote the trapdoor function with respect to f_i, and $\mathsf{b}_i : \mathcal{D}_i \to \{0,1\}$ denote the hardcore predicate of f_i.

function $\mathsf{KeyGen}(1^\kappa)$
  Sample $i \xleftarrow{\$} I$.
  Set $\mathsf{pk} \leftarrow (\mathsf{f}_i, \mathsf{b}_i)$ and $\mathsf{sk} \leftarrow (\mathsf{f}_i, \mathsf{f}_i^{-1}, \mathsf{b}_i)$.
  Return $(\mathsf{pk}, \mathsf{sk})$.
end function

function $\mathsf{Enc}(\mathsf{pk}, m)$
  Parse pk as (f, b) and m as $m_1, \ldots, m_k$.
  for i **from** 1 **to** k **do**
    Set $x_i \xleftarrow{\$} \mathcal{D}$
    Set $ct_i \leftarrow \mathsf{f}(x_i) \,\|\, (\mathsf{b}(x_i) \oplus m_i)$
  end for
  Set $ct \leftarrow ct_1, \ldots, ct_k$ and return ct.
end function

function $\mathsf{Dec}(ct)$
  Parse sk as $(\mathsf{f}_i, \mathsf{f}_i^{-1}, \mathsf{b}_i)$ and ct as $ct_1, \ldots, ct_k$.
  for i **from** 1 **to** k **do**
    Parse ct_i as $ct_i' \,\|\, b_i'$
    Compute $x_i' \leftarrow \mathsf{f}_i^{-1}(ct_i')$ and set $m_i' \leftarrow \mathsf{b}(x_i') \oplus b_i'$
  end for
  Set $m' \leftarrow m_1', \ldots, m_k'$ and return m'.
end function

function $\mathsf{Dec}^*(ct)$
  Parse sk as $(\mathsf{f}_i, \mathsf{f}_i^{-1}, \mathsf{b}_i)$ and ct as $ct_1, \ldots, ct_k$.
  for i **from** 1 **to** k **do**
    Parse ct_i as $ct_i' \,\|\, b_i'$
    Compute $x_i' \leftarrow \mathsf{f}_i^{-1}(ct_i')$ and set $m_i' \leftarrow h \oplus b_i'$ where $(\textsc{eval}, h) = \mathcal{G}_{\mathsf{rpoRO}}(\textsc{eval}, x_i')$
  end for
  Set $m' \leftarrow m_1', \ldots, m_k'$ and return m'.
end function

We note that the first three algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ in PKE has actually instantiated a CPA-secure public-key encryption scheme. Nonetheless, the subtlety of using PKE in our protocol is that the secret key is stored in the enclave, and the simulator cannot control the parameter that the black-box adversary would like to query $\mathcal{G}_{\mathsf{att}}$. Hence, equivocation in the ideal world should happen in a delicate fashion — by exploiting the last algorithm Dec^*, it provides a simple way for equivocating dummy ciphertexts without letting the enclave to directly interact with the simulator.

Remark 1. The PKE scheme can also be instantiated more efficiently using one-way trapdoor permutation. In particular, we can sample n hard-core bits from just one element x of the function domain by computing $\mathsf{b}(x) \parallel \mathsf{b}(\mathsf{f}(x)) \parallel \ldots \parallel \mathsf{b}(\mathsf{f}^{n-1}(x))$ [KO04].

Looking ahead, in the ideal world, the simulator can equivocate honest transaction by using the fact that only the senders (which are now dummy party) are allowed to equivocate their transactions and all other partiess cannot distinguish whether a response $(\text{TX-DEC}, m, ct)$ from $\mathcal{G}_{\mathsf{att}}$ is decrypted by using Dec or Dec^*.

Lemma 3. *No PPT distinguisher D, after observing polynomially many (m, ct), can tell whether m is decrypted from ct by using $\mathsf{PKE.Dec}$ or $\mathsf{PKE.Dec}^*$ with non-negligible probability.*

Proof (Sketch). To prove the lemma we show that giving only the ciphertext and its decryption (we consider one bit message),

$$\begin{aligned}|\mathbf{Pr}_{b\leftarrow\{0,1\}}[\mathcal{A}(\mathsf{Enc}(b), \mathsf{Dec}(\mathsf{Enc}(b))) = 1] \\ - \mathbf{Pr}_{b\leftarrow\{0,1\}}[\mathcal{A}(\mathsf{Enc}(b), \mathsf{Dec}^*(\mathsf{Enc}(b))) = 1]| < \mathsf{negl}(\kappa).\end{aligned}$$

We first consider a new distinguisher D_1 trying to tell whether a plaintext is decrypted using $\mathsf{PKE.Dec}$ or $\mathsf{PKE.Dec}'$ where in $\mathsf{PKE.Dec}'$, the challenger returns decryption computed by randomly sample a bit and XOR it with the last bit of $\mathsf{Enc}(b)$ (and he remembers this bit for later queries). We show that D_1 cannot distinguish with non-negligible probability, by showing a reduction to the security game of the hardcore bit. Roughly speaking, the adversary in the hardcore bit game runs D_1; if D_1 decides on $\mathsf{PKE.Dec}$, then the adversary picks one (m, ct) outputs m XOR the last bit of c. If D_1 distinguishes with non-negligible probability, then the adversary finds $\mathsf{b}(x)$ of $\mathsf{f}(x)$ with probability non-negligibly more than one half, contradicting the security of hard-core bit.

We consider the next distinguisher D_2 trying to tell whether a plaintext is decrypted using $\mathsf{PKE.Dec}^*$ or $\mathsf{PKE.Dec}'$. This is true because when D_2 is given a ciphertext $ct = y \parallel b$, he cannot check whether the point $\mathsf{f}^{-1}(y)$ is programmed in $\mathcal{G}_{\mathsf{rpoRO}}$. Otherwise, a reduction to the security of one-way function can be made where the distinguisher of one-way function can invert and find preimage with non-negligible probability. By combining these two we conclude the proof. □

Enclave public-key selection. Since all maintainers are equipped with trusted hardware and each holds a public key, in the sense of communication complexity it is infeasible for a client to encrypt transactions to all the enclaves. To reduce the size of an encrypted transaction, the clients should pick a small subset of all public keys and only encrypt to these specific enclaves.

Arguably, as long as the client sends an encrypted transaction `tx` to at least one honest maintainer, it guarantees the correct and timely decryption of `tx`. Nonetheless, it is a non-trivial task to design a key subset selection scheme and the naïve solution to let clients pick public keys by themselves does not work. The reason is that when a transaction issuer takes full control over selecting public keys to encrypt to, the adversary can collude with her and select only the public keys of enclaves that are controlled by the adversary. Especially, the adversary can create different

transactions and release them simultaneously to the network; later, he selectively opens one of them that profits him the most[9].

To address this dilemma between the size of encrypted transactions and the security against selective decryption, we propose a new random subset selection scheme using $\mathcal{G}_{\mathsf{att}}$. Specifically, protocol maintainers run a program $\mathsf{BlockTicket}$ parameterized with a pseudorandom function f_{PRF}. This program has only one interface which takes a κ-bit string h as input. Upon queried, it returns $f_{\mathsf{PRF}}(\mathsf{key}, h)$ where key is a stored PRF key (which is sampled uniformly upon initialization). For every maintainer, once they receive a valid block $\mathcal{B}$ with hash h, they generate a ticket (h, τ) and diffuse it to the network.

Program 2 $\mathsf{BlockTicket}(f_{\mathsf{PRF}})$

Ticket generation. Upon receiving $(\text{TICKET-GEN}, h)$:

1. If key is not stored, generate $\mathsf{key} \xleftarrow{\$} \{0,1\}^\kappa$ and store key.
2. Set $\tau \leftarrow f_{\mathsf{PRF}}(\mathsf{key}, h)$ and return $(\text{TICKET-GEN}, (h, \tau))$.

We then consider a function $f_{\mathsf{select}}(\mathcal{S}, m, \tau)$ which, taking a set $\mathcal{S}$, a threshold $m \in \mathbb{N}^+$ and $\tau \in \{0,1\}^\kappa$ as parameters, outputs an element in $\mathcal{S}$. Formally, let $\binom{\mathcal{S}}{m}$ denote the the (ordered) set of all m-combinations of $\mathcal{S}$ (when $|\mathcal{S}| < m$, f_{select} returns $\mathcal{S}$ directly), we define the selection function as

$$f_{\mathsf{select}} = \binom{\mathcal{S}}{m}\left[\tau \mod \left|\binom{\mathcal{S}}{m}\right|\right].$$

Looking ahead, when a client is to issue a transaction $\mathtt{tx}$, she picks a block hash h on (the settled part of) her local chain $\mathcal{C}_{\text{loc}}$ and encrypts $\mathtt{tx}$ concatenated with h. Then, she selects a ticket (h, τ) (from the set of all tickets $(h, \cdot)$ that she has) and applies f_{select} on the set of public keys published on the blockchain up to block with hash h, $m = \mathsf{polylog}\kappa$ which will be a parameter of the protocol and the ticket (h, τ).

We provide some intuition on why this block ticket scheme underpins a secure subset selection over the public keys. Notice that $m = \mathsf{polylog}\kappa$ and the public keys of enclaves controlled by honest parties account for a constant fraction of all the public keys. By randomly selecting an element in $\binom{\mathcal{S}}{m}$, the probability that the chosen m-combination contains no honest public key is negligible with respect to the security parameter κ. Furthermore, since only a ticket that is associated with a settled block will be used by parties to select public keys (otherwise the encrypted transaction will be considered illegitimate), the total number of tickets that are eligible to use throughout an execution running for $L = \mathsf{poly}(\kappa)$ rounds is polynomially bounded by κ and n where n is the number of enclaves. Refer to Lemma 6 for the formal analysis of this scheme.

Remark 2. Our public-key subset selection scheme still allows transaction issuers to select the ticket that she prefers. I.e., when the issuer chooses not to trust some enclaves, she can select tickets that does not contain their public keys. This can be useful in the model where some enclaves can be compromised, but later honest parties get notified and avoid using their public keys. In such a way the system self-heals from temporarily losing input causality.

Issuing a transaction. We elaborate on how a client can issue a transaction, the code description of this procedure is presented in Algorithm 7 in Appendix C. Recall that clients are not equipped

[9]This is also known as the selective opening problem, which is a major issue when using commitment to hide transaction contents. While this problem can be partially mitigated by employing a penalty scheme, under certain MEV circumstances the revenue of selective opening can be much higher than the penalty.

with trusted hardware thus they should encrypt their transactions using the maintainers' public keys. They can acquire the master verification key mvk from $\mathcal{G}_{\mathsf{att}}$, and the maintainers will publish their public key on the blockchain. Further, by listening to the network they get tickets for every block in the settled part of the blockchain[10].

In order to send a transaction $\mathtt{tx}$, the client first picks a block $\mathcal{B}$ in the settled part of her local chain; i.e., $\mathcal{B} \leftarrow \mathsf{head}(\mathcal{C}_{\mathrm{loc}}^{\lceil k})$. Let h denote the hash of $\mathcal{B}$ and $\mathcal{S}_h$ the set of tickets associated with h. The client first picks a $(h, \tau) \in \mathcal{S}_h$ randomly, and then selects the public keys $\mathcal{S}_{\mathsf{pk}}^{\tau} \leftarrow f_{\mathsf{select}}(\mathcal{S}_{\mathsf{pk}}, m, \tau)$ where $\mathcal{S}_{\mathsf{pk}}$ denotes the set of on-chain enclave public keys up to block $\mathcal{B}$. Let $m' = |\mathcal{S}_{\mathsf{pk}}^{\tau}|$. The client then encrypts $\mathtt{tx}$ as $ct = (ct_1, ct_2, \ldots, ct_{m'})$ such that

$$ct_i = \mathsf{PKE.Enc}(\mathsf{pk}_i, (\mathtt{tx} \,\|\, h)) \text{ where } \mathsf{pk}_i = \mathcal{S}_{\mathsf{pk}}^{\tau}[i]$$

using the encryption algorithm in Algorithm 4. Note that $\mathtt{tx}$ denotes both the transaction content and metadata and they are all encrypted[11], hence each ct_i is indistinguishable from a random string and can only be decrypted using the trusted hardware with public key pk_i.

