A Composability Treatment of Bitcoin's Transaction Ledger with Variable Difficulty

Juan Garay[1] Yun Lu[2] Julien Prat[3] Brady Testa[4]
Vassilis Zikas[5]

[1] Texas A&M University, garay@tamu.edu.
[2] University of Victoria, yunlu@uvic.ca.
[3] CREST, Ecole Polytechnique, IP Paris, Julien.Prat@ensae.fr.
[4] Texas A&M University, btesta@tamu.edu.
[5] Georgia Tech, vzikas@gatech.edu.

Abstract. As the first proof-of-work (PoW) permissionless blockchain, Bitcoin aims at maintaining a decentralized yet consistent transaction ledger as protocol participants ("miners") join and leave as they please. This is achieved by means of a subtle PoW difficulty adjustment mechanism that adapts to the perceived block generation rate, and important steps have been taken in previous work to provide a rigorous analysis of the conditions (such as bounds on dynamic participation) that are sufficient for Bitcoin's security properties to be ascertained.
Such existing analysis, however, is *property-based*, and as such only guarantees security when the protocol is run **in isolation**. In this paper we present the first (to our knowledge) simulation-based analysis of the Bitcoin ledger in the dynamic setting where it operates, and show that the protocol abstraction known as the *Bitcoin backbone* protocol emulates, under certain participation restrictions, Bitcoin's intended specification. Our formulation and analysis extend the existing Universally Composable treatment for the fixed-difficulty setting, and develop techniques that might be of broader applicability, in particular to other composable formulations of blockchain protocols that rely on difficulty adjustment.

1 Introduction

Nakamoto's introduction of the Bitcoin protocol [30] put forth the novel notion of blockchains to solve the continuous (distributed) consensus problem (cf. [16]), also known in the distributed computing literature as *state machine replication* [35]. Nakamoto's protocol was designed to emulate the concept of a secure financial ledger: Users would be able to spend or receive BTCs provided they own the BTCs they are trying to use, while being prevented from "double-spending" them. Broadly speaking, the security of Bitcoin is guaranteed by the cryptographic primitive known as *Proof of Work* (PoW) [14], in which participants known as "miners" solve moderately difficult cryptographic puzzles to earn the right to insert the

next block. As of 2024, the Bitcoin protocol is still up and "humming," boasting a market cap of over USD 1.3T.

This nascent protocol quickly garnered the attention of wide range of researchers who investigated its properties and vulnerabilities. One of the first rigorous and property-based treatments of blockchains was presented by Garay *et al.* [17]. They examined the basic case of static difficulty (i.e., fixed but unknown number of participants) and defined two properties: *common prefix* and *chain quality*. A third property, *chain growth*, was also discussed in [17], and made explicit in [25]. They provide an abstract description of the Bitcoin protocol as a distributed protocol, termed the *Bitcoin backbone*, and then proceed to demonstrate that, under specific parameter assumptions, the protocol upholds the aforementioned properties in the cryptographic sense—i.e., they hold except with negligible probability in the security parameter. The initial formalization in [17] assumed a synchronous network. Subsequently, Pass *et al.* [31] examined the blockchain protocol also in the static participation setting, but in the more realistic bounded-delay network setting (sometimes also referred to as the "partially synchronous" setting [13]), where there is an (unknown) upper bound on the communication delay, proving similar properties.

The need for a composable ledger. The above first works paved the way for a deeper understanding of the Bitcoin protocol and proved tight conditions for their security properties. However, while such property-based analysis is an excellent first approximation for proving the security of cryptographic protocols, and has traditionally been the first step in any cryptographic study of novel protocol concepts, it is known that, especially in the presence of malicious protocol participants, who might arbitrarily misbehave, such a property-based treatment tends to miss important aspects of the functionality offered by the primitive being analyzed, which might affect its security under more adversarial scenarios [8,12]. In addition, such property-based analysis typically gives guarantees about the protocol running *in isolation*, which might no longer hold when the protocol is run alongside other protocols (or even another execution of the same protocol) and/or used by other protocols as a subroutine. This is particularly problematic for ledger protocols, such as Bitcoin, which, on one hand, has spawned several variants of itself running in parallel and has seeded a plethora of alternative blockchain-based ledger protocols, and on the other hand, it is meant to be utilized by higher-level constructions for many applications such as secure transactions, fair contract signing, NFTs, and more.

The literature has recognized the above shortcomings of the property-based treatment, and has proposed the *simulation paradigm* as the means to offer such stronger composable security guarantees. In more detail, in simulation-based security (cf. [7, 23, 27, 28, 34]), the first step is to completely describe the desired behavior that the distributed protocol/primitive *should* have even in the presence of an attacker, instead of listing some properties that *should not* be violated. The idea is to capture the desired behavior of the primitive by defining how a (centralized) *fully trusted* party—the so-called *ideal functionality*—could best offer the services of the distributed protocol to its users. The simulation-based paradigm, then, requires that any attack (captured by a cryptographic adversary) to the protocol can be simulated as an attack to (an invocation) of the ideal functionality. Formally, this means that for any adversary attacking the actual protocol, there exists an ideal adversary (the *simulator*) attacking the (ideal evaluation of) the functionality which yields an indistinguishable input/output behavior between the real and the ideal execution.

Importantly, by requiring that no (computationally bounded) distinguisher be able to discern between the real and ideal executions—not even one that spawns/observes executions of other protocols in parallel, or creates higher-level protocols that utilize the primitive being analyzed—the paradigm ensures that the protocol is secure for its specification/functionality in any context. This yields so-called *composition theorems* that solve the property-based definitions' nuisances pointed to above. Namely, the protocol's security guarantees remain the same even when the protocol being composed with other protocols (or itself), or when it is used as a subroutine in a higher-level protocol.

The first simulation-based analysis of the Bitcoin (backbone) protocol was done by Badertscher *et al.* [4], who used the state-of-the-art and by far most commonly used simulation-based security framework, namely, Canetti's Universal Composability (UC) framework [9]. In a nutshell, they captured, for the first time, the behavior of the Bitcoin backbone protocol by means of a UC functionality, called the *Bitcoin Ledger* functionality; provided a UC abstraction of the backbone protocol; and proved that it UC-emulates the above functionality. Along the way, they also provided the way to capture assumptions such as an honest majority of hashing power—which has been known to be necessary even for the property-based security analyses [17, 31]—by defining appropriate resources as UC *hybrid* functionalities[6], and providing a UC-friendly way for capturing

[6] Those are functionalities available to the protocol in the real world that abstract resources that the protocol might use.

such assumptions by means of so-called *functionality wrappers* (cf. [22])[7]. For example, the above assumption of an honest majority of hashing power was captured by abstracting the hash function as a random-oracle hybrid (functionality) and wrapping it to enforce that the adversarially controlled parties/miners (collectively) are allowed less queries than the honest miners (cf. [21]).

Beyond fixed-difficulty analysis. As discussed, all the above analyses assume an abstraction of Bitcoin in which the difficulty value for a PoW to be successful is fixed and the number of parties in the protocol is always within some fixed bound. While this analysis is very useful as a first step, it does not capture reality—indeed, because of its "permissionlessness," Bitcoin is a dynamic protocol, where participants join and leave the network unpredictably and at will. To account for this, Bitcoin changes the difficulty of the PoW according to the density of the recently produced blocks and the corresponding timestamps: If blocks are being produced too quickly, then the difficulty increases, while if blocks are being produced too slowly, then the difficulty decreases. It is known that there are attacks against such a difficulty adjustment mechanism, as presented in [6, 15, 29], making the formal examination of Bitcoin's blockchain protocol in the dynamic difficulty setting imperative.

In fact, shortly after the first property-based analyses of the Bitcoin backbone protocol, Garay *et al.* [18] were able to extend their treatment to this variable difficulty realm. In more detail, this was achieved in [18] by parameterizing the environment—which in [18] is in charge of spawning and removing parties—in such a way that it is (γ, s)-*respecting*. In a nutshell, within any s rounds of the protocol execution, the ratio between the largest and smallest number of participants must be bounded by γ. It is worth noting that this use of the term *environment* in [18] is not intended to capture simulation-based indistinguishability, and is therefore different from its utilization in UC—in fact, restricting the environment in such a way would preclude universal composition!

Thus, the above state of affairs left, once again, an important open question/gap in the literature:

> *Can we devise a UC treatment of (the) Bitcoin (backbone protocol) which incorporates, as in* [18]*, the dynamic participation of miners?*

[7] Recall that the primary purpose of this technique is to fine-tune those ideal functionalities that, while conveying the essence of the cryptographic task at hand, might lack the level of detail required by the particular setting or realization.

Our work answers the above question in the affirmative, by providing such an analysis. Interestingly, such an analysis needs to adapt the original UC Ledger functionality from [5] to make a list of features explicit, such as the trade-offs between liveness/chain-growth and chain quality. In passing, we note that while such a UC treatment of *proof-of-stake* blockchain protocols with dynamic availability exists [2], to our knowledge, no such analysis was previously done for PoW-based blockchains, which, given both the timing of the protocols and their market relevance, is somewhat surprising.

The balance of the paper is organized as follows. In Section 2 we present some basic blockchain terminology, network assumptions and the resources available to the protocol, and an abridged Universal Composability background. In Section 3 we present our UC abstraction of Bitcoin's PoW-based lottery system in the dynamic participation setting, while in Section 4 we present our formulation of the ideal ledger functionality supporting dynamic participation. Finally, in Section 5 we present our ledger protocol abstraction, $\mathsf{ModularLedger}$, which realizes the ideal ledger functionality above while using the PoW lottery functionality. Due to space limitations, complementary material, detailed specifications of functionalities and protocols, as well as proofs, are presented in the appendix.

2 Preliminaries

2.1 Blockchain Essentials

We present the fundamentals of PoW-based blockchains below. We refer the reader to [17, 33] for a more detailed discussion of the topic in the fixed difficulty setting, and to [18] in the variable difficulty setting.

Blockchain basics. A *blockchain* $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ is a (finite) sequence of blocks where each *block* $\mathbf{B}_i = \langle \mathsf{p}_i, \mathtt{st}_i, \mathsf{n}_i, T_i, t_i \rangle$ is a quintuple consisting of the (hash) *pointer* to the previous block p_i, the *state block* $\mathtt{st}_i$ containing transactions, the target for the hash value T_i, the timestamp t_i, and the *nonce* n_i. The *head* of a chain $\mathcal{C}$ is denoted $\mathsf{head}(\mathcal{C}) = \mathbf{B}_n$, and *length* $\mathsf{length}(\mathcal{C}) = n$. The sequence of the first $\mathsf{length}(\mathcal{C}) - k$ blocks of $\mathcal{C}$, i.e $(\mathcal{C}^{\lceil k} = \mathbf{B}_1, \ldots, \mathbf{B}_{length(\mathcal{C})-k})$ is denoted $\mathcal{C}^{\lceil k}$. Note that $\mathcal{C}^{\lceil k}$ itself is also a valid chain. If we have two chains C_1, C_2, and write $C_1 \preceq C_2$, we mean that C_1 is a prefix of C_2.

In a blockchain $\mathcal{C}$, the state $\vec{\mathtt{st}} = \mathtt{st}_1 \parallel \cdots \parallel \mathtt{st}_n$ contains the data of the ledger, i.e., transactions in Bitcoin. To allow for an abstract representation of this information, we map each state block $\vec{\mathtt{st}} = \mathsf{blockify}_{\mathbb{B}}(\vec{N})$

following the notation introduced in [26] and adopted in [4]. Here, $\vec{N}$ is a vector of transactions. A special type of initial block called the *genesis block* also exists. It is defined as $\mathbf{G} = \langle \perp, \mathtt{gen}, \perp, T_0, 0 \rangle$; $\mathtt{gen}$ is the genesis state, while T_0 refers to the initial target difficulty for the PoW.

To account for dynamic participation of parties and thus fluctuations in hashing power, Bitcoin performs a *target recalculation* procedure in which participants take the time stamps of the previous m blocks, and compare the average time to generate a block against the ideal time $\mathsf{b}_{\mathsf{tgt}}$ (in Bitcoin, $m = 2016$ and $\mathsf{b}_{\mathsf{tgt}} = 10$ minutes). Further, a *dampening filter* τ prevents the change in difficulty over a single epoch from exceeding a multiple of $[\frac{1}{\tau}, \tau]$. This is necessary to avoid certain attacks on the target recalculation function, such as the 'difficulty raising attack' [6], in which an adversary forges his timestamps to successfully manipulate the difficulty. In Bitcoin, τ is set to 4.

When discussing the validity of a chain, we make a distinction between *syntactic correctness* and *semantic correctness*. In a nutshell, syntactic correctness refers to the structure of the metadata (e.g., pointers, timestamps, difficulty, etc.), while semantic correctness ensures that the data itself is meaningful and properly formed, such as having valid transactions.

In more detail, *syntactic correctness* is defined with regards to a blockchain $\mathcal{C}$, an initial difficulty parameter (the 'target') $T_0 \in [2^\kappa]$ where κ is a security parameter, a hash function $H : \{0,1\}^* \to \{0,1\}^\kappa$ such that for all $i > 1$ blocks $\mathbf{B}_i \in \mathcal{C}$, we have that $H[\mathbf{B}_{i-1}] = \mathsf{p}_i$. That is, the hash of a block should be the same as the pointer in the next block. Note that in the dynamic participation setting, the difficulty is not a constant, and is dependent on the previous timestamps. We require that if $i < j$ then two timestamps t_i, t_j must satisfy $t_i < t_j$. Additionally, for chain $\mathcal{C}^{\lceil i}$ and the corresponding state $\vec{\mathtt{st}} = \vec{\mathtt{st}}_1 \parallel, \ldots, \parallel, \vec{\mathtt{st}}_i$, it holds that $H[\mathbf{B}_i] < \mathsf{gettarget}(\vec{\mathtt{st}})$ is true for all $i > 1$. The function $\mathsf{gettarget}$, described in Sec. 3.2, calculates the target based on previous timestamps and is used within the overall algorithm for checking syntactic correctness.

Blockchains can also have multiple blocks that extend a single block, branching out into multiple directions. For example, a fork at block $\mathbf{B}_{i-1}$ would have $H[\mathbf{B}_{i-1}] = \mathsf{p}_i = \mathsf{p}_{i'}$ where $i \neq i'$. Near universally, this is an undesirable event. This is naturally represented as a tree $\mathcal{T}$. Because participants in the protocol will potentially have differing views from each other, a tree structure is a natural representation of a party's overall view.

Semantic correctness is defined with respect to the state $\vec{\mathtt{st}}$ encoded in the chain. We follow the approach of [4] and model semantic correctness

via a predicate $\mathsf{ValidTx}_{\mathbb{B}}$, and an associated predicate $\mathsf{isvalidstate}$. Such predicates are dependent on the particular blockchain application being analyzed. In addition, semantic validity requires that the state begins with a genesis state, and that the beginning of each block contains a "coinbase" transaction which entitles the block miner to claim the block reward and fees.

A chain $\mathcal{C}$ is valid if it satisfies syntactic correctness and semantic correctness: for $\vec{\mathtt{st}}$ associated with $\mathcal{C}$, we have $\mathsf{validStruct}_{\mathbb{B}}^{m,T_0,\tau,\mathsf{b}_{\mathsf{tgt}}}(\mathcal{C}) \wedge \mathsf{isvalidstate}(\vec{\mathtt{st}})$ is true. In the variable difficulty setting, the "longest chain" is the chain which cumulatively has the largest difficulty, implemented by algorithm $\mathsf{maxvalid}$.

There have been property-based treatments of blockchains in the variable difficulty setting, notably [18] and the follow-up work in [19]. These elaborate on the need to restrict the rate at which parties join and leave the protocol in order to prove the protocol cryptographically secure (i.e., that the protocol satisfies certain properties [see Appendix B] except with negligible probability). Intuitively, the ideal properties rely on the difficulty remaining (somewhat) stable over the execution of the protocol. This requires the participation in the protocol to not fluctuate too much over a defined period. To this end, they propose the following notion of a (γ, s)-respecting sequence. Let n_r be the number of active parties in round r:

Definition 1 ([18]). *For $\gamma \in \mathbb{R}^+$, a sequence is called $(n_r)_{r\in\mathbb{N}}$ (γ, s)-*respecting *if for any set S of at most s consecutive rounds, $\max_{r\in S} n_r \leq \gamma \cdot \min_{r\in S} n_r$.*

That is, within any s number of consecutive rounds, the ratio between the minimum and maximum number of parties is bounded by γ. In [18], the sequence of parties executing the protocol $\mathbf{n} = \{n_r\}_{r\in\mathbb{N}}$ is thus required to be a (γ, s)-*respecting sequence*. This restriction is necessary to prove two properties: *common prefix* and *chain quality*, discussed further in Appendix B.