In order to prevent the adversary from issuing "bad" transactions (e.g., different ct_i encrypts different transactions, or $\mathtt{tx}$ is not a valid transaction), a NIZKPoK proof should be attached. Our statement is of form $\mathsf{stmt} = (h, \mathtt{st}, (ct_1, \ldots, ct_m), (\mathsf{pk}_1, \ldots, \mathsf{pk}_m))$ and witness $\mathsf{w} = (\mathtt{tx}, (r_1, \ldots, r_m))$. The NP relation is defined as follows (where we slightly abuse the notation and let Enc taking an additional parameter r to replace the randomness generation by using r):

$$\forall i \in [m], \exists (r_i, \mathtt{tx}) \text{ s.t. } ct_i = \mathsf{PKE.Enc}(\mathsf{pk}_i, (\mathtt{tx} \,\|\, h), r_i) \text{ and } \mathcal{Q}_{\mathsf{validTX}}(\mathtt{tx}, \mathtt{st}) = 1. \tag{1}$$

Note that to simplify the transaction validation, we write $\mathcal{Q}_{\mathsf{validTX}}$, taking input a transaction $\mathtt{tx}$ and a blockchain state $\mathtt{st}$, as the predicate such that $\mathcal{Q}_{\mathsf{validTX}}(\mathtt{tx}, \mathtt{st}) = 1$ if and only if $\mathtt{tx}$ is a valid transaction with respect to state $\mathtt{st}$.

We elaborate on the NIZKPoK scheme that we are going to use. Let Π_{NIZK} denote a NIZKPoK protocol [LR22] consisting of six algorithms Setup, Prove, Verify, $\mathsf{SimSetup}$, $\mathsf{SimProve}$ and $\mathsf{Extract}$. Especially, Π_{NIZK} satisfies three properties, namely (i) overwhelming completeness; (ii) non-interactive multiple special honest-verifier zero-knowledge; and (iii) non-interactive special simulation-soundness. Note that Π_{NIZK} can be implemented in the $\mathcal{G}_{\mathsf{rpoRO}}$ model.

After selecting the block ticket τ, the set of public keys $\mathcal{S}_{\mathsf{pk}}^{\tau}$ and generating the ciphertexts ct, the client then generates a NIZK proof $\pi = \Pi_{\mathsf{NIZK}}.\mathsf{Prove}((h, \mathtt{st}, ct, \mathcal{S}_{\mathsf{pk}}^{\tau}), (\mathtt{tx}, r))$ where $\mathtt{st}$ is the (hash of) blockchain state with respect to h. The client now is ready to issue the new transaction by diffusing (π, ct, τ) to the network.

Transaction decryption. To open a transaction associated with block hash h, a chain mined from h, progressing for at least Λ blocks, should be provided. We note that while in our description, clients should select a block in the immutable part of the blockchain (thus h will be extended with overwhelming probability), there is actually no restrictions on the clients to prevent them from selecting a very recent block. However, this comes with the risk of hurting liveness — if a block $\mathcal{B}$ in the unsettled part is selected for issuing a transaction and $\mathcal{B}$ get orphaned, then that transaction cannot be considered legitimate on this chain.

[10] Note that our protocol requires an initial enclave public key registration stage to gather sufficiently many on-chain honest enclave public keys. This requires the clients to wait until the blockchain grows to at least $2k$ blocks (where k is the common prefix parameter) so that at least one honest block gathers honest public keys and gets buried sufficiently deep in the settled blockchain. This "warm-up" stage can be captured by enforcing the transaction process mechanism to start after the chain grows to $2k$ blocks.

[11] Previous works encrypt only transaction content but metadata (e.g., transaction issuer's public key) remains transparent.

While a transaction associated with an orphaned block won't be accepted on the main chain, we highlight that all valid transactions can be opened eventually, when there is a curious adversary that works on the orphaned fork alone and hence open the encryption using his own enclave.

5.2 Blind Transaction Serialization

In this section we present our PoW blockchain framework and show how a virtual ledger $\mathcal{L}$ can be built on top of the chain $\mathcal{C}$. In a nutshell, the blockchain $\mathcal{C}$ can be seen as derived from the Nakamoto-style PoW chain by additionally binding the mining procedure of the blocks that forms the chain $\mathcal{C}$ and a new type of block called "profile blocks", using 2×1 (pronounced "two-for-one") PoW. Note that the 2×1 PoW blockchain has been proposed in [GKL15]. We borrow their basic constructions but adapt it specifically to reach agreement over the timestamp of each transaction. The virtual ledger $\mathcal{L}$ is built by serializing all transactions based on their aggregated timestamp.

In 2×1 PoW, a hash function output u is checked twice for u and its reversed bit-string $[u]^{\mathsf{R}}$. If $u < T$, a block that extends the chain $\mathcal{C}$ is produced; and if $[u]^{\mathsf{R}} < T$, a new profile block PB is mined (which we will detail soon). In such a way, the mining procedure of the chain $\mathcal{C}$ and profile blocks are "bound" together and the adversary cannot gain advantage on either type of blocks by dropping from mining the other.

The blockchain $\mathcal{C}$ in $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ is a sequence of blocks connected by the hash reference to the previous block, and all validation and chain-selection algorithms follow the longest-chain rule. The only difference between our chain $\mathcal{C}$ in $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ and that in, e.g., Bitcoin, is that blocks in $\mathcal{C}$ does not include transactions "directly". Instead, they only include valid profile blocks and valid enclave public keys (also, valid transaction decryption signed by the enclave). As for the profile blocks, they share the same block head structure with on-chain blocks; further, for each PB, its content is a transaction list of pair (tag, t) where tag is the transaction identifier (hash of the tuple of encrypted transaction, NIZK proof and block ticket (ct, π, τ)) and t is its local receiving time.

When a maintainer P receives a transaction in the form of (ct, π, τ), it first extracts the block hash h associated with this transaction. If h is not on P's local chain, the transaction gets rejected. Further, if the proof π fails to verify, or it contains a state st does not match h, or ct is encrypted from the wrong set of enclave public keys associated with block ticket τ, P rejects this transaction either. When all these verification pass, P bookkeeps the current time r with this transaction in its buffer as $((ct, \pi, \tau), r)$

Transaction timestamping. To facilitate transaction timestamping via profile blocks, we employ two additional parameters for validating profile blocks — the recency parameter R which is used to guarantee the freshness of profile blocks and the profile window length parameter K. Specifically, a profile block PB should "attach" to a valid block $\mathcal{B}$ in the settled part of the blockchain (by including $\mathcal{B}$'s blockhash in its header), as the freshness proof that PB is not mined too early ago. Meanwhile, the blockchain $\mathcal{C}$ should reject stale profile blocks (to prevent the adversary from withholding them and releasing in the very future). Let ℓ denote the height of block $\mathcal{B}$ that PB attaches to. PB is considered as a valid profile block only when PB is included in a block with height less than $\ell + R$.

We detail the transaction settlement and timestamping scheme. Consider a transaction `tx` (at this stage miners only know its unique identifier tag) and $\mathcal{B}$ the first block that contains a profile block PB with entry $(\mathsf{tag}, \cdot)$. Let ℓ denote the block height of $\mathcal{B}$. The timestamp of `tx` (or, its position in the final ledger) is computed by calculating the median timestamps of tag from all profile blocks included in blocks with height at most $\ell + K$. Formally,

$$\mathsf{TS}(\mathsf{tag}) \triangleq \mathsf{med}\{t \mid (\mathsf{tag}, t) \in \mathsf{PB} \in B \wedge \ell \leq \mathsf{height}(B) < \ell + K\}.$$

Note that if a profile block in the K-block window does not report an entry $(\mathsf{tag}, \cdot)$, it is counted as reporting $(\mathsf{tag}, +\infty)$ (i.e., the miner had never received tag thus cannot decrypt it). In other words, in order for a transaction $\mathtt{tx}$ with identifier tag to be included in the ledger $\mathcal{L}$, the majority of these profile blocks should report an entry $(\mathsf{tag}, \cdot)$, otherwise the computed timestamp is ∞ meaning tag fails to settle into the ledger[12].

Security guarantees. Looking ahead, in our analysis we show that our blockchain $\mathcal{C}$ provides all security guarantees as the conventional ones, with additional good properties on the profile blocks. In particular, for any transaction $\mathtt{tx}$ and its associated K-block window, at least half of the profile blocks in this window are produced by honest parties. Given the overwhelming probability on winning the majority of profile blocks, the ledger built on top of $\mathcal{C}$ is a consistent and live one with two additional properties — sender fairness and input causality. In terms of sender fairness, for each transaction $\mathtt{tx}$ that enters the network at round r, eventually our ledger will finalize a timestamp t such that $r \leq t \leq r + \Delta$ for $\mathtt{tx}$. Further, regarding input causality, since no party except the transaction issuer can learn (any information about) $\mathtt{tx}$ before the chain extends for at least $\Lambda > K$ blocks, the adversary can only issue transactions after $\mathtt{tx}$ has got finalized.

5.3 Further Discussions

In this section we discuss how our protocol can be naturally translated to a Nakamoto-style Proof-of-Stake protocol, and how it can operate with dynamic participation.

Proof-of-Stake instantiation. Our protocol turns out to have a natural adaption to the PoS setting with a few minor adaptions. Note that the transaction encryption mechanism that we employ are blockchain-agnostic hence no additional modification is required. We hence focus on the blockchain part.

On one hand, we shall instantiate the 2×1 PoW blockchain using PoS. This translation is immediate: the 2×1 PoW can be replace by using two independent VRF evaluations, one for extending the PoS chain and the other for generating profile blocks. This replacement is provably equivalent and has been seen wide usage (see, e.g., [FGKR20] for more discussion). On the other hand, in PoS the simulation for chain extending is "for free", and since the enclaves are clock-oblivious they should not decrypt transactions using the "progression" on a PoS chain. To address this problem, we point to the "density"-based chain selection rule in Ouroboros Genesis [BGK+18], where the selection of two chains with long forks is based on selecting the denser chain after the fork, capturing the fact that honest parties win more slots than the adversary. A similar approach can be applied in the enclave decryption program: in order to decrypt a transaction $\mathtt{tx}$, a chain extending from the state bound with $\mathtt{tx}$ should be provided, with sufficiently many blocks in each interval (i.e., the chain should be dense), until some pre-defined time length.

Dynamic participation. Our protocol can operate with dynamic participation, with certain modifications that we provide a high-level intuition. The 2×1 PoW blockchain that we use can go with fluctuating computational power (the 2×1 mining with difficulty adjustment has been used and proved secure in, e.g., [KLS24]). Further, the enclave program can be modified to decrypt transactions if certain amount of block difficulty has been accumulated since the state that transaction bound to. The main challenge is when maintainers come and go, the set of enclave public keys that clients are supposed to use should be updated periodically. Notice that in [KLS24] the mining difficulty is adjusted by epochs, hence we could employ an additional public key registration phase for each epoch, and based on the block that a transaction is bound with, it selects the

[12]Under this majority rule, even if the adversary with minority of computational power completely drop from mining a transaction tag, honest parties can still settle tag on their own with a good timestamp.

appropriate set of keys to apply the ticket selection — and of course, this requires a more refined parametrization.

6 Security Analysis

Overview of the security analysis. In the ideal world, the simulator $\mathcal{S}$ simulates the network functionality for a black-box adversary $\mathcal{A}$; $\mathcal{S}$ also simulates the ledger maintenance procedure of honest parties and jointly builds a blockchain with $\mathcal{A}$. When dummy parties issue new transactions, $\mathcal{S}$ generates a fake ciphertext of an all-zero string and send it to the simulated network. When an encrypted transaction is received from $\mathcal{A}$, $\mathcal{S}$ extracts the plain transaction by using Π_{NIZK}. Importantly, when the simulated blockchain has progressed such that $\mathcal{S}$ learns the transaction order, $\mathcal{S}$ proposes those transactions (using txid) to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ and thus learn the honest transactions. $\mathcal{S}$ then programs the $\mathcal{G}_{\mathsf{rpoRO}}$ (and hides this "isPrograMed" information to $\mathcal{A}$) and equivocate the corresponding encrypted transactions to the real honest transaction.

We provide some more details on handling honest transactions. When the ledger $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ notifies $\mathcal{S}$ that a dummy honest party P submits a new transaction of length m with txid at time τ, $\mathcal{S}$ simulates the $\mathsf{IssueNewTransaction}$ procedure for P by associating a block $\mathcal{B}$ with this transaction, based on P's simulated local chain state. In order to learn the honest transactions, $\mathcal{S}$ 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 $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ and thus learns the transaction $\mathtt{tx}$ that is associated with txid. Note that since the transaction decryption interval parameter Λ is set slightly longer than the time that it will get settled on the blockchain, $\mathcal{S}$ always has enough time to program $\mathcal{G}_{\mathsf{rpoRO}}$ and equivocate the ciphertext of an all-zero string to $\mathtt{tx}$.