2.2 Protocol Resources and Network Assumptions

In this section we describe the resources, specified as ideal UC functionalities (see Section A), available to the protocol in the variable difficulty setting. We also describe assumptions about the network model.

– **The clock:** $\bar{\mathcal{G}}_{\text{CLOCK}}$ [4, 24] is a global functionality ensuring that the protocol proceeds in synchronized rounds.[8] Specific to the variable difficulty setting, the clock also provides the timestamps used in the target recalculation function. $\bar{\mathcal{G}}_{\text{CLOCK}}$ works by enforcing protocol participants to register with a REGISTER command. Note that this enforces a regularity condition necessary for global functionalities, and having individual parties register with the functionality before interacting with it. We refer to the session id for the clock as *cid*. In a nutshell, the clock CLOCK-READ keeps track of a counter τ_{cid}, associated with each different session, a party set $\mathcal{P}$, and variable d_P associated with each party P. After receiving inputs from every party in the session, the clock increments its value. Registered parties in a valid session can query the clock by sending a message $(\text{CLOCK-READ}, cid)$, to whom $\bar{\mathcal{G}}_{\text{CLOCK}}$ returns $(\text{CLOCK-READ}, cid, \tau_{cid})$.
– **The random oracle:** $\mathcal{F}_{\text{RO}}$ models an idealized hash function. $\mathcal{F}_{\text{RO}}$ keeps an internal table H of input-output pairs. Upon receiving input $x \in \{0,1\}^*$ from a party, the functionality queries $H[x]$. If the value is present, then it returns $H[x]$. If it is not, then a value y is sampled uniformly at random from $\{0,1\}^\kappa$, and is saved in the table as $H[x] = y$. This value is returned to the party that requested it. In Bitcoin, $\mathcal{F}_{\text{RO}}$ is used in particular for the computation and verification of PoWs. Restricting the number of queries to the oracle is accomplished by means of a wrapper $\mathcal{W}^q(\mathcal{F}_{\text{RO}}^\kappa)$, which ignores queries beyond the allotted amount.
– **The message diffusion functionality:** $\mathcal{F}_{\text{DIFF}}$. This functionality allows parties to broadcast messages across the Bitcoin network, to share new transactions and extensions to the blockchain. $\mathcal{F}_{\text{DIFF}}$ maintains a party set, and a list of messages for which it keeps track of the current time remaining before the message can be received (see below).

The above ideal functionalities are described in detail in Appendix D.

We use $\mathcal{F}_{\text{DIFF}}$ presented above as our mechanism for communication among parties. Because the UC framework natively operates in the asynchronous model, where messages are not guaranteed to be delivered in order, it is necessary to model eventual, bounded delivery—the so-called 'bounded-delay network' model (cf. [13]; see also [24]), where there exists

[8] The underlying assumption here, following [4, 19], is that Bitcoin's timestamping mechanism works and can be abstracted as such ideal functionality. See [19] for its detailed analysis.

an unknown delay Δ in the delivery of messages, measured in number of rounds. $\mathcal{F}_{\text{DIFF}}$ is based on the model described in [4, 20, 32]. In particular, this means that Δ is unknown to the participants in the protocol, an immediate consequence of which is that a protocol operating in this circumstance cannot use Δ directly (say, as a time-out). Our formulation incorporates this fact, even under arbitrary protocol composition. Initially, the delay is set to 0. The adversary then sets the delay parameter Δ by passing (SET-DELAY, sid, n) to $\mathcal{F}_{\text{DIFF}}$, which it stores.

Any message sent by a party $P_s \in \mathcal{P}$ to $\mathcal{F}_{\text{DIFF}}$ with command (MULTICAST, sid, m), is stored internally in a buffer with a timer associated with each copy of the message to be sent to all parties. The adversary is forwarded the content of the message, along with all of the message IDs. It then can swap message IDs and enforce arbitrary delays (up to Δ).

3 Modeling Bitcoin's Dynamic Participation in the UC Framework

We introduce the abstraction of the PoW-based lottery system in the dynamic participation and variable difficulty setting. We follow the convention in [4] to incorporate these aspects into a 'State Exchange' functionality ($\mathcal{F}_{\mathtt{STX}}(\mathcal{P}, \Delta, p_H, p_A)$; see below for details). This modular approach allows for simplified analysis. Here, this means that we first analyze the lottery procedure before applying it to the Bitcoin backbone protocol. When parameters are clear from context or unneeded, we also refer to this as $\mathcal{F}_{\mathtt{STX}}$. At a high level, $\mathcal{F}_{\mathtt{STX}}$ handles extending the chain state, and propagating the resulting chains to the other participants while obeying the delay constraint. Upon receiving (SUBMIT-NEW, $sid, \vec{\mathtt{st}}, \mathtt{st}$) from a party, $\mathcal{F}_{\mathtt{STX}}$ runs a Bernoulli experiment with the given probability, extending the state on success. $\mathcal{F}_{\mathtt{STX}}$ also stores an internal buffer $\vec{M}$ of states that parties may not have yet received; those parties can send (FETCH-NEW, sid) to retrieve them if the message delay has been satisfied.

The formulation in [4], however, only applies to the fixed number of participants and difficulty setting. Here we extend the formulation to the dynamic participation/variable difficulty setting by applying the "functionality wrapping" technique (cf. [22]; see also [21]). We will then show that this (wrapped) functionality is UC-realized by a protocol we call StateExchange, in the $(\mathcal{W}^q(\mathcal{F}_{\text{RO}}^{\kappa}), \mathcal{F}_{\text{DIFF}})$-hybrid model. Later on, in Section 4, we will use this functionality to realize, under certain constraints, a ledger functionality that captures Bitcoin in the permissionless setting

where it is intended to operate. A glossary of parameters that will be used in our specifications and analyses can be found in Table 1 (App. C).

3.1 The Lottery Mechanism

As mentioned above, the 'State Exchange' functionality $\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A)$ from [4] for the static setting handles the process of extending the state of the blockchain as well as propagating the result of the extension to the other participants. In particular, it takes as parameters $\mathcal{P}$, the party set for that invocation, Δ, the maximal delay for a message, and p_H (p_A), the probability of an honest party (resp. corrupted party) successfully querying the functionality to extend the state. (The full specification of the static functionality can be found in Appendix D.)

Next, we introduce our wrapped functionality, which will manage the current set of active parties, compute the target value for the PoW accordingly, and capture the restrictions on the environment (recall Definition 1) that will ensure the realization of the underlying functionality. Let $\mathsf{stx\text{-}params} = \{T_0, m, \mathsf{b_{tgt}}, \tau, \gamma, s\}$. The full specification of our wrapper functionality, $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A))$, appears below. We proceed to describe its various aspects.

Functionality $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A))$

The functionality maintains a party set $\mathcal{P} \leftarrow \emptyset$, and a set $C = \{C_1, \ldots, C_n\}$, where $C_i = |\mathcal{P}|$ at round i. The functionality maintains a set of trees for each party $\mathcal{T}_P$ of 3-tuple $(\vec{\mathtt{st}}, \vec{T}, \vec{\tau_s})$, where each $\vec{\mathtt{st}}$ is unique. The functionality also maintains a value $\Delta \in \mathbb{N}$, initialized to 0. It also registers with the global clock functionality $\bar{\mathcal{G}}_{\text{CLOCK}}$

Setting the delay :
Upon receiving (SET-DELAY, sid, d) from the adversary $\mathcal{A}$, if $d \in \mathbb{N}$ and SET-DELAY has never been received by this functionality, then set $\Delta = d$. Return (SET-DELAY, sid, ok)

Registration :
Upon receiving any REGISTER or DE-REGISTER command, sends (CLOCK-READ, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$ to receive the answer (CLOCK-READ, cid, τ_L)

- Upon receiving (REGISTER, sid) from some party P (or from $\mathcal{A}$ on behalf of a corrupted P) Let i refer to the current round
  1. If $|\mathcal{P}| + 1 > \gamma \cdot C_j$ for any $j \in [\tau_L - s, \tau_L]$, then return (REGISTER, $sid, \perp$). Otherwise, proceed.

2. Set $\mathcal{P} = \mathcal{P} \cup \{P\}$, set $C[\tau_L] = |\mathcal{P}|$ and initialize the tree $\mathcal{T}_P \leftarrow \mathtt{gen}$ where each rooted path corresponds to a valid state the party has received. Return $(\text{REGISTER}, sid, P)$ to the caller.

– Upon receiving $(\text{DE-REGISTER}, sid)$ from some party $P \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $P \in \mathcal{P}$)
1. If $|\mathcal{P}| - 1 < \gamma \cdot C_j$ for any $j \in [\tau_L - s, \tau_L]$, then do nothing. Otherwise, proceed.
2. Set $\mathcal{P} := \mathcal{P} \setminus \{P\}$, set $C[\tau_L] = |\mathcal{P}|$, and return $(\text{DE-REGISTER}, sid, P)$ to the caller.

Submit/receive new states :

– Upon receiving $(\text{SUBMIT-NEW}, sid, \vec{\mathtt{st}}, \mathtt{st})$ from some participant $P_s \in \mathcal{P}$, if $\mathsf{isvalidstate}_{\mathbb{B}}(\vec{\mathtt{st}} \parallel \mathtt{st}) = 1$ and $(\vec{\mathtt{st}}, \cdot, \cdot) \in \mathcal{T}_P$, then do the following:
1. Set $T \leftarrow \mathsf{gettarget}_{m,T_0,\tau,\mathsf{b}_{\mathsf{tgt}}}(\vec{\mathtt{st}})$
2. Set $p_H \leftarrow 1 - (1 - \frac{T}{2^\kappa})^q$, $p_A \leftarrow \frac{T}{2^\kappa}$ and forward $(\text{SUBMIT-NEW}, sid, \vec{\mathtt{st}}, \mathtt{st})$ to $\mathcal{F}_{\text{StX}}(\mathcal{P}, \Delta, p_H, p_A)$
3. Upon response $(\text{SUCCESS}, sid, B)$, if $B = 1$, then add $(\vec{\mathtt{st}} \parallel \mathtt{st}, T, \tau_L)$ to $\mathcal{T}_{\mathcal{P}}$ and send $(\text{CONTINUE}, sid)$ to $\mathcal{F}_{\text{StX}}(\mathcal{P}, \Delta, p_H, p_A)$.
4. Send $(\text{SUCCESS}, sid, B, T, \tau_L)$ to P_s

– Upon receiving any other input, forward the request to $\mathcal{F}_{\text{StX}}(\mathcal{P}, \Delta, p_H, p_A)$, and return its response.

Because it is necessary to capture certain restrictions to guarantee the basic blockchain properties—common prefix, chain quality and chain growth (cf. App. B)—we do so by having the wrapper reject any communication that would violate these restrictions. This applies in particular to the case of the (γ, s)-respecting sequence, which occurs during registration and de-registration; if the environment activates participants so that they attempt to register/deregister too quickly, $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{StX}}(\mathcal{P}, \Delta, p_H, p_A))$ ignores the message. The set C provides the necessary context from previous rounds so that the wrapper knows how many participants were present, and is able to determine if a violation occurs.

The wrapper handles the party management, passing off the current party set as a parameter to $\mathcal{F}_{\mathbf{StX}}$ to handle the message handling and state extension. Upon receiving a request to attempt to extend the state, $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{StX}})$ first calculates the target difficulty from the provided $\vec{\mathtt{st}}$. From this, $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{StX}})$ calculates the appropriate probabilities p_H, p_A, then passes off the result to $\mathcal{F}_{\mathbf{StX}}(\mathcal{P}, \Delta, p_H, p_A)$.

The wrapper also handles timestamps and PoW target values by storing the corresponding entry for each $\mathtt{st}$ in the tree. This is necessary to properly handle any calculations of the PoW difficulty in future calls to the ideal functionality, as well as be able to calculate the probabilities p_A, p_H.

Thus, from the perspective of $\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A)$, the functionality operates as if it were in the static-difficulty case per invocation. Upon receiving message (SUBMIT-NEW, $sid, \vec{\mathtt{st}}, \mathtt{st}$) from the wrapper $\mathcal{W}^{\mathsf{stx\text{-}params}}(\cdot)$, functionality $\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A)$ samples a Bernoulli distribution with the probabilities passed in as a parameter. If $B = 1$, the party 'wins' the lottery, and the result is forwarded back to the wrapper, which, upon responding to continue, $\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A)$ creates new messages with separate message ids into its internal buffer $\vec{M}$.

Regarding message delays, the adversary $\mathcal{A}$ can enforce a delay up to Δ on the state propagation messages by sending (DELAY, sid, T, mid) to the wrapper, which forwards it to $\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A)$ as a parameter. Additionally, the adversary is able to swap messages in the buffer, given two $\mathsf{mid}, \mathsf{mid}'$, provided they exist. Note that modeling the delay Δ as a value that the adversary sets instead of a parameter to the functionality is necessary since we are operating under the assumption that Δ itself is finite but **unknown**. Having Δ be fixed as a parameter would leak information about the delay and as such violate this assumption. Passing off Δ from the wrapper to $\mathcal{F}_{\mathbf{STX}}$ does not expose the value to the participants, as all interaction with the functionality occurs through the wrapper, which instead keeps it hidden.

Finally, note that the honest and adversarial probabilities are not equal, similarly to the analysis in [17] and [4]. This is because the adversary is not obligated to follow the protocol. The honest parties do follow the protocol, and this means that their probability of success is the probability of receiving at least one success within the q PoW-solving (mining) attempts. When the probability of solving a PoW is p, this results in the expression $1 - (1 - p)^q$.

3.2 Implementing the Lottery Mechanism

To implement the State Exchange functionality, it is necessary for the protocol to implement the PoW system, as well as to handle the message diffusion. As usual, the party is expected to register and deregister with the relevant functionalities, $\mathcal{F}_{\text{DIFF}}^{bc}$ and $\mathcal{W}^q(\mathcal{F}_{\text{RO}}^{\kappa})$. Toward the first aim, upon receiving (SUBMIT-NEW, $sid, \vec{\mathtt{st}}, \mathtt{st}$), the party first checks the validity of the state, and if it finds a corresponding state in its tree, then it attempts to run the hash calculation q times (recall that it is assumed that the parties have a bound q on the number of RO calls per round [17,18]).

If the party 'wins' the PoW lottery through the RO calls, the extended state is appended to the tree $\mathcal{T}$. The process then propagates the message via $\mathcal{F}_{\text{DIFF}}^{bc}$. To implement the fetching aspects of the State Exchange

functionality, the party sends (FETCH, sid) to $\mathcal{F}_{\text{DIFF}}^{\mathsf{bc}}$, parses the output for valid chains, adds them to $\mathcal{T}$, and then extracts the states from the chains. The protocol is depicted in Appendix E.1.

Note that in contrast to [4], protocol StateExchange takes additional parameters as well, which are used by the extendchain algorithm. Upon executing the protocol for the first time, the party P is expected to first register with $\mathcal{F}_{\text{DIFF}}^{bc}$ and $\mathcal{W}^q(\mathcal{F}_{\text{RO}}^{\kappa})$. StateExchange uses two subroutines: isvalidstate and extendchain, both introduced in [4]. The former is responsible for taking a state vector as input and ensuring that all blocks in the state are composed of valid transactions, including the first transaction which is required to be a coinbase transaction. isvalidstate is presented in Appendix E.4 ; extendchain is depicted in Appendix E.7. extendchain takes in the epoch length m, the expected block generation time $\mathsf{b_{tgt}}$, the initial difficulty T_0, the number of RO queries per round q, the dampening factor τ, and the chain and state $\mathcal{C}, \mathtt{st}$ respectively. StateExchange uses the algorithm gettarget, which first computes the new target value based on the previous timestamps, and for q attempts, the party calculates the hash of the block, which is accepted if its value is below the previously derived target. The party will also periodically fetch messages from the network to receive new chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ that other parties have been working on, storing the valid chains in its own internal tree. We present the target recalculation algorithm gettarget below.

We now state the main result of this section. (Proof in Appendix F.)

Theorem 1. *Protocol* $\mathsf{StateExchange}^{q,T_0,\tau,m}(P)$ *UC-realizes functionality* $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}})$ *in the* $(\mathcal{W}^q(\mathcal{F}_{\text{RO}}^{\kappa}), \mathcal{F}_{\text{DIFF}}^{\mathsf{bc}})$*-hybrid model.*

Algorithm $\mathsf{gettarget}^{m,T_0,\tau,\mathsf{b_{tgt}}}(\vec{\mathtt{st}})$

1. Calculate $i = \lfloor \frac{|\vec{\mathtt{st}}|}{m} \rfloor$
2. If $i = 0$, then return $T = T_0$
3. Parse $\vec{\mathtt{st}} = (st_1, T_1, t_1)|| \ldots ||(st_n, T_n, t_n)$
4. Set $\mathsf{T_{diff}} = t_{i \cdot m - 1} - t_{(i-1) \cdot m}$
5. Set $T = \frac{\mathsf{T_{diff}}}{m \cdot \mathsf{b_{tgt}}} \cdot T_{(i-1) \cdot m}$.
6. If $T > \tau T_{(i-1) \cdot m}$, then set $T = \tau T_{(i-1) \cdot m}$. If $T < \tau T_{(i-1) \cdot m}$, then set $T = \frac{\tau}{T_{(i-1) \cdot m}}$.
7. Return T

4 The Ledger Functionality with Dynamic Participation

In this section we present our formulation of the ledger functionality supporting dynamic participation. We follow a similar approach to the pre-

vious section's, by appropriately "wrapping" the ledger functionality for the static setting in [4], which participants interact with instead. In a nutshell, [4]'s static functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$ abstracts the process of maintaining and extending a distributed ledger. (The full specification of $\bar{\mathcal{G}}_{\text{LEDGER}}$ is presented in Appendix D.)

Let $\mathsf{ledger\text{-}params} = \{\texttt{windowSize}, \gamma, \mathsf{Validate}, \mathsf{ExtendPolicy}, \mathsf{Blockify}, \texttt{predict-time}\}$, where $\texttt{windowSize}, s \in \mathbb{N}$ and $\gamma \in \mathbb{R}$. We refer to our dynamic-participation functionality as $\mathcal{W}^{\mathsf{ledger\text{-}params}}(\bar{\mathcal{G}}_{\text{LEDGER}})$. The list of parameters capture the restrictions that are sufficient for the protocol in Section 5 below to UC-realize the ledger. The full specification of $\mathcal{W}^{\mathsf{ledger\text{-}params}}(\bar{\mathcal{G}}_{\text{LEDGER}})$ is given below, followed by an explanation of its various aspects.

Functionality $\mathcal{W}^{\mathsf{ledger\text{-}params}}(\bar{\mathcal{G}}_{\text{LEDGER}})$

Parameters: $\texttt{windowSize}, s \in \mathbb{N}, \gamma \in \mathbb{R}$; Algorithms $\mathsf{Validate}$, $\mathsf{ExtendPolicy}$, $\mathsf{Blockify}$, $\texttt{predict-time}$.

Clock-time: The functionality maintains a variable τ_L that is kept in-sync with clock-time: Upon any activation (and thus also initialization), the ledger first sends $(\text{CLOCK-READ}, cid)$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$ to receive the answer $(\text{CLOCK-READ}, cid, \tau_L)$, then proceeds with the remaining actions.

Variables and initialization: The functionality initializes state, s_{ep}, $\texttt{NxtBC}$, $\vec{\mathcal{I}}_H^T \leftarrow \epsilon$, $\mathsf{buffer} \leftarrow \emptyset$, $\Delta = 0$ as well as party sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS} \leftarrow \emptyset$. It also keeps track of the cardinality per round of the party sets, indexed by i i.e $(\mathcal{P}_i, \mathcal{H}_i, \mathcal{P}_{DS,i})$ The functionality also keeps track of a set $C = \{C_1, \ldots, C_n\}$, where $C_i = |\mathcal{P}|$ at round i

Setting Delay :
Upon receiving $(\text{SET-DELAY}, sid, d)$ from the adversary $\mathcal{A}$, if $d \in \mathbb{N}$ and SET-DELAY has never been received by this functionality, then set $\Delta = d$.

γ, s **Restriction Enforcement:**
- Upon receiving $(\text{REGISTER}, sid)$ from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), if $|\mathcal{P}| + 1 > \gamma \cdot C_j$ for any $j \in [\tau_L - s, \tau_L]$, then do nothing. Otherwise, forward the request to $\bar{\mathcal{G}}_{\text{LEDGER}}$.
- Upon receiving $(\text{DE-REGISTER}, sid)$ from some party $P \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $P \in \mathcal{P}$), if $|\mathcal{P}| - 1 < \gamma \cdot C_j$ for any $j \in [\tau_L - s, \tau_L]$, then do nothing. Otherwise, forward the request to $\bar{\mathcal{G}}_{\text{LEDGER}}$.

Party Set Management:
Upon any other input I received from a party $P_i \in \mathcal{P}$ or from the adversary $\mathcal{A}$ the following steps are taken:

1. If $P_i \in \mathcal{H}$ or if I is a corruption message from $\mathcal{A}$ targeting $P_i \in \mathcal{H}$, then update $\vec{\mathcal{I}}_H^T \leftarrow \vec{\mathcal{I}}_H^T \parallel (I, P_i, \tau_L)$. If a party P_i gets corrupted, additionally update $\mathcal{H} \leftarrow \mathcal{H} \setminus \{P_i\}$, $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \setminus \{P_i\}$

2. Let $\hat{P} := \{P \in \mathcal{P}_{DS} | \tau_P^{reg} < \tau_L - 4\Delta\}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \{\hat{P}\}$.
3. If the message was not received from $\bar{\mathcal{G}}_{\text{LEDGER}}$, forward the message, along with the appropriate parameters and variables that this functionality stores.

Restrictions

1. Upon any message (RESOURCE-CHECK, sid, p, x) from $\bar{\mathcal{G}}'_{\text{LEDGER}}$, where $x \in \{0,1\}^*$, $p \in \{\text{LEDGER-STARTUP}, \text{LEDGER-GROWTH}\}$
   - If $p =$ LEDGER-STARTUP, parse x as $\vec{\tau}_{\mathtt{state}}, \vec{\mathtt{hf}}, \vec{T}$, do the following:
     (a) Find the minimum interval in the timestamps during startup. Let $j \in [0, \tau_L - \mathtt{maxTime}_{\text{window}}], i \in [j + \mathtt{maxTime}_{\text{window}}, \tau_L]$. Then $lr \leftarrow \min_{j,i}(\frac{\mathtt{windowSize} + |n \in [|\vec{\tau}_{\mathtt{state}}|] : j \leq \vec{\tau}_{\mathtt{state}}[n] \leq i|}{i-j+1})$
     (b) $ar \leftarrow \sum_{i=s \wedge \vec{\mathtt{hf}}[i]=1}^{t}(T_i)$
     (c) If $lr < \frac{\mathtt{windowSize}}{\mathtt{maxTime}_{\text{window}}} \vee ar > \delta - 3\epsilon$, then set $\mathsf{inv} = 1$
   - If $p =$ LEDGER-GROWTH, parse x as $\vec{\tau}_{\mathtt{state}}, \vec{\mathtt{hf}}, \vec{T}$, do the following:
     (a) For all (s,t) such that $s \in [1, |\vec{\tau}_{\mathtt{state}}| - \ell + 1]$ and $t \in [s + \ell - 1 \, |\vec{\tau}_{\mathtt{state}}|]$ set $\mathsf{inv} \leftarrow 1$ if $\sum_{i=s}^{t}(T_i]) < \mathtt{chaingrowth}(s,t)$ for any s,t
     (b) $ar \leftarrow \max_{s=1,\ldots|\vec{\tau}_{\mathtt{state}}|-(\ell+2\Delta)+1; t=s+(\ell+2\Delta)-1 \ldots,|\vec{\tau}_{\mathtt{state}}|}$ $(\sum_{i=s \wedge \vec{\mathtt{hf}}[i]=1}^{t}(T_i))$
     (c) If $ar < \delta - 3\epsilon$, then set $\mathsf{inv} = 1$
2. Return (RESOURCE-CHECK, inv) to $\bar{\mathcal{G}}_{\text{LEDGER}}$

Our ledger formulation also enforces the γ, s restrictions in order to prevent too many parties from entering or leaving in a short time. The wrapper also handles party management for the honest parties $\mathcal{H}$, the party set $\mathcal{P}$, and the honest but de-synchronized parties $\mathcal{P}_{DS}$. The latter is necessary to model, as a de-synchronized party is effectively unable to extend the blockchain with its computing resources. Worse, under the model, the parties may end up inadvertently extending the adversary's chain.

Assuming the above restrictions are satisfied, our wrapper forwards all of the queries it receives to the ledger. In $\mathsf{ExtendPolicy}$ and $\mathsf{DefaultExtension}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \mathsf{buffer}, \mathtt{NxtBC}, s_{ep})$, the ledger functionality forwards the necessary information back to the wrapper in order to compute whether the proposed ledger extension has a high-enough number of honestly generated blocks, and that the ledger is not introducing blocks too slowly. $\mathsf{DefaultExtension}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \mathsf{buffer}, \mathtt{NxtBC}, s_{ep})$ is triggered upon an adversarial attempt to include an extension that is not admissible (e.g, by including too many adversarial blocks in one period). This default extension is disadvantageous for the adversary, and so the adversary will attempt to prevent its invocation.

The adversary can also set the state slackness via the command SET-SLACK, supplemented with a list of (P_i, state_i) pairs. The adversary is allowed to

violate this constraint for the case of de-synchronized parties, as they do not yet have a complete view of what is going on in the network.

5 Bitcoin as a Variable-Difficulty Ledger Protocol

In this section we present our ledger protocol, ModularLedger, which realizes the functionality $\mathcal{W}^{\mathsf{ledger\text{-}params}}(\bar{\mathcal{G}}_{\text{LEDGER}})$ above. Due to space limitations, the protocol's full specification is presented in Appendix E.2; here we give a high-level overview. The protocol assumes access to the $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}})$ functionality from Section 3.2. A party first sends (REGISTER, sid) to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}})$ and awaits its response. If the party receives an affirmative response (REGISTER, sid, P), the party proceeds to register with $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}$ and $\bar{\mathcal{G}}_{\text{CLOCK}}$. Otherwise, the party is considered to be unregistered. This process is similar for de-registration.

The reason this sequential approach is necessary is to enforce the (γ, s)-respecting sequence on the party protocol. In order to keep the $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}, \bar{\mathcal{G}}_{\text{CLOCK}}$ functionalities canonical, we enforce the (γ, s) requirement on the $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}})$ side. If we had allowed the registration for $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}, \bar{\mathcal{G}}_{\text{CLOCK}}$ to occur simultaneously with the registration for $\mathcal{W}^{\mathsf{stx\text{-}params}}$ $(\mathcal{F}_{\mathbf{STX}})$, then a mismatch could occur in party sets for the functionalities when $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}})$ rejects registration, rendering the party registered with some of the functionalities, but not the others.

To extend the ledger, parties first execute LedgerMaintenance. Here, the party first runs the FetchInformation sub-protocol, where the party fetches the most recent messages from $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}})$ and then updates its own local chain. Then, the party gathers the transactions in its buffer, formats them with a coinbase transaction, and encodes them in a state. The ExtendState sub-protocol is then invoked, where the party attempts to query the (wrapped) lottery functionality to determine if it is allowed to extend the state. Of course, this operation corresponds to solving successful PoWs. As in [4], the protocol is run in an (MAINTAIN-LEDGER, sid, $minerID$)- interruptible manner. The idea is that when a party passes a message to another machine, it loses control of the execution, despite the fact that the remaining steps are necessary for the correctness of the protocol. This interruptiblity requirement can be satisfied by storing an anchor; for a protocol of m steps, if we relinquish activation on step $i < m$, then we store $i + 1$ in the associated anchor; otherwise, we store $i = 2$. We are now ready to state our main theorem.

Theorem 2. *Assume that the parameter constraints specified in Table 2 are satisfied. Then protocol* $\mathsf{ModularLedger}^{T_0,m,\mathsf{b}_{\text{tgt}},q,\tau,\mathit{windowSize}}$ *UC-realizes* $\mathcal{W}^{\mathsf{ledger\text{-}params}}(\bar{\mathcal{G}}_{\text{LEDGER}})$ *in the* $(\bar{\mathcal{G}}_{\text{CLOCK}}, \mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}}), \mathcal{F}_{\text{DIFF}})$*-hybrid model.*

References

1. C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. Cryptology ePrint Archive, Paper 2020/1209, 2020.
2. C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 913–930. ACM Press, Oct. 2018.
3. C. Badertscher, J. Hesse, and V. Zikas. On the (ir)replaceability of global setups, or how (not) to use a global ledger. Cryptology ePrint Archive, Paper 2020/1489, 2020.
4. C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In J. Katz and H. Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 324–356. Springer, 2017.
5. C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 324–356. Springer, Heidelberg, Aug. 2017.
6. L. Bahack. Theoretical bitcoin attacks with less than half of the computational power (draft). Cryptology ePrint Archive, Paper 2013/868, 2013. `https://eprint.iacr.org/2013/868`.
7. J. Camenisch, S. Krenn, R. Küsters, and D. Rausch. iUC: Flexible universal composability made simple. In S. D. Galbraith and S. Moriai, editors, *ASIACRYPT 2019, Part III*, volume 11923 of *LNCS*, pages 191–221. Springer, Heidelberg, Dec. 2019.
8. R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, Jan. 2000.
9. R. Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.
10. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. Cryptology ePrint Archive, Paper 2006/432, 2006.
11. R. Canetti, A. Jain, and A. Scafuro. Practical UC security with a global random oracle. In G.-J. Ahn, M. Yung, and N. Li, editors, *ACM CCS 2014*, pages 597–608. ACM Press, Nov. 2014.
12. R. Cohen, J. Garay, and V. Zikas. Completeness theorems for adaptively secure broadcast. In H. Handschuh and A. Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 3–38, Cham, 2023. Springer Nature Switzerland.
13. C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.
14. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E. F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.
15. I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable, 2013.
16. J. A. Garay and A. Kiayias. SoK: A consensus taxonomy in the blockchain era. In S. Jarecki, editor, *CT-RSA 2020*, volume 12006 of *LNCS*, pages 284–318. Springer, Heidelberg, Feb. 2020.

17. J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.
18. J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol with chains of variable difficulty. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 291–323. Springer, Heidelberg, Aug. 2017.
19. J. A. Garay, A. Kiayias, and N. Leonardos. Full analysis of nakamoto consensus in bounded-delay networks. *IACR Cryptol. ePrint Arch.*, page 277, 2020.
20. J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. *J. ACM*, 71(4):25:1–25:49, 2024.
21. J. A. Garay, A. Kiayias, R. M. Ostrovsky, G. Panagiotakos, and V. Zikas. Resource-restricted cryptography: Revisiting MPC bounds in the proof-of-work era. In A. Canteaut and Y. Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2020.
22. J. A. Garay, P. D. MacKenzie, and K. Yang. Strengthening zero-knowledge protocols using signatures. *J. Cryptol.*, 19(2):169–209, 2006.
23. S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.
24. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *Theory of Cryptography Conference*, pages 477–498. Springer, 2013.
25. A. Kiayias and G. Panagiotakos. Speed-security tradeoffs in blockchain protocols. Cryptology ePrint Archive, Paper 2015/1019, 2015. `https://eprint.iacr.org/2015/1019`.
26. A. Kiayias, H.-S. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*, page 705–734, Berlin, Heidelberg, 2016. Springer-Verlag.
27. Y. Lindell. *How to Simulate It – A Tutorial on the Simulation Proof Technique*, pages 277–346. Springer International Publishing, Cham, 2017.
28. U. Maurer and R. Renner. Abstract cryptography. In B. Chazelle, editor, *ICS 2011*, pages 1–21. Tsinghua University Press, Jan. 2011.
29. D. Meshkov, A. Chepurnoy, and M. Jansen. Revisiting difficulty control for blockchain systems. Cryptology ePrint Archive, Paper 2017/731, 2017. `https://eprint.iacr.org/2017/731`.
30. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `http://bitcoin.org/bitcoin.pdf`.
31. R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In J.-S. Coron and J. B. Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 643–673, Cham, 2017. Springer International Publishing.
32. R. Pass, L. Seeman, and A. Shelat. Analysis of the blockchain protocol in asynchronous networks. In J. Coron and J. B. Nielsen, editors, *Advances in Cryptology*

- EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II, volume 10211 of *Lecture Notes in Computer Science*, pages 643–673, 2017.