The simulator terminates when certain bad event happens. Specifically, we define the following bad events $\mathsf{BAD\text{-}CP}$ (violation of common prefix), $\mathsf{BAD\text{-}CQ}$ (violation of chain quality), $\mathsf{BAD\text{-}CG}$ (violation of chain growth), $\mathsf{BAD\text{-}PROFILE}$ (violation of honest profile majority in any K consecutive blocks), $\mathsf{BAD\text{-}NIZK}$ (failure of the NIZK scheme), $\mathsf{BAD\text{-}DEC}$ (failure of transaction equivocation), $\mathsf{BAD\text{-}TICKET}$ (failure of selecting enclave public-key subset with at least one honest key). We prove that all these bad events happen with negligible probability with respect to the security parameter. Hence, the simulator can run the simulation for any polynomial time without abortion.

Refer to Appendix D for a full description of simulator.

6.1 Blockchain Security Properties

In our protocol, a blockchain is built as an intermediate step to agree on transaction timestamp and provide long chains for decrypting transactions. We thus first focus on the good properties on the chain — namely, common prefix, chain growth and chain quality.

- **Common Prefix (CP); parameterized with $k \in \mathbb{N}^+$.** The chains $\mathcal{C}_1, \mathcal{C}_2$ possessed by two alert parties at the onset of the round $r_1 < r_2$ are such that $\mathcal{C}_1^{\lceil k} \preceq \mathcal{C}_2$, where $\mathcal{C}_1^{\lceil k}$ denotes the chain obtained by removing the last k blocks from $\mathcal{C}_1$, and $\preceq$ denotes the prefix relation.
- **Chain Growth (CG); parameterized with $\tau \in (0, 1]$ and $s \in \mathbb{N}$.** Consider a chain $\mathcal{C}$ possessed by an alert party at the onset of a round r. Let r_1 and r_2 be two previous rounds for which $r_1 + s \leq r_2 \leq r$. Then $|\mathcal{C}[r_1 : r_2]| \geq \tau \cdot s$ where τ is the speed coefficient.
- **Chain Quality (CQ); parameterized with $\mu \in (0, 1]$ and $s \in \mathbb{N}$.** Consider any portion of length at least s of the chain possessed by an alert party at the onset of a round; the ratio of blocks originating from alert parties is at least μ. We call μ the chain quality coefficient.

Note that our blockchain framework can be viewed as a "superset" of the Bitcoin backbone protocol, where they share the same chain structure (recall that all our revisions are regarding letting the chain accept new types of "profile blocks" and the transaction encryption scheme is blockchain-agnostic). Hence, following the similar arguments in [GKL15, PSs17, BMTZ17], the following Lemma 4 concluding that these properties hold throughout the entire execution except with negligible probability is immediate. Note that to simplify the presentation, we omit the detailed parametrization of the protocol but only focus on the achieved properties. Specifically, common prefix is achieved by pruning last $k = \mathsf{polylog}(\kappa)$ blocks, and the chain quality parameter $\mu > 0$, indicating that there is at least one honest block for every k consecutive blocks.

Lemma 4. *There exist protocol parametrizations such that the following properties hold throughout the an execution of $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ for $L = \mathsf{poly}(\kappa)$ rounds: (i) common prefix property with parameter $k = \Theta(\mathsf{polylog}(\kappa))$ (ii) chain growth property with parameter τ_{CG} and s_{CG}; and (iii) chain quality with parameter $\mu > 0$ and $s = k$, except with probability negligibly small in the security parameter κ.*

We next focus on the properties regarding profile blocks. We show that honest parties can always produce profile blocks that account for the majority, for any $K = \Theta(k)$ consecutive blocks on the blockchain. We provide a sketched proof using only the common prefix, chain growth and chain quality parameter; a more refined proof and parametrization can be found in, e.g., [PS17, KLS24].

Lemma 5 (Majority of honest profile blocks). *If the properties as in Lemma 4 are not violated during the execution, then in an execution of $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ over a lifetime of $L = \mathsf{poly}(\kappa)$ rounds, for any segmentation of the blockchain of $K = \Theta(k)$ consecutive blocks, the majority of the profile blocks included are produced by honest parties.*

Proof (Sketch). Consider any consecutive K blocks $\mathcal{B}_i, \ldots \mathcal{B}_{i+K-1}$. Due to chain quality, there is at least one honest block $\mathcal{B}_h$ such that $i \leq h < i + k - 1$ and $\mathcal{B}_{h'}$ such that $i + K - k < h' \leq i + K - 1$. It suffices to show that the profile blocks produced by honest parties in the time interval between $\mathcal{B}_h$ and $\mathcal{B}_{h'}$ is larger than those produced by the corrupted parties in an interval between the time of t and t' where $t = \mathsf{timestamp}(\mathcal{B}_i) - R$ and $t' = \mathsf{timestamp}(\mathcal{B}_{i+K-1}) + k$. By setting $R = 3k$ and $K = \Theta(k)$ (the constants depends on the advantage of honest computational power compared with the corrupted parties, and the chain growth parameter τ_{CG}) we conclude the proof. □

Given that for any sliding window of K blocks the majority of the profile blocks included are produced by honest parties, we consider a transaction `tx` enter the system at time t (when `tx` is issued by corrupted parties, let t denote the earliest time such that `tx` is learnt by at least one honest party). Every honest maintainer will receive `tx` at a time t' such that $t \leq t' < t + \Delta$, hence majority of the profile blocks report time in this interval — i.e., in the final ledger $\mathcal{L}$, the position of `tx` will be later than any transaction that enters the system at time before $t - \Delta$; meantime, `tx` is positioned at a place earlier than those transactions entering the system at time later than $t + \Delta$.

6.2 Composable Guarantees

Section 6.1 shows that when the simulator jointly builds the blockchain with the black-box adversary, bad events regarding the violation of good blockchain properties, namely BAD-CP, BAD-CQ, BAD-CG and BAD-PROFILE, happens with negligible probability. We next prove that the simulator $\mathcal{S}$ can simulate the transaction encryption mechanism well, by showing that all bad events BAD-TICKET, BAD-NIZK and BAD-DEC happen with negligible probability.

First, we consider BAD-TICKET which implies that the adversary $\mathcal{A}$ can send an encrypted transaction such that all public keys used in the encryption are from enclaves controlled by the corrupted parties thus $\mathcal{A}$ can decided whether to open the transaction or not.

Lemma 6 (Good block tickets). *Consider an execution of $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ over a lifetime of $L = \mathsf{poly}(\kappa)$ rounds. The probability such that there exists a block ticket (h, τ) such that h equals the hash of a block in the settled part of the blockchain held by an honest party (at any time during the execution) is negligible with respect to the security parameter κ.*

Proof. Recall that the mining target T is appropriately parameterized such that the block generation rate $f < 1$ is a small constant, by applying Chernoff bound (Theorem 10), the number of blocks generated in $L = \mathsf{poly}(\kappa)$ rounds is bounded by $(1 + \epsilon)f \cdot L$ except with probability $\exp(-\epsilon^2 fL/3) = \exp(-\Omega(\mathsf{poly}(\kappa)))$ which is negligibly small with respect to κ. I.e., consider n enclaves, at most $n \cdot \mathsf{poly}(\kappa)$ block tickets will be associated with hash of a block in the settled blockchain.

Let $p \in (0, 1]$ denote the fraction of honest enclave public keys. We first consider a variant of Program 2 where for each hash h, the enclave returns a randomly selected value (and records this value for h for later queries), which applies a random subset selection on the set of enclave public-key set $\mathcal{S}_{\mathsf{pk}}$ for each h. We show that by randomly select a subset of size $m = \mathsf{polylog}(\kappa)$ in $\mathcal{S}_{\mathsf{pk}}$ for $n \cdot \mathsf{poly}(\kappa)$ times, the probability such that event E — there exists at least one selected subset such that all keys are from the enclaves controlled by corrupted corrupted parties — is negligible. For each random subset selection, the probability that no honest key is selected is bounded by $(1 - p)^m < \exp(-\Omega(\mathsf{polylog}\kappa))$, by selecting $n \cdot \mathsf{poly}(\kappa)$ we get $\mathbf{Pr}[E] = 1 - (1 - \exp(-\Omega(\mathsf{polylog}\kappa)))^{n \cdot \mathsf{poly}(\kappa)} = \exp(-\Omega(\mathsf{polylog}(\kappa)) + \ln n)$ which is negligible in κ.

Now it suffices to show that by replacing the random number with $f_{\mathsf{PRF}}(\mathsf{key}, h)$, for each ticket there is still at least one key controlled by honest party get selected. This can be proved by a reduction to the indistinguishability experiment of the underlying pseudorandom function f_{PRF} which we omit here (note that while each enclave possesses her own key, since the total number of the enclaves are polynomially bounded this does not hurt the argument). □

Then we consider BAD-NIZK which implies that the simulator $\mathcal{S}$ cannot extract the encrypted transactions issued by the corrupted parties thus the simulation fails (note that it is not hard to see that the event such that honest parties fail to create a proof, or the corrupted parties generate a proof for a relation not in Equation (1) is negligible).

Lemma 7 (Good NIZK). *The probability that the simulator fails to extract the witness from a NIZK proof created by the black-box adversary is negligibly small in the security parameter κ.*

Proof. We prove this by a reduction to the security game of the special simulation-soundness (Game 1 in Definition 3), and show that if the simulator fails to extract, then the game returns `fail` with non-negligible probability, contradicting the fact that Π_{NIZK} is non-interactive special simulation-soundness.

The reduction in general works as follows. When the simulator $\mathcal{S}$ is to generate a NIZK proof for dummy honest parties, $\mathcal{S}$ forwards the query $(\texttt{Prove}, x, w)$ as that in Game 1; when receiving a proof (x, π) from the black-box adversary, $\mathcal{S}$ forwards $(\texttt{Challenge}, x, \pi)$ and extracts the witness. If this extraction fails and simulator aborts, then in Game 1 it must be the case that after running the Extract algorithm and get witness w it holds that $R(x, w) = 0$. I.e., if the simulator aborts for the failed extraction with non-negligible probability, then in Game 1 it also returns `Fail` with non-negligible probability. □

Recall that in Lemma 3, the black-box adversary $\mathcal{A}$ cannot distinguish if a plaintext of PKE is decrypted by Dec or Dec^*, finally we consider the event $\mathsf{BAD\text{-}DEC}$ which implies that the simulator fails to equivocate a faked ciphertext to its corresponding honest transaction.

Lemma 8 (Good decryption). *The probability such that when $\mathcal{A}$ queries $\mathcal{G}_{\mathsf{att}}$ a faked ciphertext ct of an all-zero string corresponding to transaction $\mathtt{tx}$ and $\mathcal{G}_{\mathsf{att}}$ returns a response other than $(\text{TX-DEC}, (\mathtt{tx}, ct))$ is negligible with respect to the security parameter κ.*

Proof. The event of $\mathsf{BAD\text{-}DEC}$ happens, because either $\mathcal{S}$ has no time to program the random oracle, or the programming fails.

We first consider the case that $\mathcal{S}$ has no time to program. Note that since the decryption parameter is set as $\Lambda > 4k + K$, for an honest transaction $\mathtt{tx}$ that associates with a block with height ℓ, there will exist an honest block with height at most $\ell + 2k$ (due to chain quality) such that it contains the profile blocks with $\mathtt{tx}$. I.e., when the blockchain has progressed to length $\ell + 4k + K$, the timestamp of $\mathtt{tx}$ get settled, and the same holds for all transactions that enters the system before $\mathtt{tx}$. Hence $\mathcal{S}$ can propose a block containing $\mathtt{tx}$ and learn its plaintext before the chain grows to length $\ell + \Lambda > \ell + 4k + K$; i.e., when the simulator program all ciphertext for $\mathtt{tx}$ the TX-DEC interface will never try to use Dec^* to decrypt ct of $\mathtt{tx}$.

Regarding the event that programming $\mathcal{G}_{\mathsf{rpoRO}}$ fails, note that this only happens when the point x to program has already been queried before. Note that for any point x to program, there exists a (piece of) ciphertext ct with respect to $\mathsf{pk} = (\mathsf{f}, \mathsf{b})$, such that $ct = \mathsf{f}(x)$. If the event that any programming on point x fails, then it implies $\mathbf{Pr}_{x \in \{0,1\}^\kappa}[\mathcal{A}(f(x)) \in \mathsf{f}^{-1}(\mathsf{f}(x))]$ is non-negligible, contradicting the fact that f is a secure one-way function. □

We conclude that our protocol $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$, when appropriately parameterized, securely realizes $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$ in an $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{W}(\mathcal{G}^{\mathsf{PoW}}_{\mathsf{RO}}), \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{G}_{\mathsf{att}}, \mathcal{G}_{\mathsf{rpoRO}})$-hybrid environment.

Theorem 9. *Let Δ denote the network delay, τ_{CG} the chain-growth coefficient, k the common prefix parameter, K the length of profile block window and PKE, Λ the parameter of Program 1 where PKE is as specified in Algorithm 4. Assuming honest majority in terms of computational power and honest parties controlling a constant fraction of enclave public keys, there exists protocol parametrization ($k = \Theta(\Delta \log^2 \kappa)$, $K = \Theta(k)$ and $\Lambda = K + 5k$, $h/n = \Omega(\log^2 \kappa)$) such that Protocol $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ UC-realizes $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$ parameterized with Δ-ApproxSenderOrder, with $\mathtt{windowSize} = k$, $\mathtt{Delay} = 2\Delta$ and $\mathtt{waitTime} = (4k + K)/\tau_{\mathsf{CG}}$, in the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{W}(\mathcal{G}^{\mathsf{PoW}}_{\mathsf{RO}}), \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{G}_{\mathsf{att}}, \mathcal{G}_{\mathsf{rpoRO}})$-hybrid environment.*

Proof (Sketch). We prove the theorem by providing a simulator $\mathcal{S}_{\text{ledger}}$ in the ideal world such that the protocol execution in the $(\mathcal{G}_{\mathsf{Clock}}, \mathcal{W}(\mathcal{G}^{\mathsf{PoW}}_{\mathsf{RO}}), \mathcal{F}_{\mathsf{Diffuse}}, \mathcal{G}_{\mathsf{att}}, \mathcal{G}_{\mathsf{rpoRO}})$-hybrid world is indistinguishable from the ideal-world execution with the ledger functionality and the simulator. We detail the full description of $\mathcal{S}_{\text{ledger}}$ in Appendix D. This simulation can be done perfectly, as the only events that prevent a successful simulation are those defined by $\mathsf{BAD\text{-}CP}$, $\mathsf{BAD\text{-}CQ}$, $\mathsf{BAD\text{-}CG}$, $\mathsf{BAD\text{-}PROFILE}$, $\mathsf{BAD\text{-}NIZK}$ and $\mathsf{BAD\text{-}DEC}$, which we have been proving in the previous Lemma 4, 5, 6, 7 and 8 that happens with negligible probability. □

Acknowledgements

Yu Shen's research has been supported by Input Output (iohk.io) through their funding of the University of Edinburgh Blockchain Technology Lab.

References

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

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

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

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

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

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press. https://doi.org/10.1109/SFCS.2001.959888.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

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

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

[CMSZ22] Christian Cachin, Jovana Micic, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In Ittay Eyal and Juan A. Garay, editors, *FC 2022: 26th International Conference on Financial Cryptography and Data Security*, volume 13411 of *Lecture Notes in Computer Science*, pages 316–333, Grenada, May 2–6, 2022. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-031-18283-9_15.

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

[DM16] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. In *ISOC Network and Distributed System Security Symposium – NDSS 2016*, San Diego, CA, USA, February 21–24, 2016. The Internet Society.

[FGKR20] Matthias Fitzi, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ledger combiners for fast settlement. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 322–352, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-030-64375-1_12.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310, Sofia, Bulgaria, April 26–30, 2015. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-662-46803-6_10.

[GKL17] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 291–323, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-319-63688-7_10.

[GKM+22] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. *Distributed Computing*, 35(1):1–15, Feb 2022. https://doi.org/10.1007/s00446-021-00399-2.

[KDK22] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. In Jason Paul Cruz and Naoto Yanai, editors, *APKC '22: Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS 2022, Nagasaki, Japan, 30 May 2022*, pages 3–14. ACM, 2022. https://doi.org/10.1145/3494105.3526239.

[KLS24] Aggelos Kiayias, Nikos Leonardos, and Yu Shen. Ordering transactions with bounded unfairness: Definitions, complexity and constructions. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 34–63. Springer Nature Switzerland, 2024. https://doi.org/10.1007/978-3-031-58734-4_2.

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

[KO04] Jonathan Katz and Rafail Ostrovsky. Round-optimal secure two-party computation. In Matthew Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 335–354, Santa Barbara, CA, USA, August 15–19, 2004. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-540-28628-8_21.

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

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

[LR22] Anna Lysyanskaya and Leah Namisa Rosenbloom. Universally composable Σ-protocols in the global random-oracle model. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022: 20th Theory of Cryptography Conference, Part I*, volume 13747 of *Lecture Notes in Computer*

Science, pages 203–233, Chicago, IL, USA, November 7–10, 2022. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-031-22318-1_8.

[MS23] Dahlia Malkhi and Pawel Szalachowski. Maximal Extractable Value (MEV) Protection on a DAG. In Yackolley Amoussou-Guenou, Aggelos Kiayias, and Marianne Verdier, editors, *4th International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2022)*, volume 110 of *Open Access Series in Informatics (OASIcs)*, pages 6:1–6:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.Tokenomics.2022.6.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008. URL: https://bitcoin.org/bitcoin.pdf.

[PS17] Rafael Pass and Elaine Shi. FruitChains: A fair blockchain. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *36th ACM Symposium Annual on Principles of Distributed Computing*, pages 315–324, Washington, DC, USA, July 25–27, 2017. Association for Computing Machinery. https://doi.org/10.1145/3087801.3087809.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, apr 1980. https://doi.org/10.1145/322186.322188.

[PSs17] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-319-56614-6_22.

[PST17] Rafael Pass, Elaine Shi, and Florian Tramèr. Formal abstractions for attested execution secure processors. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 260–289, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany. https://doi.org/10.1007/978-3-319-56620-7_10.

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

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

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

A Preliminaries (Cont'd)

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

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

The functionality manages the set $\mathcal{P}$ of registered identities, i.e., partyset $\mathcal{P} = (\text{pid}, \text{sid})$. It also manages the set F of functionalities (together with their session identifier). Initially, $\mathcal{P} \leftarrow \emptyset$ and $F \leftarrow \emptyset$.
For each session sid the clock maintains a variable τ_{sid}. For each identity $\mathsf{P} = (\text{pid}, \text{sid}) \in \mathcal{P}$ it manages variable d_{P}. For each pair $(\mathcal{F}, \text{sid}) \in F$ it manages variable $d(\mathcal{F}, \text{sid})$ (all integer variables are initially 0).

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

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

Diffuse functionalities. We model the Δ-bounded delay network with $\mathcal{F}_{\mathsf{Diffuse}}$ [BMTZ17, BGK+18]. Note that once a corrupted message reaches at least one honest party at round r, $\mathcal{F}_{\mathsf{Diffuse}}$ guarantees to deliver this message to all honest parties before round $r + \Delta$ (i.e., honest parties keep "echoing" messages). By convention, different types of messages are diffused by different functionalities, and we write $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{pb}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{pk}}_{\mathsf{Diffuse}}$ to denote the network for chains, transactions, profile blocks and enclave publick keys respectively.

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

The functionality is parameterized with a set possible senders and receivers $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$.

- **Honest sender diffusion**. Upon receiving $(\text{DIFFUSE}, \text{sid}, m)$ from some $\mathsf{P} \in \mathcal{P}$, where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set, choose n new unique message-IDs $\text{mid}_1, \ldots, \text{mid}_n$ of the form $\text{mid}_i = (\text{mid}_n, i)$, initialize $2n$ new variables $D_{\text{mid}_1} := D^{MAX}_{\text{mid}_1} \ldots := D_{\text{mid}_n} := D^{MAX}_{\text{mid}_n} := 1$, a per message delay $\Delta_{\text{mid}_i} = \Delta$ for $i = 1, \ldots, n$ and set $\mathbf{M} := \mathbf{M} \parallel (m, \text{mid}_1, D_{\text{mid}_1}, U_1) \parallel \ldots \parallel (m, \text{mid}_n, D_{\text{mid}_n}, U_n)$, and send $(\text{DIFFUSE}, \text{sid}, m, \mathsf{P}, (U_1, \text{mid}_1), \ldots, (U_n, \text{mid}_n))$ to the adversary.
- **Adversarial sender diffusion**. Upon receiving $(\text{DIFFUSE}, \text{sid}, m)$ from some $\mathsf{P} \in \mathcal{P}$ (where $\mathcal{P} = \{U_1, \ldots, U_n\}$ denotes the current party set), do execute it the same way as an honest-sender diffusion, with the only difference that $\Delta_{\text{mid}_i} = \infty$.
- **Honest party fetching.** Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathsf{P} \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P if P is corrupted):
  1. For all tuples $(m, \text{mid}, D_{\text{mid}}, \mathsf{P}) \in \mathbf{M}$, set $D_{\text{mid}} := D_{\text{mid}} - 1$.
  2. Let $\mathbf{M}^{\mathsf{P}}_0$ denote the subvector $\mathbf{M}$ including all tuples of the form $(m, \text{mid}, D_{\text{mid}}, \mathsf{P})$ with

$D_{\text{mid}} = 0$ (in the same order as they appear in $\mathbf{M}$). Then, delete all entries in $\mathbf{M}_0^{\mathsf{P}}$ from $\mathbf{M}$ and in case some $(m, \text{mid}, D_{\text{mid}}, \mathsf{P})$ is in $\mathbf{M}_0^{\mathsf{P}}$, where P is honest, set $\Delta_{\text{mid}'} = \Delta$ for any $(m, \text{mid}', D_{\text{mid}'}, \mathsf{P}')$ in $\mathbf{M}$ and replace this record by $(m, \text{mid}', \min\{D_{\text{mid}'}, \Delta\}, \mathsf{P}')$. Finally, send $\mathbf{M}_0^{\mathsf{P}}$ to P.

- **Adding adversarial delays**. Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots, (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\text{mid}_{i_j}}, \text{mid}_{i_j})$: if $D_{\text{mid}_{i_j}}^{MAX} + T_{\text{mid}_{i_j}} \leq \delta_{\text{mid}_{i_j}}$ and mid_{i_j} is a message-ID registered in the current $\mathbf{M}$, set $D_{\text{mid}_{i_j}} := D_{\text{mid}_{i_j}} + T_{\text{mid}_{i_j}}$ and set $D_{\text{mid}_{i_j}}^{MAX} := D_{\text{mid}_{i_j}}^{MAX} + T_{\text{mid}_{i_j}}$; otherwise, ignore this pair.
- **Adversarially reordering messages**. Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\mathbf{M}$, then swap the triples $(m, \text{mid}, D_{\text{mid}}, \cdot)$ and $(m, \text{mid}', D_{\text{mid}'}, \cdot)$ in $\mathbf{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.

Random oracle for PoW and its wrapper. Our random oracle for generating PoW follows that in [BMTZ17] except that now it is a shared functionality.

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

The functionality is parameterized by the security parameter κ. It maintains a dynamically updatable function table H where $H[x] = \perp$ denotes the fact that no pair of the form $(x, \cdot)$ is in H. Initially, $H = \emptyset$.

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

In order to limit the adversary on making a certain number of queries per round, we adopt a functionality wrapper [BMTZ17] that wraps the corresponding resource to capture such restrictions. Note that since now $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$ is shared among different sessions, the adversary can ask the environment to make queries on behalf. Hence, our wrapper $\mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}})$ also puts restrictions on queries that are made by the environment.

Functionality $\mathcal{W}(\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}})$

The wrapper functionality is parameterized by a set of parties $\mathcal{P}$, and an upper bound t which restricts the evaluations of all corrupted party per round. The functionality manages the variable $\tau \in \mathbb{N}^+$ and the current set of corrupted miners $\mathcal{P}'$. It also manages variable $t_{\mathcal{A}}$. Initially, $\tau = 1$.

General: The wrapper stops the interaction with the adversary as soon as the adversary tries to exceed its budget of t queries per nominal round.