33. R. Pass and a. shelat. Micropayments for decentralized currencies. In I. Ray, N. Li, and C. Kruegel, editors, *ACM CCS 2015*, pages 207–218. ACM Press, Oct. 2015.
34. B. Pfitzmann and M. Waidner. Composition and integrity preservation of secure reactive systems. In D. Gritzalis, S. Jajodia, and P. Samarati, editors, *ACM CCS 2000*, pages 245–254. ACM Press, Nov. 2000.
35. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

A Universal Composability Background

Universal Composability (UC), proposed by Ran Cannetti, is a framework for analyzing the security of cryptographic protocols under arbitrary composition [9]. We present here an executive summary . In short, a proof of universal composability aims to show that the execution of a real-world protocol Π is indistinguishable from an idealized functionality $\mathcal{F}$ with a simulator $\mathcal{S}$ to an environment $\mathcal{Z}$. Typically, $\mathcal{F}$ is modeled as a trusted party that cannot be corrupted by the adversary.

UC models the environment, protocol participants/parties, the simulator, and the adversary as Interactive Turing Machines (ITM), which are provided special externally writable tapes. These tapes model communication between participants, the environment, and the adversary $\mathcal{A}$—they include an input tape, subroutine-output tape, and a backdoor tape accessible by $\mathcal{A}$.

Universal Composability also has the ability to model dynamic participation. Various protocols, including Bitcoin, are designed to operate so that parties come and go, as opposed to being fixed at the start of execution. An ITM can spawn another machine by calling an external-write command with a forced-write flag.

Two important instructions are also included: external-write and read-next-message. The former instruction potentially places the content of its output message tape onto the tape of another ITM, or potentially spawns a new ITM who will have the output placed on its tape. The latter instruction, as the name implies, allows the machine to automatically place any of its tape heads in position to read the next message.

An *instance* M of an ITM $\mathcal{M}$, also referred to as an ITI, contains the contents of its identity tape (the code μ and identity string id), implying that $M = (\mu, id)$. We say that a *configuration* of an ITM $\mathcal{M}$ describes all of the contents of its tapes, including current pointer locations. This can be viewed as a snapshot of the state of computation of the ITM. We also define a configuration of an instance M to be a configuration of an ITM $\mathcal{M}$ if $M = (\mu, id)$ is consistent with the identity tape in the ITM $\mathcal{M}$. The model itself contains no explicit assumptions on synchronicity; instead such assumptions are modelled as ideal functionalities which may be later composed with other protocols.

UC uses probability ensembles to describe the output of an execution of a protocol π. $\text{EXEC}_{\pi,A,\mathcal{Z}}(x)$ is to be understood as the random variable that results from the output of an execution of protocol π, with an adversary $\mathcal{A}$, in the presence of environment $\mathcal{Z}$, on input $x \in$

$\{0,1\}^*$. Consequently, we define the probability ensemble as $\text{EXEC}_{\pi,A,\mathcal{Z}} = \text{EXEC}_{\pi,A,\mathcal{Z}}(x)_{x\in\{0,1\}^*}$ The corresponding ensemble and random variable definition for the ideal world is $\text{EXEC}_{\pi',\mathcal{S},\mathcal{Z}} = \text{EXEC}_{\pi,\mathcal{S},\mathcal{Z}}(x)_{x\in\{0,1\}^*}$ We provide some essential definitions below:

Definition 2. *Two probability distribution ensembles are considered to be indistinguishable from each other, (i.e X, Y), if for any $c, d \in \mathcal{N}, \exists k_0 \in \mathcal{N}$ s.t $\forall k > k_0$ and $\forall z \in \cup_{\kappa\leq k^d}\{0,1\}^\kappa$, we have that*

$$|\Pr(X(k,z)=1) - \Pr(Y(k,z)=1)| < k^{-c}.$$

Below, we define the meaning of a protocol UC-emulating a functionality. The definition below is for the computationally bounded setting; it is worth noting that a similar definition exists for statistical security.

Definition 3. *Let π and $\mathcal{F}$ be a PPT protocol and an ideal functionality respectively. If for any polytime adversary $\mathcal{A}$, we have a polytime simulator $\mathcal{S}$ such that for any polytime environment $\mathcal{Z}$, we have that the probability ensembles are indistinguishable from each other, (i.e that $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$), then we say that protocol π UC-emulates functionality $\mathcal{F}$.*

The notion of UC-emulation is necessary to describe how protocols can be composed. In general, we want to show that a protocol π, with some reliance on an ideal functionality $\mathcal{F}$, can provide the same security guarantees as π that runs with some real-world protocol ϕ, as long as ϕ UC-emulates $\mathcal{F}$.

Let ρ, π, ϕ be protocols, where ϕ is a subroutine protocol of ρ. When we write $\rho^{\phi\to\pi}$, we mean the protocol ρ, where every invocation of ϕ is replaced by calls to π. We present the simplified UC-composition theorem present in [9].

Definition 4. *Let ρ, ϕ, π be protocols where ψ is a subroutine of ρ, π UC-emulates ϕ, and π is identity compatible with ρ and ϕ. Then protocol $\rho^{\phi\to\pi}$ UC-emulates ρ.*

Dynamic participation. The UC framework contains mechanisms necessary to model protocols in which the number of participants/parties is not fixed during the execution of the model. To mediate the protocol's procession, a special control function is introduced. The control function is described as $C : \{0,1\}^* \to \{allow, disallow\}$. In UC, control functions are paired up with ITMs to form a system of ITMs. These system ITMs

are defined as $S = (I, C)$, where I refers to the initial ITM, and C, the control function.

A external-write instruction has parameters that can specify whether or not to create a new ITM process. Specifically, for a machine $M = (\mu, id)$, on an external-write command, we parse the contents of M's outgoing message tape as (f, M', t, r, M, m). Here, $f \in \{0, 1\}$ is interpreted as the forced-write flag, M' is the extended identity of the recipient ITM, $t \in \{\text{input}, \text{subroutine-output}, \text{backdoor}\}$, $r \in \{0, 1\}$ is the reveal-sender-id flag, M is the extended identity of the sending ITM, and $m \in \{0, 1\}^*$ is the message.

When this instruction is executed, the control function C first is queried on (f, M', t, r, M, m). If the output is *disallow*, then the message is not passed, and the initial ITM is instead activated. If $f = 1$, then we interpret the value M' as $M' = (\mu', id')$. If the output is *allow*, $f = 1$, and $M' = (\mu', id')$ is in the system, then the message m is inserted into the tape of M'. If the reveal-sender-id$r = 1$, then we also insert (μ, id) into M' tape. Execution is passed to M'.

If the output is *allow*, $f = 1$, and $M' = (\mu', id')$ is not in the system, then we spawn a new machine M' with code (μ', id'). The random tape is populated as it was for the initial ITM. Through this mechanism, we allow new machines to be created, which can correspond to a variety of events, such as a new participant joining the protocol. We say that M invoked M'.

Global setup. Sometimes, the mechanisms and protocols that we desire to model in the UC framework are shared by other protocols. Without careful review, the security guarantees of a UC proof may not hold. These are referred to as global subroutines. A prominent example appears in [11]: Consider the circumstance in which a UC non-interactive zero knowledge protocol (NIZK) π attempts to naively use a random oracle $H(\cdot)$. That same hash function can be used by anyone else in order to verify any resulting proof from the NIZK protocol, which contradicts the definition of the NIZK functionality. This is of course, problematic: it renders our ability to use certain primitives or models less effective.

Future work [1, 3, 10] poses and elaborates on the following question: *Under what circumstances can a protocol designer use a protocol available beyond the protocol under examination?*. One crucial component of the answer can be found in [1], which introduces the notion of a "subroutine" respecting requirement. Informally, a protocol π is considered to be subroutine respecting when for any session of π that occurs, four constraints are kept. First, no main party or sub-party passes input to

a machine not part of the session, they reject all incoming subroutine output from a machine not within the session, no sub-party will pass its subroutine output to another machine not in the session, and the sub-parties of the session will reject inputs passed from a machine that is not a main party or a subroutine of a main party. This acts as a restriction on how the protocol and subroutine interacts with machines outside of the protocol. It is shown in [1] that a subroutine respecting restriction is necessary to show UC emulation with global subroutines. The reader is invited to explore the aforementioned works for further detail.

B Property-based Treatment of Variable Difficulty (cont'd)

Previous work in the static, property-based analysis of Bitcoin [17,31] define what it means for a typical execution to occur by defining bounds on which random variables based on events in the execution must obey. [19] follows with the same approach.

Their work occurs in a similar model to our work. First, they model the hash functionality as a random oracle, identical to our approach. They operate in the partial synchronous model, in which a delay factor Δ is finite, but unknown to the parties, even during runtime. Parties are permitted access to a diffusion functionality that enables parties to broadcast messages.

Several important random variables are defined with respect to an execution. Firstly, D_r, which is to be understood as equal to the sum of the difficulties of all blocks that the honest party computes at a given round r. Correspondingly, for a set S of rounds, or set J, we define $D(S)$ to be the sum of the difficulties of all blocks the honest party computes for all rounds in S, that is $D(S) = \sum_{r \in S}(D_r)$. n_r, as in our work, represents the number of honest parties in the system. Q_r is the maximum difficulty among all blocks computed by the honest parties at a given round r if no other block was honestly mined during rounds $[r, r + \Delta]$.

At any given round, the probability that a single miner out of n parties queries successfully to the oracle at a target T is

$$f(T, n) = 1 - (1 - \frac{T}{2^\kappa})^n \leq \frac{Tn}{2^\kappa}$$

On inputs T_0, n_0, they refer to the value as $f(T_0, n_0) = f$. The authors then define the notion of a typical execution, the idea being that certain bounds can be defined on the random variables, and that these typical

executions occur with overwhelming probability. We highlight their definitions and resulting theorem toward this end.

Definition 5 ([19]). *An insertion occurs when given a chain $\mathcal{C}$ with two consecutive blocks B and B', a block B^* is created after B' is such that B, B^*, B' form three consecutive blocks of a valid chain. A copy occurs if the same block exists in two different positions. A prediction occurs when a block extends one with later creation time.*

Definition 6 (Typical Execution [19]). *An execution E is typical if the following hold.*

- *For any set S of at least ℓ consecutive good rounds,*

$$(1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta}(1/2^{\kappa})n(S) < Q(S) \leq D(S) < (1+\epsilon)1/2^{\kappa}$$

- *For any set J of consecutive adversarial queries and $\alpha(J) = \frac{2(\frac{1}{\epsilon}+\frac{1}{3})\lambda}{T(J)}$,*

$$A(J) < \frac{|J|}{2^{\kappa}} + \max(\frac{\epsilon|J|}{2^{\kappa}}, \tau\alpha(J))$$

$$B(J) < \frac{|J|}{2^{\kappa}} + \max(\frac{\epsilon|J|}{2^{\kappa}}, \alpha(J))$$

- *No insertions, no copies, and no predictions occurred in E*

Note that $n(S) = \sum_{r \in S} n_r$

Theorem 3 ([19]). *Assuming the ITM system $(\mathcal{Z}, C)$ runs for L steps, the probability of the event "$\mathcal{E}$ is not typical" is bounded by $O(L^2)(e^{-\lambda} + 2^{\kappa})$*

Their work then shows that because an execution is typical with overwhelming probability, the common prefix, chain quality and chain growth properties can be proven. From there, one can show that with these underlying blockchain properties, we can make guarantees about the behaviour of the application layer. These are most notably the ledger properties consistency and liveness. In our work, we apply these results to the UC setting to enforce that the ideal world specification is sufficiently restricted enough to be realized by the real world protocol.

Definition 7 ([18]). *The* common prefix *property Q_{cp} with parameter $k \in \mathbb{N}$ states that for any two parties P_1, P_2 holding chains $\mathcal{C}_1, \mathcal{C}_2$ at rounds r_1, r_2, with $r_1 \leq r_2$, it holds that $C_1^{\lceil k} \preceq C_2$*

At a high level, the common prefix property guarantees that if a party removes the most recent k blocks from his version of the blockchain, the resulting blockchain will be the prefix of any other (honest) party's blockchain. In other words, all participants will agree on the blocks more than k-blocks deep. This is instrumental to show that transactions on the blockchain will eventually be 'settled,' with all the participants agreeing on its inclusion.

Let $\text{VIEW}^{P}_{\Pi,\mathcal{A},\mathcal{Z}}$ represent a random variable ensemble that results from the view of a party P upon completion of an execution of protocol Π, with environment $\mathcal{Z}$ and adversary $\mathcal{A}$, with no auxiliary information passing (e.g, standalone execution.) We omit P from the expression to mean the concatenation of all party's views: $\text{VIEW}_{\Pi,\mathcal{A},\mathcal{Z}}$.

Definition 8 ([18]). *The* chain quality *property Q_{cp} with parameters $\mu \in \mathbb{R}$ and $\ell \in \mathbb{N}$, states that for any party P with chain $\mathcal{C}$ in $\text{VIEW}_{\Pi,\mathcal{A},\mathcal{Z}}$ and any segment of that chain of difficulty d such that the first block of the segment was computed at least ℓ rounds earlier than the last block, the blocks that the honest parties have contributed in the segment have total difficulty at least $\mu \cdot d$.*

At a high level, chain quality guarantees that no adversary will be able to contribute too many blocks of high difficulty to the blockchain in any large enough segment. This is important to prevent the adversary from obtaining too much control.

It is shown in [18] that common prefix and chain quality hold with overwhelming probability (in κ) during the execution of the abstraction of the Bitcoin protocol. To this end, they define the meaning of a 'typical' execution, and show that such executions imply the aforementioned properties will hold.

In the context of implementing a ledger, we are also interested in defining desirable traits at the application layer. The key distinction is that at the blockchain level, we are interested in the behaviour of the data structure, where as at the application layer we are interested in how the selected application behaves on top of the blockchain. Let $\mathcal{L}$ represent the ledger that contains the sequence of transactions that are settled, and $\hat{\mathcal{L}}$ represent the full ledger.

Definition 9 ([18]). *A ledger satisfies* Consistency*, if for any two honest parties P_1, P_2 reporting $\mathcal{L}_1, \mathcal{L}_2$ at rounds $r_1 \leq r_2$, it holds that $\mathcal{L}_1$ is a prefix of $\hat{\mathcal{L}}_2$.*

Definition 10 ([18]). *A ledger satisfies* Liveness *with a wait-time parameter $u \in \mathbb{N}$ when, if a transaction $\mathtt{tx}$ is provided to all honest parties for u consecutive rounds, it holds that for any player P, $\mathtt{tx}$ is in $\mathcal{L}$.*

Theorem 4 ([19]). *For a typical execution in a $(\gamma, \frac{(1+\delta)\gamma^2 m}{f})$-respecting environment, the common-prefix property holds for parameter (ϵm).*

Theorem 5 ([19]). *For a typical execution in a $(\gamma, \frac{(1+\delta)\gamma^2 m}{f})$-respecting environment, the chain quality property holds with parameters $\ell + 2\Delta$ and $\mu = \delta - 3\epsilon$*

Theorem 6 ([19]). *Suppose that at round u of an execution E an honest party broadcasts a chain of difficulty d. Then, by round v, every honest party has received a chain of difficulty at least $d + Q(S)$, where $S = \{r : u + \Delta \leq r \leq v - \Delta\}$*

Theorem 7 ([19]). *For a typical execution in a* $(\gamma, \frac{(1+\delta)\gamma^2 m}{f})$*-respecting environment, consistency is satisfied by setting the settled transactions to be those reported more than* ϵm *blocks deep.*

Theorem 8 ([19]). *For a typical execution in a* $(\gamma, \frac{(1+\delta)\gamma^2 m}{f})$*-respecting environment, Liveness is satisfied for depth* ϵm *with wait-time* $\frac{(4\gamma^2+1)\epsilon m}{f}$*.*

C Glossaries and Tables

Parameter	Meaning
ϵ	Quality of concentration of variables.
`windowSize`	Cutoff parameter for common prefix.
δ	Advantage of honest parties, such that $t_r \leq (1-\delta)h_r$ holds true for every round.
Δ	Maximum delay parameter for communication on the network. Note that the delay parameter is unknown to the parties.
m	Size of an epoch in blocks.
$\mathsf{b}_{\mathsf{tgt}}$	The intended average time between successful proof of works. In Bitcoin, this value is 10 minutes.
τ	Dampening factor for the target recalculation. $\tau = 4$ in Bitcoin.
γ, s	Values that describe the maximum fluctuation of participation with a given period of time. See definition 1.
ℓ	Size of the interval used in the analysis.
T_0	The initial target difficulty for the first epoch.
κ, λ	These are the security parameters. κ corresponds to the hash size; λ corresponds to the probability that desired properties do not hold.