Relaying inputs to the random oracle:

- Upon receiving $(\text{EVAL}, \text{sid}, x)$ from $\mathcal{A}$ on behalf of a corrupted party $P \in \mathcal{P}'$ or any party $(\text{pid}', \text{sid}')$ such that $\text{sid}' \neq \text{sid}$, first execute *Round Reset*. Then, set $t_{\mathcal{A}} \leftarrow t_{\mathcal{A}} + 1$ and only if $t_{\mathcal{A}} \leq t_\tau$ forward the request to $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$ and return to $\mathcal{A}$ whatever $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$ returns.
- Any other request from any participant or the adversary is simply relayed to the underlying

functionality without any further action and the output is given to the destination specified by the hybrid functionality.

Corruption Handling: Upon receiving $(\text{CORRUPT}, \text{sid}, \mathsf{P})$ from the adversary, set $\mathcal{P}' \leftarrow \mathcal{P}' \cup \mathsf{P}$.

Procedure Round-Reset: Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$ and receive $(\text{CLOCK-READ}, \text{sid}_C, \tau')$ from $\mathcal{G}_{\mathsf{Clock}}$. If $|\tau - \tau'| > 0$, then set $t_{\mathcal{A}} \leftarrow 0$ for the adversary and set $\tau \leftarrow \tau'$.

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

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

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

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

Non-interactive zero knowledge scheme Π_{NIZK}. The NIZKPoK scheme that we use in this paper are from [LR22], specifically a non-interactive, straight-line extractable (NISLE) proof system satisfying the following three properties.

Definition 1 (Overwhelming Completeness). *A NISLE proof system* $\Pi_R^{\mathtt{SLC}} = (\mathtt{Setup}^H, \mathtt{Prove}^H, \mathtt{Verify}^H, \mathtt{SimSetup}, \mathtt{SimProve}, \mathtt{Extract})$ *for relation* R *in the random-oracle model has the* overwhelming completeness *property if for any security parameter* λ*, any random oracle* H*, any* $(x, w) \in R$*, and any proof* $\pi \leftarrow \Pi_R^{\mathtt{SLC}}.\mathtt{Prove}^H(x, w)$*,* $\mathbf{Pr}[\Pi_R^{\mathtt{SLC}}.\mathtt{Verify}^H = 1] \geq 1 - \mathsf{negl}(\lambda)$.

Definition 2 (Non-Interactive Multiple SHVZK). *A NISLE proof system* $\Pi_R^{\mathtt{SLC}} = (\mathtt{Setup}^H, \mathtt{Prove}^H, \mathtt{Verify}^H, \mathtt{SimSetup}, \mathtt{SimProve}, \mathtt{Extract})$ *for relation* R *in the random-oracle model has the* non-interactive multiple special honestverifier zero-knowledge (NIM-SHVZK) *property if for*

any security parameter λ, any random oracle H, any PPT adversary $\mathcal{A}$, and a bit $b \xleftarrow{\$} \{0,1\}$, ther exist some negligible function negl *such that* $\mathbf{Pr}[b' = b] \leq 1/2 + \mathsf{negl}(\lambda)$*, where b' is the result of running the game* $\mathsf{NIM\text{-}SHVZK}^{H_*,*}_{\mathcal{A},\Pi^{\mathtt{SLC}}_R}(1^\lambda, b)$*. We say $\mathcal{A}$ wins the* NIM-SHVZK *game if* $\mathbf{Pr}[b' = b] > 1/2 + \mathsf{negl}(\lambda)$.

Definition 3 (Non-Interactive Special Simulation-Soundness). *A NISLE proof system* $\Pi^{\mathtt{SLC}}_R = (\mathtt{Setup}^H, \mathtt{Prove}^H, \mathtt{Verify}^H, \mathtt{SimSetup}, \mathtt{SimProve}, \mathtt{Extract})$ *for relation R in the random-oracle model has the* non-interactive special simulation-soundness *property if for any security parameter λ, any random oracle H, any PPT adversary $\mathcal{A}$, there exist some negligible function* negl *such that* $\mathbf{Pr}[\mathtt{Fail} \leftarrow \mathsf{NIM\text{-}SSS}^{H_*,\mathtt{Prog}}_{\mathcal{A},\Pi^{\mathtt{SLC}}_R}(1^\lambda)] \leq \mathsf{negl}(\lambda)$.

For completeness, we also describe the security game $\mathsf{NIM\text{-}SSS}^{H_*,\mathtt{Prog}}_{\mathcal{A},\Pi^{\mathtt{SLC}}_R}(1^\lambda)$ related to the non-interactive special simulation-soundness property of the NIZK protocol where $\mathcal{Q}_\mathcal{A}$ are $\mathcal{A}$'s queries to the RO (details see [LR22]).

Game 1 $\mathsf{NIM\text{-}SSS}^{H_*,\mathtt{Prog}}_{\mathcal{A},\Pi^{\mathtt{SLC}}_R}(1^\lambda)$

$L \leftarrow \perp$
$\mathtt{ppm}, z \leftarrow \Pi^{\mathtt{SLC}}_R.\mathsf{SimSetup}^{\mathtt{prog}_L}(1^\lambda)$
$\mathtt{st} \leftarrow \mathcal{A}^{H_L}(1^\lambda, \mathtt{ppm})$
$\mathtt{pflist}, \mathtt{Response} \leftarrow \perp$
while $\mathtt{st} \neq \perp$ **do**
  $(\mathtt{Query}, \mathcal{Q}_\mathcal{A}, \mathtt{st}) \leftarrow \mathcal{A}^{H_L}(\mathtt{st})$
  if $\mathtt{Query} = (\mathtt{Prove}, x, w)$ **then**
    if $R(x, w) = 1$ **then**
      $\Pi \leftarrow \Pi^{\mathtt{SLC}}_R.\mathsf{SimProve}^{\mathtt{prog}_L}(z, x)$
      Append (x, π) to $\mathtt{pflist}$
      Set $\mathtt{Response} \leftarrow (x, \pi)$
    end if
  else if $\mathtt{Query} = (\mathtt{Challenge}, x, \pi)$ **then**
    if $\Pi^{\mathtt{SLC}}_R.\mathsf{Verify}^{\mathtt{prog}_L}(x, \pi) = 1$ **and** $(x, \pi) \notin \mathtt{pflist}$ **then**
      $w \leftarrow \Pi^{\mathtt{SLC}}_R.\mathsf{Extract}(x, \pi, \mathcal{Q}_\mathcal{A})$
      if $R(x, w) = 0$ **then**
        return $\mathtt{Fail}$
      end if
    end if
  end if
end while
return $\mathtt{Success}$

B The Extended Ledger Functionality

We provide a complete description of our ledger $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$. For brevity, major revisions compared with [BMTZ17] are marked with blue text.

Functionality $\mathcal{G}_{\text{Ledger}}^{\text{Fair}}$

General: The functionality is parametrized by four algorithms Validate, ExtendPolicy, Blockify, and predict-time, along with two parameters $\texttt{windowSize}, \texttt{Delay}, \texttt{waitTime} \in \mathbb{N}$. The functionality manages variables $\texttt{state}$, $\texttt{NxtBC}$, $\texttt{buffer}$, $\tau_{\mathcal{L}}$ and $\vec{\tau}_{\texttt{state}}$. Initially, $\texttt{state} \leftarrow \varepsilon, \texttt{NxtBC} \leftarrow \varepsilon, \vec{\tau}_{\texttt{state}} \leftarrow \varepsilon, \texttt{buffer} \leftarrow \emptyset, \tau_{\mathcal{L}} \leftarrow 0$.
For each party $\mathsf{P}_i \in \mathcal{P}$ the functionality maintains a pointer $\texttt{pt}_i$ (initially set to 1) and a current-state view $\texttt{state}_i \leftarrow \varepsilon$ (initially set to empty). The functionality keeps track of the timed honest-input sequence $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T \leftarrow \varepsilon$).

Party management: The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$ (following the definition in the previous paragraph). The sets $\mathcal{P}$, $\mathcal{H}$, $\mathcal{P}_{DS}$ are all initially set to $\emptyset$. When a new honest party is registered at the ledger, if it is registered with the clock already then it is added to the party sets $\mathcal{H}$ and $\mathcal{P}$ and the current time of registration is also recorded; if the current time is $\tau_{\mathcal{L}} > 0$, it is also added to $\mathcal{P}_{DS}$. Similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ (and therefore also from $\mathcal{P}_{DS}$ or $\mathcal{H}$). The ledger maintains the invariant that it is registered (as a functionality) to the clock whenever $\mathcal{H} \neq \emptyset$. A party is considered fully registered if it is registered with the ledger and the clock.

Upon receiving any input I from any party or from the adversary, send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{Clock}}$ and upon receiving response $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ set $\tau_{\mathcal{L}} \leftarrow \tau$, and do the following

1. Let $\hat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that have been registered (continuously) since time $\tau' < \tau_{\mathcal{L}} - \texttt{Delay}$ (to both ledger and clock). Set $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \backslash \hat{\mathcal{P}}$. On the other hand, for any synchronized party $\mathsf{P} \in \mathcal{H} \backslash \mathcal{P}_{DS}$, if P is not registered to the clock, then $\mathcal{P}_{DS} \cup \{\mathsf{P}\}$.

2. If I was received from an honest party $\mathsf{P}_i \in \mathcal{P}$:
   (a) Set $\vec{\mathcal{I}}_H^T \leftarrow \vec{\mathcal{I}}_H^T \,\|\, (I, \mathsf{P}_i, \tau_{\mathcal{L}})$;
   (b) Compute $\vec{N} = (\vec{N}_1, \ldots \vec{N}_\ell) \leftarrow \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}, \vec{\tau}_{\texttt{state}})$ and if $\vec{N} \neq \varepsilon$ set $\texttt{state} \leftarrow \texttt{state} \,\|\, \mathsf{Blockify}(\vec{N}_1) \,\|\, \ldots \,\|\, \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\texttt{state}} \leftarrow \vec{\tau}_{\texttt{state}} \,\|\, \tau_{\mathcal{L}}^\ell$ where $\tau_{\mathcal{L}}^\ell = \tau_{\mathcal{L}} \,\|\, \ldots \,\|\, \tau_{\mathcal{L}}$;
   (c) For each $\texttt{BTX} \in \texttt{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 0$ then delete $\texttt{BTX}$ from $\texttt{buffer}$. Also, reset $\texttt{NxtBC} \leftarrow \varepsilon$.
   (d) If there exists $\mathsf{P}_j \in \mathcal{H} \backslash \mathcal{P}_{DS}$ such that $|\texttt{state}| - \texttt{pt}_j > \texttt{windowSize}$ or $\texttt{pt}_j < |\texttt{state}_j|$, then set $\texttt{pt}_k \leftarrow |\texttt{state}|$ for all $\mathsf{P}_k \in \mathcal{H} \backslash \mathcal{P}_{DS}$.

3. Depending on the input I and the ID of the sender, execute the respective code:

   – *Submitting a transaction:*
   If $I = (\text{SUBMIT}, \text{sid}, \texttt{tx})$ and is received from a party $\mathsf{P} \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party P) do the following:
   (a) Choose a unique transaction ID txid and set $\texttt{BTX} \leftarrow (\texttt{tx}, \text{txid}, \tau_{\mathcal{L}}, \mathsf{P})$;
   (b) If $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = \mathsf{true}$, then $\texttt{buffer} \leftarrow \texttt{buffer} \cup \{\texttt{BTX}\}$;
   (c) Send $(\text{SUBMIT}, (|\texttt{BTX.tx}|, \texttt{BTX.txid}, \texttt{BTX.}\tau_{\mathcal{L}}, \texttt{BTX.P}))$ to $\mathcal{A}$.

   – *Reading the state:*

If $I = (\text{READ}, \text{sid})$ is received from a fully registered party $\mathsf{P}_i \in \mathcal{P}$ then set $\mathtt{state}_i \leftarrow \mathtt{state}|_{\min\{\mathtt{pt}_i, |\mathtt{state}|\}}$ and return $(\text{READ}, \text{sid}, \mathtt{state}_i)$ to the requestor. If the requestor is $\mathcal{A}$ then send $(\mathtt{state}, \widehat{\mathtt{buffer}}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$ where $\widehat{\mathtt{buffer}} \triangleq \{(|\mathtt{tx}|, \cdot, \cdot, \cdot) \,|\, \mathtt{BTX} = (\mathtt{tx}, \cdot, \cdot, \cdot) \in \mathtt{buffer}\}$.

- *Maintaining the ledger state:*
  If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ is received by an honest party $\mathsf{P}_i \in \mathcal{P}$ and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \hat{\tau} > \tau_{\mathcal{L}}$ then send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$. Else send I to $\mathcal{A}$.
- *The adversary proposing the next block:*
  If $I = (\text{NEXT-BLOCK}, \text{hFlag}, (\text{txid}_1, \ldots, \text{txid}_\ell))$ is sent from the adversary, update $\mathtt{NxtBC}$ as follows:
  (a) Set listOfTxid $\leftarrow \varepsilon$;
  (b) For $i = 1, \ldots, \ell$ do: if there exists $\mathtt{BTX} = (\cdot, \text{txid}, \tau_{\mathcal{L}}, \mathsf{P}_i) \in \mathtt{buffer}$ with ID txid $=$ txid_i then set listOfTxid $\leftarrow$ listOfTxid $\|$ txid_i;
  (c) Finally, set $\mathtt{NxtBC} \leftarrow \mathtt{NxtBC} \,\|\, (\text{hFlag}, \text{listOfTxid})$ and output $(\text{NEXT-BLOCK}, \mathsf{ok})$ to $\mathcal{A}$.
- *The adversary setting state-slackness:*
  If $I = (\text{SET-SLACK}, (P_{i_1}, \hat{\mathtt{pt}}_{i_1}), \ldots, (P_{i_\ell}, \hat{\mathtt{pt}}_{i_\ell}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{H} \backslash \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:
  (a) If for all $j \in [\ell]$: $|\mathtt{state}| - \hat{\mathtt{pt}}_{i_j} \leq 4\mathtt{windowSize}$ and $\hat{\mathtt{pt}}_{i_j} \geq |\mathtt{state}_{i_j}|$, set $\mathtt{pt}_{i_j} \leftarrow \hat{\mathtt{pt}}_{i_j}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, \mathsf{ok})$ to $\mathcal{A}$;
  (b) Otherwise set $\mathtt{pt}_{i_j} \leftarrow |\mathtt{state}|$ for all $j \in [\ell]$.
- *The adversary setting the state for desychronized parties:*
  If $I = (\text{DESYNC-STATE}, (\mathsf{P}_{i_1}, \mathtt{state}'_{i_1}), \ldots, (\mathsf{P}_{i_\ell}, \mathtt{state}'_{i_\ell}))$, with $\{\mathsf{P}_{i_1}, \ldots, \mathsf{P}_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\mathtt{state}_{i_j} \leftarrow \mathtt{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, \mathsf{ok})$ to $\mathcal{A}$.

C A Full Protocol Description

We introduce the main $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ protocol instance that dispatches to the relevant subprocesses. This protocol is parameterized with the k the common prefix, R the recency parameter, m the public-key subset size and K the profile window length. Refer to Table 1 in Appendix F for a detailed explanation.

Protocol $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$

Global Variables:
- Parameters: k, R, m, K
- Local states: $\mathtt{r}, \mathcal{C}_{\text{loc}}, \mathtt{buffer}_{\mathsf{tx}}, \mathtt{buffer}_{\mathsf{PB}}, \mathtt{buffer}_{\mathsf{pk}}$

Registration/Deregistration:
- Upon receiving $(\text{REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}, \mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{att}}\}$, execute $\mathsf{Registration}(\mathsf{P}, \text{sid}, \mathtt{Reg}, \mathcal{R})$.
- Upon receiving $(\text{DE-REGISTER}, \mathcal{R})$, where $\mathcal{R} \in \{\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}, \mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{att}}\}$, execute

$\mathsf{Deregistration}(\mathsf{P}, \text{sid}, \mathtt{Reg}, \mathcal{R})$.

- Upon receiving input $(\text{IS-REGISTERED}, \text{sid})$ return $(\text{REGISTER}, \text{sid}, 1)$ if the local registry $\mathtt{Reg}$ indicates that this party has successfully completed a registration with $\mathcal{R} = \mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ (and did not de-register since then). Otherwise, return $(\text{REGISTER}, \text{sid}, 0)$.

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

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

Handling external calls:

- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ forward it to $\mathcal{G}_{\mathsf{Clock}}$ and output $\mathcal{G}_{\mathsf{Clock}}$'s response.
- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$, record that a clock-update was received in the current round. If this protocol instance is currently only registered to the clock (and no other functionality), then forward $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\mathsf{Clock}}$.

Registration and de-registeration. In order to participate in the protocol, parties need to register with their resources. Algorithm 5 captures the registration procedure. Note that before registering with the network and random oracle, parties should check if they have access to all global functionalities.

Algorithm 5 $\mathsf{Registration}(\mathsf{P}, \text{sid}, \mathtt{Reg}, \mathcal{G})$

if $\mathcal{G} \in \{\mathcal{G}_{\mathsf{Clock}}, \mathcal{G}_{\mathsf{att}}\}$ **then**
    Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{G}$, set registration status to registered with $\mathcal{G}$, and output the valued received by $\mathcal{G}$.
else if $\mathcal{G} = \mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ **then**
    if the party is not registered with $\mathcal{G}_{\mathsf{Clock}}$ or $\mathcal{G}_{\mathsf{att}}$ or is already registered with all setup functionalities **then**
        ignore this input
    else
        Send $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\Delta}$ and $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$.
        Output $(\text{REGISTER}, \text{sid}, \mathsf{P})$ once completing the registration with all the above resources $\mathcal{F}$.
    end if
end if

The de-registration procedure (Algorithm 6) is analogous to the above, where de-registering from the ledger is simplified as dropping from the network functionality.

Algorithm 6 $\mathsf{Deregistration}(\mathsf{P}, \mathrm{sid}, \mathtt{Reg}, \mathcal{G})$

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

The transactions issuing procedure. In order to issue a transaction $\mathtt{tx}$, party P needs to encrypt $\mathtt{tx}$ using a subset of on-chain public keys, computed from a ticket associated with a block in the settled blockchain. Additionally, a NIZKPoK that proves the validity of $\mathtt{tx}$ and the correctness of each ciphertext should be attached (See the relation in Equation (1)).

Algorithm 7 $\mathsf{IssueNewTransaction}(\mathsf{P}, \mathrm{sid}, \mathtt{tx})$

1: $h \leftarrow \mathrm{head}(\mathcal{C}^{\lceil k}_{\mathrm{loc}})$ and $\mathtt{st} \leftarrow$ blockchain state associated with h
2: $\mathcal{S}_{\mathsf{pk}} \leftarrow$ all enclave public keys up to block with hash h
3: $\mathcal{S}^{\tau}_{\mathsf{pk}} \leftarrow f_{\mathsf{select}}(\mathcal{S}_{\mathsf{pk}}, m, \tau)$ and $m' \leftarrow |\mathcal{S}^{\tau}_{\mathsf{pk}}|$
4: **for** i **from** 1 **to** m' **do**
5: $\mathsf{pk}_i \leftarrow \mathcal{S}^{\tau}_{\mathsf{pk}}[i], r_i \xleftarrow{\$} \{0,1\}^{\kappa}$
6: $ct_i \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_i, \mathtt{tx}, h; r_i)$
7: **end for**
▷ Generate NIZK proof.
8: $\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{Prove}((h, \mathtt{st}, (ct_1, \ldots, ct_{m'}), (\mathsf{pk}_1, \ldots, \mathsf{pk}_{m'})), (\mathtt{tx}, (r_1, \ldots, r_{m'})))$
9: $\mathtt{buffer}_{\mathsf{tx}} \leftarrow \mathtt{buffer}_{\mathsf{tx}} \,\|\, (\pi, ct_1, \ldots, ct_{m'})$
10: Send $(\mathsf{diffuse}, \mathrm{sid}, (\pi, ct_1, \ldots, ct_{m'}, \tau))$ to $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$

Fetch information. Parties fetch information from different diffusion functionalities — $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{pb}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$, $\mathcal{F}^{\mathsf{pk}}_{\mathsf{Diffuse}}$— to learn the new chains, profile blocks, transactions and enclave public keys, respectively. When receiving an encrypted transaction, they verify if it provides a valid block ticket and whether it attaches a good NIZK proof. If the transaction verifies, bookkeep its local receiving time.

Algorithm 8 $\mathsf{FetchInformation}(\mathsf{P}, \mathrm{sid})$

▷ Fetch blocks.
1: Send $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$; denote the response from $\mathcal{F}^{\mathsf{bc}}_{\mathsf{Diffuse}}$ by $(\textsc{fetch}, \mathrm{sid}, b)$.
2: Extract chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from b.
3: $\mathcal{C}_{\mathrm{loc}} \leftarrow \mathsf{maxvalid}(\mathcal{C}_{\mathrm{loc}}, \mathcal{C}_1, \ldots, \mathcal{C}_k)$
▷ Fetch profile blocks.
4: Send $(\textsc{fetch}, \mathrm{sid})$ to $\mathcal{F}^{\mathsf{pb}}_{\mathsf{Diffuse}}$; denote the response from $\mathcal{F}^{\mathsf{pb}}_{\mathsf{Diffuse}}$ by $(\textsc{fetch}, \mathrm{sid}, b)$.
5: Extract profile blocks $\mathsf{PB}_1, \ldots, \mathsf{PB}_k$ from b.

Set $\mathtt{buffer}_{\mathsf{PB}} \leftarrow \mathtt{buffer}_{\mathsf{PB}} \| (\mathsf{PB}_1, \ldots, \mathsf{PB}_k)$
▷ Fetch enclave public keys and tickets.
Send $(\text{FETCH}, \text{sid})$ to $\mathcal{F}^{\mathsf{pk}}_{\mathsf{Diffuse}}$; denote the response from $\mathcal{F}^{\mathsf{pk}}_{\mathsf{Diffuse}}$ by $(\text{FETCH}, \text{sid}, b)$.
Extract all public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_k$ from b.
Set $\mathtt{buffer}_{\mathsf{pk}} \leftarrow \mathtt{buffer}_{\mathsf{PB}} \| (\mathsf{pk}_1, \ldots, \mathsf{pk}_k)$
▷ Fetch encrypted transactions.
Send $(\text{FETCH}, \text{sid})$ to $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$; denote the response from $\mathcal{F}^{\mathsf{tx}}_{\mathsf{Diffuse}}$ by $(\text{FETCH}, \text{sid}, b)$.
Extract transactions $(\pi_1, ct_1, \tau_1), \ldots, (\pi_k, ct_k, \tau_k)$ from b.
for i **from** 1 **to** k **do**
    if τ_i is valid **and** $\Pi_{\mathsf{NIZK}}.\mathsf{Verify}(\pi) = 1$ **and** $h, st \in \pi$ match that on $\mathcal{C}_{\text{loc}}$ **then**
        $\mathtt{buffer}_{\mathsf{tx}} \leftarrow \mathtt{buffer}_{\mathsf{tx}} \| ((\pi_i, ct_i, \tau_i), \mathtt{r})$ ◁ Bookkeep local receiving time $\mathtt{r}$
    end if
end for

Chain validation rules. The validation procedure (Algorithm 9) generally follows that in Bitcoin backbone protocol, plus additionally verifying the validity and freshness of profile blocks.

For simplicity, we use $\mathsf{validBlock}^T$ to verify if a block is a successful PoW.

$$\mathsf{validBlock}^T(ctr, r, h, st, h', st') \stackrel{\text{def}}{=} H(ctr, r, h, st, h', val) < T \wedge ctr < 2^{32}$$

Similarly, we use use $\mathsf{validProfile}^T$ to verify if a profile block is a successful PoW by checking the reverse of the string.