Table 1. Glossary of important parameters.

D The Ideal Functionalities in Detail

D.1 The Random Oracle Functionality

We use a standard definition for a random oracle, with a κ security parameter. We adopt the functionality used by Baderscher et al. [4]. In short, the random oracle functionality will internally store a table of queried

values and their hash. If a value is queried that has not been queried previously, a random value is added to the table and returned.

Functionality $\mathcal{F}_{\mathrm{RO}}^{\kappa}$

Initialization :
The functionality initializes the party set $\mathcal{P} \leftarrow \emptyset$. It initializes a function table $H \leftarrow \emptyset$ (we write $H[x] = \perp$ to denote the fact that no assignment has been made)

Registration :

- Upon receiving $(\text{REGISTER}, sid)$ from some party $\mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $\mathcal{P}$), set $\mathcal{P} = \mathcal{P} \cup \{P\}$, and return $(\text{REGISTER}, sid, P)$ to the caller.
- Upon receiving $(\text{DE-REGISTER}, sid)$ from some party $P \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $P \in \mathcal{P}$), set $\mathcal{P} := \mathcal{P} \setminus \{P\}$ and return $(\text{DE-REGISTER}, sid, P)$ to the caller.
- Upon receiving $(\text{GET-REGISTERED}, sid)$ from $\mathcal{A}$, the functionality returns the response $(\text{GET-REGISTERED}, sid, \mathcal{P})$ to $\mathcal{A}$

RO queries :

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

Additionally, their work describes a wrapper for the random oracle that encapsulates the notion of q query bounded parties. If the adversary attempts to access the random oracle beyond its allowed amount, the wrapper ignores the query. Otherwise, it will forward the request to the random oracle.

Functionality $\mathcal{W}^{q}(\mathcal{F}_{\mathrm{RO}}^{\kappa})$

Initialization :
The functionality manages the variable $\mathsf{counter}$ (initially 0) and set the corrupted parties $\mathcal{P}'$ in the session. For each party $P \in \mathcal{P}'$ it manages variables count_P. Initially $\mathcal{P}' = \emptyset$ and $\mathsf{counter} = 0$

Registration :

- The wrapper does not interact with the adversary as soon as the adversary tries to exceed its budget of q queries per corrupted party. Registration-queries and their replies are simply relayed without modifications.

Relaying inputs to the random oracle :

– Upon receiving (EVAL, sid, x) from $\mathcal{A}$ on behalf of a corrupted party $P \in \mathcal{P}'$, then first execute **Round-Reset**. Then, set $\mathsf{count}_P \leftarrow \mathsf{count}_P + 1$ and only if $\mathsf{count}_P \leq q$, forward the request to $\mathcal{F}_{\text{RO}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{RO}}$ 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.

Standard UC Corruption Handling :
– Upon receiving (CORRUPT, sid, P) from the adversary, set $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{P\}$. If P has issued $t > 0$ random oracle queries in this round, set $\mathsf{count}_P \leftarrow t$. Otherwise set $\mathsf{count}_P \leftarrow 0$.

Procedure Round-Reset :

– Send (CLOCK-READ, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$ and receive (CLOCK-READ, cid, τ) from $\bar{\mathcal{G}}_{\text{CLOCK}}$. If $|\tau - \mathsf{counter}| > 0$, then set $\mathsf{count}_P \leftarrow 0$ for each participant $P \in \mathcal{P}'$ and set $\mathsf{counter} \leftarrow \tau$.

D.2 The Diffuse Functionality

$\mathcal{F}_{\text{DIFF}}$ is an ideal functionality for multicast, in which the participants are able to broadcast messages over the network. We use a similar approach to [4], except that we enforce the delay to be set by the adversary.

Functionality $\mathcal{F}_{\text{DIFF}}$

Initialization:
The functionality initializes the party set $\mathcal{P} \leftarrow \emptyset$ and a list (of messages) $\vec{M} \leftarrow []$. It also maintains a variable $\Delta \in \mathbb{N}$, initialized to 0.

Setting Delay:
Upon receiving (SET-DELAY, sid, n) from the adversary $\mathcal{A}$, if $n \in \mathbb{N}$ and SET-DELAY has never been received by this functionality, then set $\Delta = n$. Return (SET-DELAY, sid, ok)

Registration:
– Upon receiving (REGISTER, sid) from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$ and return (REGISTER, sid, P to the caller
– Upon receiving (DE-REGISTER, sid) from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$ and return (DE-REGISTER, sid, P) to the caller.

Network Capabilities:

– Upon receiving (MULTICAST, sid, m) from $P_s \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P_s if corrupted), where $\mathcal{P} = \{P_1, \ldots, P_n\}$ denotes the current party set, do:
  1. Choose n new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$,
  2. Initialize $2n$ new variables $D_{\mathsf{mid}_1} := D^{MAX}_{\mathsf{mid}_1} := \cdots := D_{\mathsf{mid}_n} := D^{MAX}_{\mathsf{mid}_n} := 1$
  3. Set $\vec{M} := \vec{M} \parallel (m, \mathsf{mid}_1, D_{\mathsf{mid}_1}, P_1) \parallel \cdots \parallel (m, \mathsf{mid}_n, D_{\mathsf{mid}_n}, P_n)$,
  4. Send (MULTICAST, $sid, m, P_s, (P_1, \mathsf{mid}_1), \ldots, (P_n, \mathsf{mid}_n)$) to the adversary.
– Upon receiving (FETCH, sid) from $P_i \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of P_s if corrupted):
  1. For all tuples $(m, \mathsf{mid}, D_{\mathsf{mid}}, P_i) \in \vec{M}$ set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.
  2. Let $\vec{M}_0^{P_i}$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \mathsf{mid}, D_{\mathsf{mid}}, P_i)$ with $D_{\mathsf{mid}} \leq 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}_0^{P_i}$ from $\vec{M}$.
  3. Output $\vec{M}_0^{P_i}$ to P_i (if P_i is corrupted, give $\vec{M}_0^{P_i}$ to $\mathcal{A}$).

Additional Adversarial Capabilities:
– Upon receiving (MULTICAST, $sid, (m_{i_1}, P_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell})$ from the adversary with $\{P_{1_\ell}, \ldots, P_{i_\ell}\} \subseteq \mathcal{P}$, do:
  1. Choose ℓ new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_\ell$,
  2. Initialize ℓ new variables $D_{\mathsf{mid}_{i_1}} := D^{MAX}_{\mathsf{mid}_{i_1}} := \cdots := D_{\mathsf{mid}_{i_\ell}} := D^{MAX}_{\mathsf{mid}_{i_\ell}} := 1$
  3. Set $\vec{M} := \vec{M} \parallel (m_{i_1}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}, P_{i_1}) \parallel \cdots \parallel (m_{i_\ell}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_\ell}}, P_{i_\ell})$,
  4. Send (MULTICAST, $sid, (m_{i_1}, P_{i_1}, \mathsf{mid}_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell}, \mathsf{mid}_{i_\ell})$ to the adversary.
– Upon receiving (DELAYS, $sid, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell})$ from the adversary do the following for each pair $(T_{\mathsf{mid}_{i_j}}, \mathsf{mid}_{i_j})$: If $D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}} \leq \Delta$ and mid is a message-ID registered in the current $\vec{M}$, set $D_{\mathsf{mid}_{i_j}} := D_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$ and set $D^{MAX}_{\mathsf{mid}_{i_j}} := D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}}$; otherwise, ignore this pair.
– Upon receiving (SWAP, $sid, \mathsf{mid}, \mathsf{mid}'$) from the adversary, if mid and mid' are message-IDs registered in hte current $\vec{M}$, then swap the triples $(m, \mathsf{mid}, D_{\mathsf{mid}}, \cdot)$ and $(m, \mathsf{mid}', D_{\mathsf{mid}'}, \cdot)$ in $\vec{M}$. Return (SWAP, sid) to the adversary.
– Upon receiving (GET-REGISTERED, sid) from $\mathcal{A}$, the functionality returns the response (GET-REGISTERED, $sid, \mathcal{P}$) to $\mathcal{A}$

D.3 The Static State Exchange Functionality

Functionality $\mathcal{F}_{\mathrm{STX}}(\mathcal{P}, \Delta, p_H, p_A)$

Initialization :
The functionality initializes a buffer $\vec{M}$ which contains successfully submitted states which have not yet been delivered to (some parties) in $\mathcal{P}$. It also manages a buffer $\mathbf{N}_{net}$ of adverbially injected chunk messages (that might not correspond to valid states).

Submit/receive new states :

– Upon receiving (SUBMIT-NEW, $sid, \vec{\mathtt{st}}, \mathtt{st}$) from some participant $P_s \in \mathcal{P}$, if $\mathsf{isvalidstate}_{\mathcal{B}}(\vec{\mathtt{st}} \parallel \mathtt{st}) = 1$ and $\vec{\mathtt{st}} \in \mathcal{T}_P$, then do the following:
  1. Sample B according to a Bernoulli-Distribution with parameter p_H (or p_A if P_s is dishonest).
  2. If $B = 1$, set $\vec{\mathtt{st}}_{new} \leftarrow \vec{\mathtt{st}} \parallel \mathtt{st}$ and add $\vec{\mathtt{st}}_{new}$ to $\mathcal{T}_{P_s}$. Else, set $\vec{\mathtt{st}}_{new} \leftarrow \vec{\mathtt{st}}$.
  3. Output (SUCCESS, sid, B) to P_s
  4. On response (CONTINUE, sid) where $\mathcal{P} = \{P_1, \dots P_n\}$ choose n new unique message-IDs $\mathsf{mid}_1, \dots, \mathsf{mid}_n$, initalize n new variables $D_{\mathsf{mid}_1} := D^{MAX}_{\mathsf{mid}_1} := \cdots := D_{\mathsf{mid}_n} := D^{MAX}_{\mathsf{mid}_n} := 1$, set $\vec{M} := \vec{M} \parallel (\vec{\mathtt{st}}_{new}, \mathsf{mid}_1, D_{\mathsf{mid}_1}, P_1) \parallel \cdots \parallel (\vec{\mathtt{st}}_{new}, \mathsf{mid}_n, D_{\mathsf{mid}_n}, P_n)$, and send (SUBMIT-NEW, $sid, \vec{\mathtt{st}}_{new}, P_s, (P_1, \mathsf{mid}_1), \dots, (P_n, \mathsf{mid}_n)$) to the adversary.
– Upon receiving (FETCH-NEW, sid) from a party $P \in \mathcal{P}$ or $\mathcal{A}$ (on behalf of P), do the following:
  1. For all tuples $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P) \in \vec{M}, \mathbf{N}_{net}$ set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$
  2. Let $\vec{M}_0^P$ denote the subvector of $\vec{M}$ including all tuples of the form $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P)$ where $D_{\mathsf{mid}} \leq 0$ (in the same order as they appear in $\vec{M}$. For each tuple $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P) \in \vec{M}_0^P$ add $\vec{\mathtt{st}}$ to $\mathcal{T}_P$. Delete all entries in $\vec{M}_0^P$ from $\vec{M}$ and send $\vec{M}_0^P$ to P. If P is corrupted, provide additionally $\mathbf{N}_{net}$ to the adversary.

Further adversarial influence on the network :

– Upon receiving (SEND, $sid, \vec{\mathtt{st}}, P'$) from $\mathcal{A}$ on behalf of some corrupted $P \in \mathcal{P}$, if $P' \in \mathcal{P}$ and $\vec{\mathtt{st}} \in \mathcal{T}_{P_s}$, choose a new unique message-ID mid, initialize $D := 1$, add $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P')$ to $\vec{M}$, and return (SEND, $sid, \vec{\mathtt{st}}, P', \mathsf{mid}$) to $\mathcal{A}$. If $\vec{\mathtt{st}} \notin \mathcal{T}$, then conduct the same steps except that $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P')$ is added to $\mathbf{N}_{net}$
– Upon receiving (SWAP, $sid, \mathsf{mid}, \mathsf{mid}'$) from $\mathcal{A}$, if mid and mid' are message-IDs registered in the current $\vec{M}$, swap the corresponding tuples in $\vec{M}$. Return (SWAP, sid) to $\mathcal{A}$.
– Upon receiving (DELAY, sid, T, mid) from $\mathcal{A}$, if T is a valid delay, mid is a message-ID for a tuple $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P)$ in the current $\vec{M}$ and $D^{MAX}_{\mathsf{mid}} + T \leq \Delta$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} + T$ and set $D^{MAX}_{\mathsf{mid}} = D^{MAX}_{\mathsf{mid}} + T$
– Upon receiving (GET-REGISTERED, sid) from $\mathcal{A}$ the functionality returns the response (GET-REGISTERED, sid, P) to $\mathcal{A}$

D.4 The Static Ledger Functionality from [4]

Functionality $\bar{\mathcal{G}}_{\text{LEDGER}}$

Party Management:

– Upon receiving (REGISTER, sid) from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$, set $C[\tau_L] = |\mathcal{P}|$, initialize $\mathtt{pt}_P \leftarrow 1, \mathsf{state}_P \leftarrow \epsilon$, and $\tau_P^{reg} \leftarrow \tau_L$. If P is an honest party and if $\mathcal{H} = \emptyset$ send (REGISTER, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$. If P is honest then update $\vec{\mathcal{I}}_H^T$ and set $\mathcal{H} \leftarrow \mathcal{H} \cup \{P\}$ and if additionally $\tau_P^{reg} > 0$ holds, set $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \cup \{P\}$. Return (REGISTER, sid, P) to the caller.

– Upon receiving (DE-REGISTER, sid) from some party $P \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted $P \in \mathcal{P}$), set $\mathcal{P} \leftarrow \mathcal{P} \setminus \{P\}$, $\mathcal{H} \leftarrow \mathcal{H} \setminus \{P\}$, $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \setminus \{P\}$, and set $C[\tau_L] = |\mathcal{P}|$. If $\mathcal{H} = \emptyset$, send (DE-REGISTER, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$. If P is honest then update $\vec{\mathcal{I}}_H^T$. Return (DE-REGISTER, sid, P) to the caller.

Upon receiving forwarded input from the wrapper, do the following:

Honest Party Operations:
If $P_i \in \mathcal{H}$ then additionally take the following steps:

1. $(\vec{N}, s') \leftarrow \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \mathtt{NxtBC}, \mathsf{buffer}; s_{ep})$. Reset $\mathtt{NxtBC} \leftarrow \epsilon$ and store $s_{ep} \leftarrow s'$.
2. If $\vec{N} \neq \epsilon$ then parse $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell)$ and update $\mathsf{state} \leftarrow \mathsf{state} \parallel \mathsf{Blockify}(\vec{N}_1) \parallel \cdots \parallel \mathsf{Blockify}(\vec{N}_\ell)$.
3. For each $\mathtt{BTX} \in \mathsf{buffer}$: if $\mathsf{Validate}(\mathtt{BTX}, \mathsf{state}, \mathsf{buffer}) = 0$ then $\mathsf{buffer} \leftarrow \mathsf{buffer} \setminus \{\mathtt{BTX}\}$.
4. If $\exists P \in \mathcal{H} \setminus \mathcal{P}_{DS}$ s.t $\mathsf{pt}_P \notin [|\mathsf{state}| - \mathtt{windowSize} + 1, |\mathsf{state}|]$, then set $\mathsf{pt}_{P_k} \leftarrow |\mathsf{state}|$ for all $P_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.