Algorithm 9 $\mathsf{IsValidChain}(\mathcal{C})$

if $\mathcal{C}$ starts with a block other than $\mathbf{G}$ **then return** false
if $\mathcal{C}$ encodes an invalid state with $\mathsf{isvalidstate}(\vec{\mathtt{st}}) = 0$ **then return** false
if $\mathcal{C}$ contains profile blocks with invalid content **then return** false
$r' \leftarrow \mathtt{r}, \mathcal{B} \leftarrow \text{head}(\mathcal{C}), h^* \leftarrow H(\mathcal{B})$
while $\mathcal{C} \neq \varepsilon$ **do**
    $\mathsf{isValid} \leftarrow \mathsf{true}$
    ▷ Check validity of block header
    if $(\mathsf{validBlock}^T(\mathcal{B}) = \mathsf{false}) \vee (h^* \neq H(\mathcal{B})) \vee (\mathsf{TS}(\mathcal{B}) \geq r')$ **then**
        $\mathsf{isValid} \leftarrow \mathsf{false}$
    end if
    ▷ Check validity of profile blocks in $\mathcal{B}$
    for $\mathsf{PB} \in \mathcal{B}$ **do**
        Parse PB as $\langle \cdot, r, \cdot, \cdot, h', \cdot \rangle$
        $\mathcal{B}' \leftarrow$ the block in $\mathcal{C}$ s.t. $H(\mathcal{B}) = h'$
        if $(\mathsf{validProfile}^T(\mathsf{PB}) = \mathsf{false}) \vee r \geq \mathsf{TS}(\mathcal{B})$ **then** ◁ Check validity of profile header
            $\mathsf{isValid} \leftarrow \mathsf{false}$
        end if
        if $(\mathcal{B}' = \varepsilon) \vee r \geq \mathsf{TS}(\mathcal{B}') + R$ **then** ◁ Check freshness and recency
            $\mathsf{isValid} \leftarrow \mathsf{false}$
        end if
    end for

20: **if** isValid = true **then**
21: $r' \leftarrow r, h^* \leftarrow h$
22: Remove the rightmost block in $\mathcal{C}$
23: $\mathcal{B} \leftarrow \text{head}(\mathcal{C})$
24: **else**
25: **return** false
26: **end if**
27: **end while**
28: **return** true

Longest chain selection. Parties use the same longest-chain rule (Algorithm 10) as the Bitcoin backbone protocol to select their working chain.

Algorithm 10 $\mathsf{maxvalid}(\mathcal{C}_1, \ldots, \mathcal{C}_k)$

1: $\mathcal{C}_{\mathsf{max}} \leftarrow \varepsilon$
2: **for** i **from** 1 **to** k **do**
3: **if** $\mathsf{IsValidChain}(\mathcal{C}_i)$ **and** $\text{len}(\mathcal{C}_i) > \text{len}(\mathcal{C}_{\mathsf{max}})$ **then**
4: $\mathcal{C}_{\mathsf{max}} \leftarrow \mathcal{C}_i$
5: **end if**
6: **end for**
7: **return** $\mathcal{C}_{\mathsf{max}}$

The mining procedure. Parties use 2×1 PoW to extend the blockchain and mine new profile blocks. If they succeed on either procedure, they diffuse the extended chain and new profile blocks to the corresponding diffusion network.

Algorithm 11 $\mathsf{MiningProceudre}(\mathsf{P}, \text{sid})$

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

1: Set $st \leftarrow$ Merkle root of profile blocks in $\mathtt{buffer}_{\mathsf{PB}}$, decrypted transactions in $\mathtt{buffer}_{\mathsf{tx}}$ and public keys in $\mathtt{buffer}_{\mathsf{pk}}$ that are not mined in $\mathcal{C}_{\text{loc}}$
2: Set $st' \leftarrow$ Merkle root of $((\pi_1, t_1, \tau_1), \ldots, (\pi_k, t_k, \tau_k))$ where π_i is a transaction that is not settled in $\mathcal{C}_{\text{loc}}$ and t_i its local receiving time
3: $h \leftarrow H(\text{head}(\mathcal{C}_{\text{loc}})), h' \leftarrow H(\text{head}(\mathcal{C}_{\text{loc}}^{\lceil k}))$
4: $u \leftarrow \mathsf{H}(ctr, \mathtt{r}, h, h', st, st')$
5: **if** $u < T$ **then**
6: $\mathcal{B} \leftarrow \langle ctr, h, h', \mathtt{r}, st, st' \rangle$ and $\mathcal{C}_{\text{loc}} \leftarrow \mathcal{C}_{\text{loc}} \,\|\, \mathcal{B}$
7: Send $(\text{DIFFUSE}, \text{sid}, \mathcal{C}_{\text{loc}})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ and proceed from here upon next activation of this procedure
8: **end if**
9: **if** $[u]^{\mathsf{R}} < T$ **then**
10: $\mathsf{PB} \leftarrow \langle ctr, h, h', \mathtt{r}, st, st' \rangle$
11: Send $(\text{DIFFUSE}, \text{sid}, \mathsf{PB})$ to $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pb}}$ and proceed from here upon next activation of this

procedure
12: **end if**
13: $ctr \leftarrow ctr + 1$

Ledger Maintenance. We group all the steps in the main ledger operation in LedgerMaintenance.

Algorithm 12 LedgerMaintenance(P, sid)

▷ The following steps are executed in an (MAINTAIN-LEDGER, sid, minerID)-interruptible manner:
1: Invoke FetchInformation(sid, P) to receive the newest messages for this round
▷ Decrypt transactions
2: **for** $(\pi, ct, \tau) \in \mathcal{C}_{\text{loc}}$ **do**
3: **if** $\text{len}(\mathcal{C}_{\text{loc}}) > \mathcal{B} + \Lambda$ where $\mathcal{B}$ is a block with hash h in π **and** $\mathsf{pk}_i \in \pi$ is miner's enclave public key **then**
4: Send (TX-DEC, $ct_i, \mathcal{C}_{\text{loc}}$) to $\mathcal{G}_{\mathsf{att}}$ and get response $(\mathtt{tx}, ct, \sigma)$
5: Add $(\mathtt{tx}, ct, \sigma)$ to $\mathtt{buffer}_{\mathsf{tx}}$
6: **end if**
7: **end for**
8: **if** $t_{\mathsf{work}} < \tau$ **then**
9: Call MiningProcedure(P, sid)
10: Set $t_{\mathsf{work}} \leftarrow \tau$
11: **end if**
12: **while** A (CLOCK-UPDATE, sid_C) has not been received during the current round **do**
13: Give up activation (set the anchor here)
14: **end while**
15: Send (CLOCK-UPDATE, sid_C) to $\mathcal{G}_{\mathsf{Clock}}$. ◁ Party will lose its activation here

Reading the state. Upon receiving a READ command, parties extract a transaction list from their local chain $\mathcal{C}_{\text{loc}}$. Recall that transaction inclusion and consensus are decoupled, we first introduce an algorithm ExtractTransactionSequence such that, taking input a chain $\mathcal{C}$, converts it to a sequence of blocks of transactions $\vec{N}$, which are extracted using the median timestamp. Note that $\vec{N}$ shares the same length as $\mathcal{C}$.

Algorithm 13 ExtractTransactionSequence($\mathcal{C}$)

1: Initialize $\mathsf{txList} \leftarrow \emptyset, \vec{N} \leftarrow \varepsilon$
▷ Extract transactin timestamps
2: **for** i **from** 1 **to** $\text{len}(\mathcal{C}) - K$ **do**
3: **for** $\mathsf{tag} = (ct, \pi, \tau) \in \mathsf{PB} \in \mathcal{B}_i$ **do**
4: Set $B = \{\mathcal{B}_j \mid i \leq j < i + K\}$
5: $t \leftarrow \mathsf{med}(\{t \mid (\mathsf{tag}, t) \in \mathsf{PB} \in B\} \cup \{+\infty \mid \mathsf{tag} \notin \mathsf{PB} \in \mathcal{B}\})$
6: **if** $t \neq +\infty$ **then** Add (tag, t) to txList
7: **end for**
8: **end for**

▷ Construct transaction sequence
for i **from** 1 **to** $\text{len}(\mathcal{C})$ **do**
  Set $\vec{N}_i \leftarrow \varepsilon$
  if $i \geq K + k$ **then**
    Let $\vec{\mathsf{tag}}$ denote the set of transaction tag s.t. the K-window of tag ends in $\mathcal{B}_{i-k}$
    Order $\vec{\mathsf{tag}}$ non-decreasingly based on timestamp
    for $\mathsf{tag} \in \vec{\mathsf{tag}}$ **do**
      if there is tag' in block $\mathcal{B}_{i-k+1}, \ldots \mathcal{B}_i$ s.t. $\mathsf{TS}(\mathsf{tag}') < \mathsf{TS}(\mathsf{tag})$ **then**
        $\vec{N}_i \leftarrow \vec{N}_i \,\|\, \tilde{\mathsf{tag}}_1 \,\|\, \ldots \tilde{\mathsf{tag}}_k \,\|\, \mathsf{tag}$ where $\tilde{\mathsf{tag}}_1 \,\|\, \ldots \tilde{\mathsf{tag}}_k$ are tag that are in txList with $\mathsf{TS}(\tilde{\mathsf{tag}}_i) < \mathsf{TS}(\mathsf{tag})$ however not in $\vec{N}$
      end if
    end for
  end if
  $\vec{N} \leftarrow \vec{N} \,\|\, \vec{N}_i$
end for
return $\vec{N}$

When the decryption of a transaction tag is available, that tag is replaced with the plain transaction. A state $\vec{\mathtt{st}}$ is acquired by extracting a prefix of the transaction sequence $\vec{N}$ such that all transactions are decrypted.

Algorithm 14 $\mathsf{ReadState}(\mathsf{P}, \text{sid})$

Invoke $\mathsf{FetchInformation}(\text{sid}, \mathsf{P})$ to receive the newest messages for this round
$\vec{N} \leftarrow \mathsf{ExtractTransactionSequence}(\mathcal{C}_{\text{loc}})$
for $\mathsf{tag} = (\pi, ct, \tau) \in \vec{N}$ **do**
  if there is a $\mathtt{tx} \in \mathcal{C}_{\text{loc}}$ such that $\mathtt{tx}$ is a signed decryption for ct **then**
    Replace tag with $\mathtt{tx}$
  end if
end for
Extract the state $\vec{\mathtt{st}}$ from $\mathsf{Blockify}(\vec{N}')$, where $\vec{N}'$ is a prefix of $\vec{N}$ up to a block such that all transactions are decrypted
Output $(\text{READ}, \text{sid}, \vec{\mathtt{st}}^{\lceil k})$ to $\mathcal{Z}$.

D The Simulator

We present the simulator used in the UC proof of $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$ that securely implements the ledger functionality $\mathcal{G}^{\mathsf{Fair}}_{\mathsf{Ledger}}$. The main structure follows those discussed in [BMTZ17, BGK+18] but we adapt it with fair-order extend policy and trusted hardwares.

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

Overview:
– The simulator internally emulates all local UC functionalities by running the code (and

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

Party sets: An honest miner P registered to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ is assumed to be registered in all simulated functionalities. Upon any activation, the simulator will query the current party set from the ledger (and simulate the corresponding message they send out to the network in the first maintain-ledger activation after registration), query all activations from honest parties $\vec{\mathcal{I}}_H^T$, and read the current clock value to learn the time. In particular, the simulator knows which parties are honest and synchronized and which parties are de-synchronized.

Messages from the Clock: Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C, U_p)$ from $\mathcal{G}_{\mathsf{Clock}}$, if U_p is an honest registered party, then remember that this party has received such a clock update (and the environment gets an activation). Otherwise, send $(\text{CLOCK-UPDATE}, \text{sid}_C, U_p)$ to $\mathcal{A}$.

Messages from the ledger:
- Upon receiving $(\text{SUBMIT}, m, \text{txid}, \tau, \mathsf{P})$ from $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$where m denotes the transaction length, execute $\text{IssueNewTransaction}(\mathsf{P}, \tau, m)$ and denote response by (π, ct, τ). Forward $(\text{DIFFUSE}, \text{sid}, (\pi, ct, \tau))$ to the simulated network $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ in the name of P. Output the answer of $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ to the adversary.
- Upon receiving $(\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ from $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$, extract from $\vec{\mathcal{I}}_H^T$ the party P that issued this query. If P has already completed its round-task, then ignore this request. Otherwise, execute $\text{SimulateMining}(\mathsf{P}, \tau_{\mathcal{L}})$.

Messages from the enclave: $\mathcal{S}$ acts as a dummy adversary and simply forward all responses from $\mathcal{G}_{\mathsf{att}}$ to its requestor.

Messages from the GRO $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$: $\mathcal{S}$ acts as a dummy adversary and forward all responses from $\mathcal{G}_{\mathsf{RO}}^{\mathsf{PoW}}$ to its requestor (the wrapper will stop talk to the adversarial request if all PoW budget has been consumed).