General Party Operations:
If the input I is a ledger instruction from a party $P_i \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted party $P_i \in \mathcal{P}$), execute the respective code:

- *Submitting a transaction:* If $I =$ (SUBMIT, $sid, \mathtt{tx}$) do the following:
  1. Choose a unique transaction ID $\mathtt{txid}$ and set $\mathtt{BTX} \leftarrow (\mathtt{tx}, \mathtt{txid}, \tau_L, P_i)$
  2. if $\mathsf{Validate}(\mathtt{BTX}, \mathsf{state}, \mathsf{buffer}) = 1$, then $\mathsf{buffer} \leftarrow \mathsf{buffer} \cup \{\mathtt{BTX}\}$.
  3. Output (SUBMIT, $\mathtt{BTX}$) to $\mathcal{A}$
- *Reading the state:* If $I =$ (READ, sid) then do the following: if $P_i \in \mathcal{H} \setminus \mathcal{P}_{DS}$ then set $\mathsf{state}_i := \mathsf{state}|_{min\{pt_i, |\mathsf{state}|\}}$. Return (READ, sid, state_i) to the caller.
- *Maintaining the ledger state:* If $I =$ (MAINTAIN-LEDGER, $sid, minerID$) and $P_i \in \mathcal{H}$ and $\mathtt{predict\text{-}time}(\vec{\mathcal{I}}_H^T) > \tau_P^{reg}$ then send (CLOCK-UPDATE, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.

If the input I is an additional adversarial capability (received on the backdoor tape from $\mathcal{A}$) execute the respective code:

- *The adversary reading the state:* If $I =$ (READ, sid), then return $(\mathsf{state}, \mathsf{buffer}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.
- *The adversarial proposing the next block:* If $I =$ (NEXT-BLOCK, $(\mathtt{txid}_1, \ldots, \mathtt{txid}_\ell)$), update $\mathtt{NxtBC}$ as follows:
  1. Set listOfTxid $\leftarrow \epsilon$
  2. For $i = 1, \ldots, \ell$ do: if there exists a $\mathtt{BTX} = (\mathtt{tx}, \mathtt{txid}, minerID, \tau_L, P_i) \in \mathsf{buffer}$ with ID $\mathtt{txid} = \mathtt{txid}_i$ then set listOfTxid := listOfTxid$\parallel \mathtt{txid}_i$.
  3. Finally set $\mathtt{NxtBC} :=$ $\mathtt{NxtBC}$ $\parallel$listOfTxid and output (NEXT-BLOCK, ok) to $\mathcal{A}$

- *The adversary setting state-slackness:* If $I = (\text{SET-SLACK}, (P_{i_1}, \hat{pt}_{i_1}), \ldots, (P_{i_\ell}, \hat{pt}_{i_\ell}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{H}$ then do the following: If for all $P_{i_j} \in \mathcal{H} \setminus \mathcal{P}_{DS}, j \in [\ell] : |\mathsf{state}| - \hat{pt}_{i_j} < \texttt{windowSize}$ and $\hat{pt}_{i_j} \geq |\mathsf{state}_{i_j}|$, then update $pt_{i_1} := \hat{pt}_{i_1}$ for every $j \in |\ell|$. Return $(\text{SET-SLACK}, ok)$ to $\mathcal{A}$
- *The adversary setting the state for desyncronized parties:* If $I = (\text{DESYNC-STATE}, (P_{i_1}, \mathsf{state}'_{i_1}), \ldots, (P_{i_\ell}, \mathsf{state}'_{i_1}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ then set $\mathsf{state}_{i_j} := \mathsf{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, ok)$ to $\mathcal{A}$.
- *The adversary obtaining the set of registered parties* If $I = (\text{GET-REGISTERED}, sid)$, then return $(\text{GET-REGISTERED}, sid, \mathcal{P})$ to $\mathcal{A}$.
- *The adversary corrupting a party*: If $I = (\text{CORRUPT}, sid, P_i)$ and $\texttt{predict-time}(\vec{I}_H^T) > \tau_L$ then send $(\text{CLOCK-UPDATE}, cid)$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$. Else return I to $\mathcal{A}$

E Protocols and Algorithms in Detail

E.1 The StateExchange protocol

Protocol $\mathsf{StateExchange}^{T_0, m, \mathsf{b_{tgt}}, q, \tau}(P)$

Initialization:
The protocol maintains a tree $\mathcal{T}_P$ of all valid chains the party sees. Initially it contains the genesis chain ($\mathbf{G}$).

Registration/De-registration:

- Upon receiving $(\text{REGISTER}, sid)$ do the following: send $(\text{REGISTER}, sid)$ to $\mathcal{F}_{\text{DIFF}}^{\mathsf{bc}}, \bar{\mathcal{G}}_{\text{CLOCK}}$ and $\mathcal{W}^q(\mathcal{F}_{\text{RO}}^\kappa)$ and output $(\text{REGISTER}, sid, P)$
- Upon receiving $(\text{DE-REGISTER}, sid)$, send $(\text{DE-REGISTER}, sid)$ to $\mathcal{F}_{\text{DIFF}}^{\mathsf{bc}}, \bar{\mathcal{G}}_{\text{CLOCK}}$ and $\mathcal{W}^q(\mathcal{F}_{\text{RO}}^\kappa)$. Set all variables back to their initial values and return $(\text{DE-REGISTER}, sid, P)$

Exchange: Exchange queries are only answered once registered.

- Upon receiving $(\text{SUBMIT-NEW}, sid, \vec{\mathtt{st}}, \mathtt{st})$ do
  - **if** $\mathsf{isvalidstate}_{\mathbb{B}}(\vec{\mathtt{st}} \parallel \mathtt{st}) = 1$ **then**
    - **if** there exists $\mathcal{C} \in \mathcal{T}$ with $\vec{\mathtt{st}}$ **then**
      - Send $(\text{CLOCK-READ}, cid)$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$, receive $(\text{CLOCK-READ}, cid, \tau_L)$
      - $\mathcal{C}_{new} \leftarrow \mathsf{extendchain}^{T_0, m, \mathsf{b_{tgt}}}(\mathcal{C}, \mathtt{st}, q, \tau)$
      - Compute $T \leftarrow \mathsf{gettarget}^{m, T_0, \tau, \mathsf{b_{tgt}}}(\vec{\mathtt{st}})$
      - **if** $\mathcal{C}_{new} \neq \mathcal{C}$ **then**
        - Update the local tree, i.e add $\mathcal{C}_{new}$ to $\mathcal{T}$
        - Output $(\text{SUCCESS}, sid, 1, T, \tau_L)$
      - **else**
        - Output $(\text{SUCCESS}, sid, 0, T, \tau_L)$
      - **end if**
      - Send $(\text{MULTICAST}, sid, \mathcal{C}_{new})$ to $\mathcal{F}_{\text{DIFF}}^{\mathsf{bc}}$

end if
end if

– Upon receiving (FETCH-NEW, sid) do the following:

Send (FETCH, sid) to $\mathcal{F}_{\text{DIFF}}^{\mathsf{bc}}$ and denote the response by (FETCH, sid, b)
Extract all valid chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from b and add them to $\mathcal{T}$
Extract states $\vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k$ from $\mathcal{C}_1, \ldots, \mathcal{C}_k$ and output them

E.2 The ModularLedger Protocol

Protocol $\mathsf{ModularLedger}^{T_0, m, \mathtt{b_{tgt}}, q, \tau, \mathtt{windowSize}}(P)$

Variables and Initial Values:

– The protocol stores a local (working) state $\mathcal{C}_{loc}$ and associated timestamp and target vectors $\vec{\tau_{loc}}, \vec{T_{loc}}$ respectively, which initially contains the genesis state.
– It additionally manages a separate chain $\mathcal{C}_{exp}$ and associated timestamp and target vectors $\vec{\tau_{exp}}, \vec{T_{exp}}$ respectively to store the current chain whose encoded state $\vec{\mathtt{st}}$ is exported as the ledger state (initially this chain contains the genesis block).
– Variable isInit stores the initialization status. Initially this variable is false.
– buffer contains the list of transactions obtained from the network. Initially this buffer is empty.
– A flag WELCOME to indicate whether an indication was received that a new party joined the network (initially WELCOME $= 0$).
– Two variables $\mathsf{doneWork}$ and $\mathsf{doneUpdate}$ (initialized to false) that indicate whether the respective round actions have been executed.
– The party stores its registration status to the hybrid functionalities internally. We do not introduce an explicit name for this variable.

Registration/De-Registration

– Upon receiving (REGISTER, sid) this party sends (REGISTER, sid) to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{STX}}(\mathcal{P}, \Delta, p_H, p_A))$. If this party receives (REGISTER, sid, P) then this party sends (REGISTER, sid) to $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}, \bar{\mathcal{G}}_{\text{CLOCK}}$ and outputs (REGISTER, sid, P). If the party receives (REGISTER, $sid, \top$), then return (REGISTER, $sid, \top$)
– Upon receiving (DE-REGISTER, sid), send (DE-REGISTER, sid) to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{STX}}(\mathcal{P}, \Delta, p_H, p_A))$ If this party receives (DE-REGISTER, sid, P), then send (DE-REGISTER, sid) to $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}$ and $\bar{\mathcal{G}}_{\text{CLOCK}}$. Set all variables back to their initial values and return (DE-REGISTER, sid, P). However, if the party receives (DE-REGISTER, $sid, \top$) (from $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{STX}})$), then return (DE-REGISTER, $sid, \top$)

Ledger Queries

Ledger queries are only answered once registered.

– Upon receiving (SUBMIT, sid, tx), set $\mathsf{buffer} \leftarrow \mathsf{buffer} \,\|\, \mathsf{tx}$, and send (MULTICAST, sid, tx) to $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}$.
– Upon receiving (READ, sid) send (CLOCK-READ, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$, receive as answer (CLOCK-READ, cid, τ) and proceed as follows:

if τ corresponds to an update mini-round and isInit and $\neg\mathsf{doneUpdate}$ **then**

Execute sub-protocol **FetchInformation** and set $\mathsf{doneUpdate} \leftarrow \mathsf{true}$.
end if
Let $\vec{\mathtt{st}}$ be the encoded state in $\mathcal{C}_{exp}$
Return $(\text{READ}, sid, \vec{\mathtt{st}}^{\mathtt{windowSize}})$

– Upon receiving $(\text{MAINTAIN-LEDGER}, sid, minerID)$ execute in a $(\text{MAINTAIN-LEDGER}, sid, minerID)$-interruptible manner the following:
1. If $\mathsf{isInit} = \mathsf{false}$, then set all variables to their initial values, set $\mathsf{isInit} \leftarrow \mathsf{true}$ and output $(\text{MULTICAST}, sid, \text{NEW-PARTY})$ to $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}$
2. Execute sub-protocol **Ledger-Maintenance**

Sub-Protocol FetchInformation

Send $(\text{FETCH-NEW}, sid)$ to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\text{STX}})$
Denote the response from $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\text{STX}})$ by
$(\text{FETCH-NEW}, sid, (\vec{\mathtt{st}}_1, T_1, \tau_1), \ldots, (\vec{\mathtt{st}}_k, T_k, \tau_k))$
Find the largest state $\vec{\mathtt{st}}_i$ from among $(\vec{\mathtt{st}}_{loc}, \vec{\mathtt{st}}_{exp}, \vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k)$ (ordering resolves ties) and do the following:
1. $(\vec{\mathtt{st}}_{loc}, T_{loc}, \tau_{loc}) \leftarrow (\vec{\mathtt{st}}_i, T_i, \tau_i)$
2. $(\vec{\mathtt{st}}_{exp}, T_{exp}, \tau_{exp}) \leftarrow (\vec{\mathtt{st}}_i, T_i, \tau_i)$

Send (FETCH, sid) to $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}$; denote the response from $\mathcal{F}_{\text{DIFF}}^{\mathsf{tx}}$ as (FETCH, sid, b)
Extract all received transactions $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$
Set $\mathsf{buffer} \leftarrow \mathsf{buffer} \parallel (\mathtt{tx}_1, \ldots, tx_k)$ from b
If a NEW-PARTY message was received, set $\text{WELCOME} \leftarrow 1$. Otherwise, set $\text{WELCOME} \leftarrow 0$.
Remove all transactions from buffer which are invalid with respect to $\vec{\mathtt{st}}_{loc}^{\mathtt{windowSize}}$

Sub-Protocol LedgerMaintenance

This sub-protocol is executed in a $(\text{MAINTAIN-LEDGER}, sid, minerID)$*-interruptible manner*

1. Send $(\text{CLOCK-READ}, cid)$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$ receive as answer $(\text{CLOCK-READ}, cid, \tau)$, and proceed according to the following case distinction.
2. If τ corresponds to a working mini-round

if $\neg\mathsf{doneWork}$ **then**
Let $\vec{\mathtt{st}}$ be the encoded state in $\mathcal{C}_{loc}$
Set $\mathsf{buffer}' \leftarrow \mathsf{buffer}$
Parse buffer as sequence $(\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$
Set $\vec{N} \leftarrow \mathtt{tx}_{minerID}^{coin-base}$
Set $\mathtt{st} \leftarrow \mathsf{Blockify}_{\bar{\mathrm{B}}}(\vec{N})$
repeat
Let $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ be the current list of (remaining) transactions in buffer'
for $i = 1$ to n **do**
if $\mathsf{ValidTx}_{\bar{\mathrm{B}}}(\mathtt{tx}_i, \vec{\mathtt{st}} \parallel \mathtt{st}) = 1$ **then**
$\vec{N} \leftarrow \vec{N} \parallel \mathtt{tx}_i$
Remove $\mathtt{tx}$ from buffer
Set $\mathtt{st} \leftarrow \mathsf{Blockify}_{\bar{\mathrm{B}}}(\vec{N})$
end if
end for

until $\vec{N}$ does not increase anymore
Execute **ExtendState(st)**
If the flag WELCOME $= 1$, send (MULTICAST, sid, buffer) to $\mathcal{F}_{\text{DIFF}}^{\text{tx}}$.
Otherwise, give up activation.
end if
Set doneWork $\leftarrow$ true, doneUpdate $\leftarrow$ false and send (CLOCK-UPDATE, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$

3. If τ corresponds to an update mini-round
if $\neg$doneUpdate **then**
Execute **FetchInformation**
end if
Set doneUpdate $\leftarrow$ true, doneWork $\leftarrow$ false, and send (CLOCK-UPDATE, cid) to $\bar{\mathcal{G}}_{\text{CLOCK}}$

Sub-Protocol ExtendState(st)
Send (SUBMIT-NEW, $sid, \vec{\mathtt{st}}_{loc}, \mathtt{st}$) to $\mathcal{W}^{\text{stx-params}}(\mathcal{F}_{\text{STX}})$
Denote the response from $\mathcal{W}^{\text{stx-params}}(\mathcal{F}_{\text{STX}})$ by (SUCCESS, sid, B, T, τ)
if $B = 1$ **then**
Update the local state, i.e $(\vec{\mathtt{st}}_{loc}, \vec{T_{loc}}, \vec{\tau_{loc}}) \leftarrow (\vec{\mathtt{st}}_{loc}, \vec{T_{loc}}, \vec{\tau_{loc}}) \parallel (\mathtt{st}, T, \tau)$
end if

There are some additional details regarding the protocol which warrant further discussion. First $\vec{\mathcal{I}}_H^T$, which is used extensively by $\bar{\mathcal{G}}_{\text{LEDGER}}$ and its wrapper. It is understood to be the timed honest input sequence, containing all of the messages given to a main ITI by the environment or adversary. It consists of 3-tuples (x_i, id_i, τ_i), where x_i is the i-th input sent to id_i, at timestamp τ_i (derived from the global clock).

Definition 11 ([19]). *A $\bar{\mathcal{G}}_{\text{CLOCK}}$-hybrid protocol has a* predictable synchronization pattern *iff there exist an efficiently computable algorithm* `predict-time`$(\cdot)$ *such that for any possible execution of Π in a session* sid *(i.e for any adversary and environment, and any choice of random coins) the following holds: If $\vec{\mathcal{I}}_H^T = ((x_1, id_1, \tau_1), \ldots, (x_m, id_m, \tau_m))$ is the corresponding timed honest-input sequence for this session, then for any $i \in [m-1]$:*

$$\texttt{predict-time}((x_1, id_1, \tau_1), \ldots, (x_i, id_i, \tau_i)) = \tau_{i+1}.$$

Armed with an algorithm `predict-time` that complies with the above constraint, we can enforce the progression of time in the ideal functionality. Without, parties would need to manage it alone, which can complicate analysis. See [4] for more details.

E.3 Algorithm validStruct

Below is the definition of the validStruct algorithm, which describes how syntactic correctness can be programmatically verified. It is similar to syntactic correctness in the static difficulty case, with a couple of key differences. First, timestamps must be in order, that is that a block b_i that comes before $b_{i'}$ cannot have a timestamp that is greater than the timestamp of $b_{i'}$. Finally, the difficulty must be calculated per block based on information provided in the previous blocks.