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

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

Simulation of the Network $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{bc}}$ (over which chains are sent) towards $\mathcal{A}$:
- Upon receiving $(\text{MULTICAST}, \text{sid}, (\mathcal{C}_{i_1}, U_{i_1}), \ldots, (\mathcal{C}_{i_\ell}, U_{i_\ell}))$ with a list of chains and corre-

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

Simulation of the Network $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ (over which transactions are sent) towards $\mathcal{A}$:
- Upon receiving (multicast, sid, m) from $\mathcal{A}$ with list a transaction $m = (\pi, ct, \tau)$ from $\mathcal{A}$ on behalf some corrupted $\mathsf{P} \in \mathcal{P}$, then do the following:
  (a) If τ is a valid ticket with respect to an existing block that only contains corrupted enclave public keys, **Abort** simulation: violation of good ticket (event BAD-TICKET)
  (b) Extract `tx` from π using Π_{NIZK}.SimSetup, Π_{NIZK}.SimProve and Π_{NIZK}.Extract. If extraction fails, textbfAbort simulation: violation of good NIZK (event BAD-NIZK)
  (c) Submit `tx` to the ledger on behalf of this corrupted party, and receive for the transaction id txid.
  (d) Forward the request to the internally simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{tx}}$, which replies for each message with a message-ID mid.
  (e) Remember the association between mid and the corresponding txid.
  (f) Provide $\mathcal{A}$ with whatever the network outputs.
- Upon receiving (fetch, sid) from $\mathcal{A}$ on behalf some corrupted $\mathsf{P} \in \mathcal{P}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (delays, sid, $(T_{\mathrm{mid}_{i_1}}, \mathrm{mid}_{i_1}), \ldots (T_{\mathrm{mid}_{i_\ell}}, \mathrm{mid}_{i_\ell})$) from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (swap, sid, mid, mid′) from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{tx}}$ and return whatever is returned to $\mathcal{A}$.

Simulation of the Network $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pb}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pk}}$ resp.) (over which input blocks (enclave public keys resp.) are sent) towards $\mathcal{A}$:
- Upon receiving (multicast, sid, m) from $\mathcal{A}$ with an input block m on behalf some corrupted $\mathsf{P} \in \mathcal{P}$, then do the following:
  (a) Forward the request to the internally simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pb}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pk}}$ resp.), which replies for each message with a message-ID mid.
  (b) Remember the association between mid and the corresponding input block.
  (c) Provide $\mathcal{A}$ with whatever the network outputs.

- Upon receiving $(\text{FETCH}, \text{sid})$ from $\mathcal{A}$ on behalf some corrupted $\mathsf{P} \in \mathcal{P}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pb}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pk}}$ resp.) and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\text{mid}_{i_1}}, \text{mid}_{i_1}), \ldots (T_{\text{mid}_{i_\ell}}, \text{mid}_{i_\ell}))$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pb}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pk}}$ resp.) and return whatever is returned to $\mathcal{A}$.
- Upon receiving $(\text{SWAP}, \text{sid}, \text{mid}, \text{mid}')$ from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pb}}$ ($\mathcal{F}_{\mathsf{Diffuse}}^{\mathsf{pk}}$ resp.) and return whatever is returned to $\mathcal{A}$.

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

procedure SIMULATEMINING(P, τ)
▷ Simulate the mining procedure of P in the protocol in round τ
  Execute $\mathsf{FetchInformation}(\mathsf{P}, \text{sid})$
  if $\mathsf{Update}_{\mathsf{P},\tau}$ **then**
    Send $(\text{CLOCK-UPDATE}, \text{sid}_C, \mathsf{P})$ to $\mathcal{A}$ if $\mathcal{S}_{\text{ledger}}$ has received such an input in round τ
  else
    Execute $\mathsf{MiningProcedure}(\mathsf{P}, \text{sid})$ and set $\mathsf{Update}_{\mathsf{P},\tau} \leftarrow \mathsf{true}$
    Before the activation goes to $\mathcal{A}$, execute EXTENDLEDGERSTATE(τ).
  end if
end procedure

procedure EXTENDLEDGERSTATE(τ)
  Let $\vec{N}$ be the longest state (extracted from $\mathcal{C}_{\text{loc}}$ by $\mathsf{ExtractTransactionSequence}$) among all such states $\vec{N}_{U_p}$ where $U_p \in \mathcal{H}$
  Let $\vec{N}_{\mathcal{L}}$ denote the sequence reconstructed from `state` in $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$
  if $|\vec{N}_{\mathcal{L}}| > |\vec{N}^{\lceil k}|$ **then** Execute ADJUSTVIEW(`state`)
  if $\vec{N}_{\mathcal{L}}$ is not a prefix of $\vec{N}^{\lceil k}$ **then**
    Abort simulation: consistency violation. ◁ Event BAD-CP
  end if
  Define the difference diff to be the block sequence s.t. $\vec{N}_{\mathcal{L}} \parallel \mathsf{diff} = \vec{N}^{\lceil k}$.
  Parse $\mathsf{diff} = \mathsf{diff}_1 \parallel \ldots \parallel \mathsf{diff}_n$.
  for j **from** 1 **to** n **do**
    Map each transaction `tx` in this block to its unique transaction ID txid.
    Let $\mathsf{list}_j = (\text{txid}_{j,1}, \ldots, \text{txid}_{j,\ell_j})$ be the corresponding list for this block diff_j
    if coinbase $\text{txid}_{j,1}$ specifies a party honest at block creation time **then**
      hFlag $\leftarrow 1$
    else
      hFlag $\leftarrow 0$
    end if
    Output $(\text{NEXT-BLOCK}, \mathsf{list}_j)$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ and receive $(\text{NEXT-BLOCK}, \mathsf{ok})$ as an immediate answer.
  end for
  if Fraction of blocks with hFlag $= 0$ in the recent k blocks $> 1 - \mu_{\mathsf{CQ}}$ **then**
    Abort simulation: chain quality violation ◁ Event BAD-CQ

else if State increases less than k blocks during the last k/τ_{CG} rounds **then**
  Abort simulation: chain growth violation ◁ Event BAD-CG
else if Honest profiles account for less than half in the last K blocks **then**
  Abort simulation: profile block violation ◁ Event BAD-PROFILE
end if
▷ Equivocate honest transactions.
Send $(\textsc{read}, \text{sid})$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$ and receive $\mathtt{state}$
for $\mathtt{tx} \in \mathtt{state}$ **and** txid corresponds to a faked transaction **and** $\mathsf{equivocate}_{\mathtt{tx}} = \mathsf{false}$ **do**
  Let $r_1, \ldots r_m$ denote the randomenss used to generate $ct_1, \ldots, ct_m$ of $\mathtt{tx}$ (via txid)
  for i **from** 1 to m **do**
    Split r_i into $|\mathtt{tx}|$ pieces and for each piece $r_{i,j}$ send $(\textsc{program}, r_{i,j}, h)$ such that $h \oplus ct_i = \mathtt{tx}[j]$ to $\mathcal{G}_{\mathsf{rpoRO}}$
    if programming $\mathcal{G}_{\mathsf{rpoRO}}$ fails **then**
      Abort simulation: failure of decryption ◁ Event BAD-DEC
    end if
  end for
  Set $\mathsf{equivocate}_{\mathtt{tx}} \leftarrow \mathsf{true}$
end for
Execute AdjustView($\mathtt{state}$)
end procedure

▷ Adjust the view of synchronized parties.
procedure AdjustView($\mathtt{state}, \tau$)
  $\mathsf{pointers} \leftarrow \varepsilon$
  for party $U_p \in \mathcal{H}$ of round τ **do**
    Execute $\mathsf{ReadState}(U_p, \text{sid})$ and let the chain's decoded state be $\vec{\mathtt{st}}_{U_p}$
  end for
  for each synchronized party $U_p \in (\mathcal{H} \backslash \mathcal{P}_{DS})$ of round τ **do**
    Determine the pointers $\mathtt{pt}_{U_p}$ s.t. $\vec{\mathtt{st}}_{U_p}^{\lceil k} = \mathtt{state}|_{\mathtt{pt}_{U_p}}$
    if such a point does not exist **then**
      return ◁ Call on invalid input or event BAD-CP occurred
    end if
    if $\mathsf{Update}_{U_p,\tau}$ **then**
      $\mathsf{pointers} \leftarrow \mathsf{pointers} \,\|\, (U_p, \mathtt{pt}_{U_p})$
    end if
  end for
  Output $(\textsc{set-slack}, \mathsf{pointers})$ to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$
  ▷ Adjust the view of desynchronized parties
  $\mathsf{pointers} \leftarrow \varepsilon$
  $\mathsf{desyncStates} \leftarrow \varepsilon$
  for each desynchronized party $U_p \in \mathcal{P}_{DS}$ of round τ **do**
    if $\mathsf{Update}_{U_p,\tau} = \mathsf{false}$ **then**
      Set $\mathtt{pt}_{U_p}$ to be $|\vec{\mathtt{st}}_{U_p}^{\lceil k}|$
      $\mathsf{pointers} \leftarrow \mathsf{pointers} \,\|\, (U_p, \mathtt{pt}_{U_p})$

22: $\quad\quad$ $\mathsf{desyncStates} \leftarrow \mathsf{desyncStates} \,\|\, (U_p, \vec{\mathtt{st}}_{U_p}^{\lceil k})$
23: $\quad$ **end if**
24: $\quad$ Output (SET-SLACK, pointers) to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$
25: $\quad$ Output (DESYNC-STATE, desyncStates) to $\mathcal{G}_{\mathsf{Ledger}}^{\mathsf{Fair}}$
26: **end for**
27: **end procedure**

1: **procedure** IssueNewTransaction(P, τ, m) ◁ Issue dummy transactions.
2: Let $\mathcal{C}$ denote the simulated blockchain for party P
3: $h \leftarrow \mathrm{head}(\mathcal{C}^{\lceil k})$ and $st \leftarrow$ blockchain state associated with h
4: $\mathcal{S}_{\mathsf{pk}} \leftarrow$ all enclave public keys up to block with hash h
5: $\mathcal{S}_{\mathsf{pk}}^{\tau} \leftarrow f_{\mathsf{select}}(\mathcal{S}_{\mathsf{pk}}, m, \tau)$ and $m' \leftarrow |\mathcal{S}_{\mathsf{pk}}^{\tau}|$
6: **if** $\mathcal{S}_{\mathsf{pk}}^{\tau}$ contains only corrupted enclave keys **then**
7: $\quad$ **Abort** simulation: violation of good block ticket ◁ Event BAD-TICKET
8: **end if**
9: **for** i **from** 1 **to** m' **do**
10: $\quad$ $\mathsf{pk}_i \leftarrow \mathcal{S}_{\mathsf{pk}}^{\tau}[i], r_i \xleftarrow{\$} \{0,1\}^{\kappa}$
$\quad\quad$ ▷ Encrypt an all-zero string.
11: $\quad$ $ct_i \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_i, 0^m \,\|\, h; r_i)$ and remember r_i for equivocation later
12: **end for**
$\quad$ ▷ Generate fake NIZK proof, telling an all-zero string is a valid transactino.
13: $\pi \leftarrow \Pi_{\mathsf{NIZK}}.\mathsf{Prove}((h, st, (ct_1, \ldots, ct_{m'}), (\mathsf{pk}_1, \ldots, \mathsf{pk}_{m'})), (\mathtt{tx}, (r_1, \ldots, r_{m'})))$
14: **return** (π, ct, τ)
15: **end procedure**

E Mathmatical Facts

Theorem 10 (Chernoff bounds). *Suppose* $\{X_i : i \in [n]\}$ *are mutually independent Boolean random variables, with* $\mathbf{Pr}[X_i = 1] = p$*, for all* $i \in [n]$*. Let* $X = \sum_{i=1}^{n} X_i$ *and* $\mu = pn$*. Then, for any* $\delta \in (0, 1]$*, it holds that*

$$\mathbf{Pr}[X \leq (1-\delta)\mu] \leq e^{-\delta^2\mu/2} \text{ and } \mathbf{Pr}[X \geq (1+\delta)\mu] \leq e^{-\delta^2\mu/3}.$$

F Glossary

Variable	Description
κ	Security parameter; length of the random oracle output.
T	The target to successful solve a PoW.
m	The size of enclave public key that clients should encrypt to; $m = \Theta(\log^2 \kappa)$.
k	The common prefix parameter; $k = \Theta(\log^2 \kappa)$. When clients issue a transaction $\mathtt{tx}$, they asscoiate $\mathtt{tx}$ with the block on the tip of $\mathcal{C}_{\mathrm{loc}}^{\lceil k}$.

R	The recency parameter of profile blocks.
K	The size of a window that is used to decide transaction timestamp, counted by the number of blocks.
h_r	Number of RO queries made by honest parties at round r.
t_r	Number of RO queries made by corrupted parties at round r.
δ	Advantage of honest parties ($t \leq (1-\delta)h$).
f	Block generation rate; the probability at least one honest party succeeds in finding a PoW in a round.

Table 1: Main parameters of $\Pi^{\mathsf{Fair}}_{\mathsf{Ledger}}$.