Algorithm $\mathsf{validStruct}_{\mathbb{B}}^{m,T_0,\tau,\mathsf{b_{tgt}}}(\mathcal{C})$

$res \leftarrow true$
$T \leftarrow \mathsf{gettarget}(\vec{\mathtt{st}}, m, T_0, \tau, \mathsf{b_{tgt}})$
if $(\mathsf{length}(\mathcal{C}) = 0)$ or $(H[\mathsf{head}]) \geq T$ **then**
  $res \leftarrow false$
else if $(\mathsf{length}(\mathcal{C}) = 1)$ **then**
  $res \leftarrow (\mathcal{C} = \mathbf{G})$
else
  $\mathcal{C}' \leftarrow \mathcal{C}$
  $\langle s', \mathtt{st}, \mathsf{n}', T', t' \rangle \leftarrow \mathsf{head}(\mathcal{C}')$
  repeat
    $\mathcal{C}' \leftarrow \mathcal{C}'^{\lceil 1}$
    $\mathbf{B} := \langle \mathsf{s}, \mathtt{st}, \mathsf{n}, T, t \rangle \leftarrow \mathsf{head}(\mathcal{C}')$
    $\mathsf{con1} \leftarrow (H[\mathbf{B}] \neq s')$
    $\mathsf{con2} \leftarrow (\mathsf{length}(\mathcal{C}') > 1$ and $H[\mathbf{B}] \geq T)$
    $\mathsf{con3} \leftarrow (\mathsf{length}(\mathcal{C}') = 1$ and $\mathbf{B} \neq \mathbf{G})$
    $\mathsf{con4} \leftarrow (T \neq \mathsf{gettarget}(\mathcal{C}', m, T_0, \tau, \mathsf{b_{tgt}}))$
    $\mathsf{con5} \leftarrow (t' < t)$
    if con1 or con2 or con3 or con4 or con5 **then**
      $res \leftarrow false$
    else
      $\langle s', \mathtt{st}', \mathsf{n}', T', t' \rangle \leftarrow \langle s, \mathtt{st}, \mathsf{n}, T, t \rangle$
    end if
  until $res = false$ or $\mathsf{length}(\mathcal{C}') = 1$
end if
return res

E.4 Algorithm isvalidstate

Below is the definition of isvalidstate. We use it as described in [4] with no modifications.

Algorithm isvalidstate($\vec{\mathtt{st}}$)

Let $\vec{\mathtt{st}} := \mathtt{st}_1 \parallel \ldots, \parallel \mathtt{st}_n$
for each $\mathtt{st}_i$ **do**
  Extract the transaction sequence $\vec{\mathtt{tx}}_i \leftarrow \mathtt{tx}_{i,1}, \ldots, \mathtt{tx}_{i,n_i}$ contained in $\mathtt{st}_i$
end for
$\vec{\mathtt{st}}' \leftarrow \mathsf{gen}$
for $i = 1$ to n **do**
  if the first transaction in $\vec{\mathtt{tx}}_i$ is not a coin-base transaction **return** false
  $\vec{N} \leftarrow \mathtt{tx}_{i,1}$
  for $j = 2$ to $|\vec{tx}_i|$ **do**
    $\mathtt{st} \leftarrow \mathsf{Blockify}_{\ddot{\mathbb{B}}}(\vec{N})$
    if $\mathsf{ValidTx}_{\ddot{\mathbb{B}}}(\mathtt{tx}_{i,j}, \vec{\mathtt{st}}' \parallel \mathtt{st})$ **return** false
    $\vec{N}_i \leftarrow \vec{N}_i \parallel \mathtt{tx}_{i,j}$
  end for
  $\vec{\mathtt{st}}' \leftarrow \vec{\mathtt{st}}' \parallel \mathtt{st}_i$
end for
return true

E.5 Algorithm maxvalid

Algorithm $\mathsf{maxvalid}^{m,T_0,\tau,\mathsf{b_{tgt}}}_{\ddot{\mathbb{B}}}(\mathcal{C}_1, \ldots, \mathcal{C}_k)$

$\mathcal{C}_{temp} \leftarrow \epsilon, D_{max} \leftarrow 0$
for $i = 1$ to k **do**
  $D \leftarrow 0$
  Parse $\vec{\mathtt{st}}$ from $\mathcal{C}_i$
  if $\mathsf{validStruct}^{m,T_0,\tau,\mathsf{b_{tgt}}}_{\ddot{\mathbb{B}}}(\mathcal{C}_i) \wedge \mathsf{isvalidstate}(\vec{\mathtt{st}})$ **then**
    Parse $\mathcal{C}_i$ as $C = \mathbf{B}_1, \ldots, \mathbf{B}_n$
    for $j = 0$ to n **do**
      $D \leftarrow D + 1/T_j$
    end for
    if $D > D_{max}$ **then**
      $D_{max} \leftarrow D$, $C_{temp} \leftarrow \mathcal{C}_i$
    end if
  end if
end for
return $\mathcal{C}_{temp}$

E.6 Algorithms for Extending Policies for the Ledger

Algorithm $\mathsf{DefaultExtension}(\vec{\mathcal{I}}^T_H, \mathsf{state}, \mathsf{buffer}, \mathtt{NxtBC}, s_{ep})$

Let τ_L be the current ledger time (computed from $\vec{\mathcal{I}}^T_H$)
Read $\vec{\tau}_{\mathtt{state}}$ and $\vec{\mathtt{hf}}$ from the passed state s_{ep}

$\vec{N}_{\mathsf{df}} \leftarrow \epsilon$
Set $\vec{N}_0 \leftarrow \mathtt{tx}^{\mathsf{coinbase}}_{\mathsf{minerID}}$ of an honest miner
Sort buffer according to the time stamps and let $\vec{\mathtt{tx}} = (\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$ be the transactions in buffer
$\mathtt{st} \leftarrow \mathsf{Blockify}_{\mathbb{B}}(\vec{N}_0)$
repeat
  Let $\vec{\mathtt{tx}} = (\mathtt{tx}_1, \ldots, \mathtt{tx}_n)$ be the current list of (remaining) transactions
  for $i = 1$ to n **do**
    if $\mathsf{ValidTx}_{\mathbb{B}}(\mathtt{tx}_i, \mathsf{state} \parallel \mathtt{st}) = 1$ **then**
      $\vec{N}_0 \leftarrow \vec{N}_0 \parallel \mathtt{tx}_i$
      Remove $\mathtt{tx}_i$ from $\vec{\mathtt{tx}}$
      Set $\mathtt{st} \leftarrow \mathsf{Blockify}(\vec{N}_0)$
    end if
  end for
until $\vec{N}_0$ does not increase anymore
$c \leftarrow 0$
if $|\mathsf{state}| < \mathtt{windowSize} - 1$ **then**
  while $|\mathsf{state}| + c < \mathtt{windowSize} - 1$ **do**
    if $c > 0$ **then**
      Set $\vec{N}_c \leftarrow \mathtt{tx}^{\mathsf{coinbase}}_{\mathsf{minerID}}$ of an honest miner
    end if
    $\vec{N}_{\mathsf{df}} \leftarrow \vec{N}_{\mathsf{df}} \parallel \vec{N}_{\mathsf{c}}$
    $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_L$
    $c \leftarrow c + 1$
  end while
end if
$\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_L$
return $\vec{N}_{\mathsf{df}}$

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

$\vec{N}_{df} \leftarrow \mathsf{DefaultExtension}(\vec{\mathcal{I}}^T_H, \mathsf{state}, \mathsf{buffer}, \mathtt{NxtBC}, s_{ep})$
Let τ_L be the current ledger time (computed from $\vec{\mathcal{I}}^T_H$)
Read $\vec{\tau}_{\mathtt{state}}$ and $\vec{\mathtt{hf}}$ from the passed state s_{ep}. If the state is empty, initialize two empty vectors.
Parse $\mathtt{NxtBC}$ as a vector $((hFlag_1, \mathtt{NxtBC}_1), \cdots, (hFlag_n, \mathtt{NxtBC}_n))$
$\mathsf{target} \leftarrow \mathsf{gettarget}^{m,T_0,\tau,\mathsf{b}_{\mathsf{tgt}}}(\mathsf{state})$
$\vec{N} \leftarrow \epsilon$
if $|\mathsf{state}| \geq \mathtt{windowSize}$ **then**
  $\tau_{low} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathsf{state}| - \mathtt{windowSize} + 1$
else
  $\vec{\tau}_{\mathtt{state}} \leftarrow 0$
end if
for each list $\mathtt{NxtBC}_i$ of transaction IDs **do**

$\vec{N}_i \leftarrow \epsilon$
Use the exid contained in $\vec{N}_i$ to determine the list of transactions
Let $\vec{\mathtt{tx}} = (\mathtt{tx}_1, \ldots, \mathtt{tx}_{|\vec{N}_i|})$ denote the transactions of $\vec{N}_i$
if $\mathtt{tx}_1$ is not a coin-base transaction **then**
$\mathsf{inv} \leftarrow 1$, goto [terminate]
else
$\vec{N}_i \leftarrow \mathtt{tx}_i$
for $j = 2$ to $|\mathtt{NxtBC}_i|$ **do**
$\mathtt{st}_i \leftarrow \mathsf{Blockify}_{\mathcal{B}}(\vec{N}_i)$
if $\mathsf{ValidTx}_{\mathcal{B}}(\mathtt{tx}_j, \mathsf{state} \parallel \vec{\mathtt{st}}_i) = 0$ **then**
$\mathsf{inv} \leftarrow 1$, goto [terminate]
end if
$\vec{N}_i \leftarrow \vec{N}_i \parallel \mathtt{tx}_j$
end for
$\mathtt{st}_i \leftarrow \mathsf{Blockify}_{\mathcal{B}}(\vec{N}_i)$
end if
$hFlag \leftarrow 1$
for each $\mathtt{BTX} = (\mathtt{tx}, txid, \tau', P_i) \in \mathsf{buffer}$ of an honest party P_i with time $\tau' < \tau - \frac{\mathtt{Delay}}{2}$ **do**
if $\mathsf{ValidTx}_{\mathcal{B}}(\mathtt{tx}_j, \mathsf{state} \parallel \vec{\mathtt{st}}_i) = 1$ but $\mathtt{tx} \notin \vec{N}_i$ **then**
$hFlag \leftarrow 0$
end if
end for
$\vec{N} \leftarrow \vec{N} \parallel \vec{N}_i$
$\vec{T} \leftarrow \vec{T} \parallel \mathsf{target}$
$\mathsf{state} \leftarrow \mathsf{state} \parallel \mathtt{st}_i$
$\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_L, \vec{\mathtt{hf}} \leftarrow \vec{\mathtt{hf}} \parallel hFlag_i$ and store those vectors in s_{ep}.
if $|\mathsf{state}| \geq \mathtt{windowSize}$ **then**
$\tau_{low} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathsf{state}| - \mathtt{windowSize} + 1]$
else
$\vec{\tau}_{\mathtt{state}} \leftarrow 0$
end if
end for
if $\tau_L < \mathtt{maxTime}_{\text{window}} \wedge \mathsf{state} = \epsilon$ **then**
return ϵ
end if
$\mathsf{inv} \leftarrow 0$
if $|\mathsf{state}| < \mathtt{windowSize}$ **then** ▷ Ensure a timely startup with enough honest blocks
Send $(\text{RESOURCE-CHECK}, sid, \text{LEDGER-STARTUP}, (\vec{\tau}_{\mathtt{state}}, \vec{\mathtt{hf}}, \vec{T}))$ to $W(\bar{\mathcal{G}}_{\text{LEDGER}})$
end if
if $|\mathsf{state}| \geq \mathtt{windowSize}$ **then** ▷ Ensure ledger growth limits and enough blocks
Send $(\text{RESOURCE-CHECK}, sid, \text{LEDGER-GROWTH}, (\vec{\tau}_{\mathtt{state}}, \vec{\mathtt{hf}}, \vec{T}))$ to $W(\bar{\mathcal{G}}_{\text{LEDGER}})$
end if
[terminate]:
if $\mathsf{inv} = 0$ **then**
return $\vec{N}$ and new state s_{ep}

else
$\vec{T} \leftarrow \vec{T} \parallel \mathsf{target} \parallel \cdots \parallel \mathsf{target}$ (extended by $|\vec{N}_{\mathsf{df}}|$ elements) and store the vectors in s_{ep}
$\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}} \parallel \tau_L \parallel \cdots \parallel \tau_L, \vec{\mathtt{hf}} \leftarrow \vec{\mathtt{hf}} \parallel 1 \parallel \cdots \parallel 1$ (extended by $|\vec{N}_{\mathsf{df}}|$ elements) and store the vectors in s_{ep}
return $\vec{N}_{\mathsf{df}}$ and new state s_{ep}
end if

E.7 Algorithm extendchain

Algorithm $\mathsf{extendchain}^{T_0,m,\mathsf{b}_{\mathsf{tgt}}}(\mathcal{C}, \mathtt{st}, q, \tau)$

Input: Chain $\mathcal{C}$ is valid with state $\vec{\mathtt{st}}$. The state $\vec{\mathtt{st}} \parallel \mathtt{st}$ is valid.
Set $\mathbf{B} \leftarrow \perp$
$\mathsf{s} \leftarrow H[\mathsf{head}(\mathcal{C})]$
Compute $T \leftarrow \mathsf{gettarget}^{m,T_0,\tau,\mathsf{b}_{\mathsf{tgt}}}(\vec{\mathtt{st}})$
for $i \in \{1, \ldots, q\}$ **do**
Choose nonce n uniformly at random from $\{0,1\}^\kappa$ and set $\mathbf{B} \leftarrow \langle \mathsf{s}, \mathtt{st}, \mathsf{n}, T, \tau \rangle$
if $H[\mathbf{B}] < T$ **then**
break
end if
end for
if $\mathbf{B} \neq \perp$ **then**
$\mathcal{C} \leftarrow \mathcal{C} \parallel \mathbf{B}$
end if
return $\mathcal{C}$

F Proofs

We first present a number of constraints that will be assumed in our analysis below.

F.1 Proof of Theorem 1

Proof. The proof proceeds similarly to Lemma 7.1 in [4]. We define a series of hybrid worlds to argue that the distribution on the joint views between $\mathsf{StateExchange}^{q,T_0,\tau,m}(P)$ and $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathbf{STX}}(\mathcal{P}, \Delta, p_H, p_A))$ are indistinguishable from one another.

The simulator $\mathsf{sim}_{\mathsf{stx}}$ is described in full below. In a nutshell, it simulates the random oracle by storing a table, and if a new entry is queried, it samples a new random value, only terminating when a collision is found. The simulator also stores a tree $\mathcal{T}$. When receiving a message

Constraint #	Restriction
1	$[1-(1+\delta)\gamma^2 f]^{\Delta} > 1-\epsilon$
2	$2\ell + 6\Delta \leq \frac{\epsilon m}{2(1+\delta)\gamma^2 f}$
3	$s \geq 2(1+\delta)\gamma^2 m/f$
4	$\texttt{windowSize} = \epsilon m$
5	$\epsilon \leq \frac{\delta}{8} \leq \frac{1}{8}$
6	$\ell = (\frac{4(1+3\epsilon)}{\epsilon^2 f[1-(1+\delta)+\gamma^2 f]^{\Delta+1}} \cdot \max(\Delta, \tau) \cdot \gamma^3 \cdot \lambda)$
7	$t_r \leq (1-\delta)h_r$
8	$\texttt{chaingrowth}(u,v) = (1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta}(1/2^{\kappa}) \sum_{i=u+\Delta}^{v-\Delta}(\lvert\mathcal{P}_i\rvert - \lvert\mathcal{H}_i\rvert)$

Table 2. Admissible parameters for the protocol execution.

(SUBMIT-NEW, $sid, \vec{\mathsf{st}}, P_s, (P_1, \hat{\mathsf{mid}}_1), \ldots, (P_n, \hat{\mathsf{mid}}_n)$), if an extension to the state occurs, the simulator finds a corresponding chain $\mathcal{C} \in \mathcal{T}$, and sets the hash of the block to be strictly less than the required target before adding it to the tree. Whether or not an extension occurs, the simulator generates message ids, appending a unique message to each party with an associated delay parameter where the message contains the chain.

Let $\mathsf{HYB0}_{\mathcal{A},\mathcal{Z}}$ refer to the random experiment of $\mathsf{StateExchange}^{q,T_0,\tau,m}(P)$. When clear from context, we use shorthand $\mathsf{HYB0}_{\mathcal{A},\mathcal{Z}} := \mathsf{HYB0}$. Next, let $\mathsf{HYB1}_{\mathcal{A},\mathcal{Z}}$ act as $\mathsf{HYB0}$, with the difference being that $\mathsf{HYB1}$ may abort with COLLISION-ERROR or TREE-ERROR. Let $\mathsf{HYB2}_{\mathcal{Z}}$ refer to the experiment $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$ with $\mathsf{sim}_{\mathsf{stx}}$ (i.e $\mathrm{EXEC}_{(\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P},\Delta,p_H,p_A)),\mathsf{sim}_{\mathsf{stx}},\mathcal{Z})}$)

We start by showing that $\mathsf{HYB1} \approx \mathsf{HYB0}$. COLLISION-ERROR occurs when $H[v] = H[v']$ where $v, v' \in \{0,1\}^{\kappa}$ and $v \neq v'$. By definition, the probability of a two hashes colliding is $O(2^{-\kappa})$, and the probability of a hash collision throughout a polynomial length execution is $O(\mathsf{poly}(\kappa) \cdot 2^{-\kappa})$ by the union bound. This implies a collision, and thus COLLISION-ERROR occurs with negligible probability in κ

Let event refer to the circumstance where $H[(s \parallel \cdot \parallel \cdot \parallel \cdot \parallel \cdot)]$ is queried but no v exists where $H[v] = s$, but later a party queries v' such that $H[v'] = s$. By definition of hash collision, the probability over the execution is $O(\mathsf{poly}(\kappa) \cdot 2^{-\kappa})$ by the union bound. This occurs with negligible probability in κ.

If event doesn't occur, and no collision occurs, TREE-ERROR doesn't occur. We prove by contradiction: assume that $\mathsf{HYB1}_{\mathcal{A},\mathcal{Z}}$ aborts with TREE-ERROR w/ noticeable probability. Let $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ such at $\mathcal{C}$ is the shortest valid chain in $\mathsf{HYB1}_{\mathcal{A},\mathcal{Z}}$ where $\mathcal{C} \in \mathcal{T}$ but $\mathbf{B}_1, \ldots, \mathbf{B}_{n-1} \notin \mathcal{T}$.

By definition of valid chain, $H[s_n \parallel \mathtt{st}_n \parallel \mathsf{n}_n \parallel T_n \parallel t_n] < \mathsf{gettarget}(\vec{\mathtt{st}}, m, T_0, \tau, \mathsf{b_{tgt}})$.

However, by assumption we know that no valid chain has s_n as the hash of it's last block, otherwise it would be in $\mathcal{T}$. This means that no such query was made to $H[v] = s_n$. This implies either a collision occurred, or event occurred, which contradicts our assumption. Thus, we conclude TREE-ERROR occurs only with negligible probability.

Consider that in the hybrid world, changing T_i or t_i after querying the random oracle would result in a different hash. This only would go unnoticed if for some $\mathbf{B}_i \neq \mathbf{B}'_i$ where the timestamps or the target differ, that $H[\mathbf{B}_i] = H[\mathbf{B}_{i'}]$, which only happens when there is a collision. Thus, $\mathsf{HYB1} \approx \mathsf{HYB0}$.

Finally, we show that $\mathsf{HYB2} \approx \mathsf{HYB1}$. Recall that queries in the HYB1 world store the results of failed queries (i.e $H[\mathbf{B}_i] > \mathsf{gettarget}(\vec{\mathtt{st}}, m, T_0, \tau, \mathsf{b_{tgt}})$), but queries in HYB2 do not. Consider a failed query $H[\mathbf{B}_i]$. If and only if a later query $H[\mathbf{B}_{i'}] = H[\mathbf{B}_i]$ is made would the executions be distinguishable. This occurs only when there is a collision, which again only occurs with probability $O(\mathsf{poly}(\kappa) \cdot 2^{-\kappa})$. Therefore, we conclude that $\mathsf{HYB2} \approx \mathsf{HYB1}$, which concludes the proof.

Simulator $\mathsf{sim_{stx}}$

Initialization:
Set up a tree of valid chains $\mathcal{T} \leftarrow \{(\mathbf{G})\}$ and an empty network buffer $\vec{M}$. Set up an empty random oracle table H and set $H[\mathbf{G}]$ to a uniform value in $\{0,1\}^\kappa$. If the simulator ever tries to add a colliding entry to H, abort with COLLISION-ERROR.

Simulating the Random Oracle:

- Upon receiving (EVAL, sid, v) for $\mathcal{W}^q(\mathcal{F}_{\text{RO}}^\kappa)$ from $\mathcal{A}$ on behalf of corrupted $P \in \mathcal{P}^a$ do the following:
  1. If H[v] is already defined, output $(\text{EVAL}, sid, v, H[v])$.
  2. If v is of the form $(\mathsf{s}, \mathtt{st}, \mathsf{n}, T, \tau)$, and there exists a chain $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ such that $H[\mathbf{B}_n] = \mathsf{s}$ proceed as follows. If $\mathcal{C} \notin \mathcal{T}$ abort with TREE-ERROR. Otherwise continue. Extract the state $\vec{\mathtt{st}}$ from $\mathcal{C}$ and extract the state block $\mathtt{st}$ from v. Send $(\text{SUBMIT-NEW}, sid, \vec{\mathtt{st}}, \mathtt{st})$ to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\text{STX}}(\mathcal{P}, \Delta, p_H, p_A))$ and denote by $(\text{SUCCESS}, B)$ the output of $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\text{STX}}(\mathcal{P}, \Delta, p_H, p_A))$.
  3. If $B \neq 1$, or the above condition is not true, set v to a uniform random value in $\{0,1\}^\kappa$ and output $(\text{EVAL}, sid, v, H[v])$.
  4. If $B = 1$, set $H[v]$ to some uniformly random value in $\{0,1\}^\kappa < \mathsf{gettarget}(\vec{\mathtt{st}}, m, T_0, \tau, \mathsf{b_{tgt}})$ and output $(\text{EVAL}, sid, v, H[v])$.

Simulating the Network:

– Upon receiving $(\text{MULTICAST}, sid, (m_{i_1}, P_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell}))$ for $\mathcal{F}_{\text{DIFF}}^{\mathsf{bc}}$ from $\mathcal{A}$ on behalf of corrupted $P \in \mathcal{P}$ with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{P}_{net}$ proceed as follows:
  1. For each (m_{i_j}, P_{i_j}) where m_{i_j} is a chain in $\mathcal{T}$ extract the state $\vec{\mathtt{st}}_{i_j}$ from m_{i_j}, and send $(\text{SUBMIT-NEW}, sid, \vec{\mathtt{st}}, P_{i_j})$ to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$. Store the message-ID $\hat{mid}_{i_j}$ returned by $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$ with mid_{i_j}. Note that if P has not yet received that state, it is first fetched by $\mathcal{A}$ on behalf of P and if an unknown state is encoded, a random oracle query is simulated for the input to simulate the chain's validity and its possible inclusion into $\mathcal{T}$.
  2. For all remaining messages that could not be parsed as states, simply inject them as chunk messages to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$ to obtain their mid.
  3. Denote the obtained message-IDs by $\mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_\ell}$, initialize ℓ new variables $D_{\mathsf{mid}_1} := \cdots := D_{\mathsf{mid}_\ell} := 1$, set $\vec{M} := \vec{M} \parallel (m_{i_1}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}) \parallel \cdots \parallel (m_{i_\ell}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_\ell}})$
  4. Output $(\text{MULTICAST}, sid, (m_{i_1}, P_{i_1}, \mathsf{mid}_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell}, \mathsf{mid}_{i_\ell}))$
– Upon receiving (FETCH, sid) for $\mathcal{F}_{\text{DIFF}}^{\mathsf{bc}}$ from $\mathcal{A}$ on behalf of corrupted $P \in \mathcal{P}_{net}$ proceed as follows.
  1. Fetch in the name of party P from $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$ and compute the list of message identifiers $\mathsf{mid}_1, \ldots, \mathsf{mid}_\ell$ for which $D_{\mathsf{mid}_i} \leq 0$.
  2. Let $\vec{M}_0^P$ denote the subvector $\vec{M}$ formed by all tuples $(m, \mathsf{mid}, D_{\mathsf{mid}}, P)$ in the same order as they appear in $\vec{M}$, where mid appears in the above list. Delete all entries in $\vec{M}_0^P$ from $\vec{M}$, and send $\vec{M}_0^P$ to $\mathcal{A}$
– Upon receiving a message $(\text{DELAYS}, sid, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ do the following for each pair (T_{mid}, mid) in this message:
  1. If T_{mid} is a valid delay (i.e it encodes an integer in unary notation) and mid is a message-ID registered in the current $\vec{M}$, set $D_{mid} := \max\{1, D_{mid} + T_{mid}\}$; otherwise ignore this tuple.
  2. If the simulator knows a corresponding $\mathcal{F}_{\text{StX}}$-message-ID $\hat{\mathsf{mid}}$ for mid send $(\text{DELAY}, sid, T_{\mathsf{mid}}, \hat{\mathsf{mid}})$ to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$
– Upon receiving a message $(\text{SWAP}, sid, \mathsf{mid}_1, \mathsf{mid}_2)$ from the adversary do the following:
  1. If mid_1 and mid_2 are message-IDs registered in the current $\vec{M}$, then swap the tuples in $\vec{M}$.
  2. If the simulator knows for both mid_1 and mid_2 $\mathcal{F}_{\text{StX}}$-message-IDs mid_1 and mid_2 send $(\text{SWAP}, sid, \hat{\mathsf{mid}}_1, \hat{\mathsf{mid}}_2)$ to $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$.
  3. Output (SWAP, sid) to $\mathcal{A}$.

Interaction with the State Exchange Functionality :

– Upon receiving $(\text{SUBMIT-NEW}, sid, \vec{\mathtt{st}}, P_s, (P_1, \hat{\mathsf{mid}}_1), \ldots, (P_n, \hat{\mathsf{mid}}_n))$ from $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}}(\mathcal{P}, \Delta, p_H, p_A))$ where $\vec{\mathtt{st}} = \mathtt{st}_1, \ldots, \mathtt{st}_k$ and $\{P_1, \ldots, P_n\} := \mathcal{P}_{net}$ proceed as follows

1. If there exist a chain $\mathcal{C} \in \mathcal{T}$ with state $\vec{\mathtt{st}}$ generate new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$, initialize $D_1 := \cdots := D_n = 1$, set $\vec{M} \parallel (\mathcal{C}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}, P_1) \parallel \cdots \parallel (\mathcal{C}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_1}}, P_n)$, and store the message-IDs $\hat{\mathsf{mid}}_i$ along the message-IDs mid_i. Output $(\text{MULTICAST}, sid, \mathcal{C}, P_s, (P_1, \mathsf{mid}_1), \ldots, (P_n, \mathsf{mid}_n))$ to the adversary.
2. Otherwise find a chain $\mathcal{C}'$ in $\mathcal{T}$ with state $\mathtt{st}_1, \ldots, \mathtt{st}_{k-1}$. Choose a random nonce n and set $\mathbf{B}_k = (H[\mathbf{B}_{k-1}], \mathtt{st}_k, \mathsf{n}, T_k, t_k)$ and set $H[\mathbf{B}_k]$ to a uniform random value in $\{0,1\}^\kappa$ strictly smaller than $T_k = \mathsf{gettarget}(\vec{\mathtt{st}}, m, T_0, \tau, \mathsf{b}_{\mathsf{tgt}})$. Add the chain $\mathcal{C} = \mathcal{C}' \parallel \mathbf{B}_k$ to $\mathcal{T}$. Generate new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$, initialize $D_1 := \cdots := D_n = 1$, set $\vec{M} \parallel (\mathcal{C}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}, P_1) \parallel \cdots \parallel (\mathcal{C}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_1}}, P_n)$, and store the message-IDs $\hat{\mathsf{mid}}_i$. Output $(\text{MULTICAST}, sid, \mathcal{C}, P_s, (P_1, \mathsf{mid}_1), \ldots, (P_n, \mathsf{mid}_n))$ to the adversary.

□

F.2 Proof of Theorem 2

Proof. Now, we need to show that $\text{EXEC}^{T_0, m, \mathsf{b}_{\mathsf{tgt}}, q, \tau, \mathtt{windowSize}}_{\mathsf{ModularLedger}, \mathcal{A}, \mathcal{Z}} \approx \text{EXEC}_{\mathcal{W}(\bar{\mathcal{G}}_{\text{LEDGER}}), \mathsf{sim}_{\mathsf{stx}}, \mathcal{Z}}$; essentially, that the execution of the protocol in the real world against the adversary is indistinguishable from the execution of the ideal functionality in the ideal world against a simulator. We provide the code of our simulator in the appendixF.2. In a nutshell, we will show the restrictions on the wrapper are necessary to uphold the functionality, as well as desirable properties for the ledger.

As in the work of [4], the simulator itself is able to simulate the protocol without issue, but is limited by the restrictions imposed upon it, including the $\mathsf{ExtendPolicy}(\vec{\mathcal{I}}^T_H, \mathsf{state}, \mathsf{buffer}, \mathtt{NxtBC}, \vec{\tau}_{\mathtt{state}})$, the restriction on party registration, and the common prefix. We start with the following observation about the simulator: $\mathsf{sim}_{\mathsf{stx}}$ will not introduce parties into its simulation beyond that of the (γ, s) restriction. This is obvious, as if it did, then it would be trivial to distinguish from the real world.

We first turn our attention to the problem of common prefix, and show that it is satisfied with overwhelming probability. First, observe that:

Lemma 1. *If for some $\eta \in \mathbb{N}$ a sequence $(n_r)_{r \in \mathbb{N}}$ is $(\gamma, s+\eta)$-respecting, then it is also (γ, s)-respecting.*

Proof. The proof is straightforward, assume a sequence $(n_r)_{r \in \mathbb{N}}$ is $(\gamma, s+\eta)$-respecting, but is not (γ, s)-respecting. This implies there is some set S of size s in which $\frac{\max_{r \in S} n_r}{\min_{r \in S} n_r} \geq \gamma$ If that is true, that implies that there is some set S' of size $s+\eta$ in which $\frac{\max_{r \in S'} n_r}{\min_{r \in S'} n_r} \geq \gamma$ is also true, which contradicts our assumption. □

Now, because of the restriction on s, we know that the (γ, s)-respecting restriction on party count also is a $(\gamma, 2(1+\delta)\gamma^2 m/f)$-respecting sequence as well. Because of the restrictions in 2, we can apply Theorem 3 to conclude that these restrictions enforce typical executions and ensure that parties do not enter or leave too quickly. This directly shows that the $\mathsf{ModularLedger}^{T_0,m,\mathsf{b}_{\mathsf{tgt}},q,\tau,\mathtt{windowSize}}$ implements the common prefix restrictions on the ledger functionality. Consistency follows as a direct consequence, by applying Theorem 7

Now, we turn our attention to the lower bound for chain growth. This restriction is captured by `chaingrowth`, which requires that a minimum amount of difficulty has been contributed on the chain within a given amount of time. Recall that from Definition 6, we have that a typical execution bounds the random variable $Q(S)$ such that $(1-\epsilon)[1-(1+\delta)\gamma^2 f]^{\Delta}(1/2^{\kappa})\sum_{r\in S}(n_S) < Q(S)$, where S is a set consisting of at least ℓ consecutive rounds. Also, because of the restrictions in Table 2, the execution in the real world is typical by 3.

We know that by the restriction on s, and by applying Lemma 1, we can then directly apply Theorem 5 to our circumstance. In particular, it applies with overwhelming probability, as typical executions occur with overwhelming probability due to Theorem 3. Therefore, we have that $\mathsf{ModularLedger}^{T_0,m,\mathsf{b}_{\mathsf{tgt}},q,\tau,\mathtt{windowSize}}$ implements the chain growth restrictions on the ledger functionality.

Chain quality follows a similar pattern. We know that we know that by the restriction on s, and by again applying Lemma 1, we can then directly apply Theorem 5. This again means that in the real world, the adversary cannot contribute too many blocks in a given period of time under the restrictions of Table 2.

Now, by Theorem 8 we conclude that the resulting ledger satisfies the Liveness property. The Theorem follows.

Simulator $\mathsf{sim}_{\mathsf{ledg}}$

Initialization:
The simulator internally manages a simulated state-exchange functionality (through its wrapper $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{STX}}(\mathcal{P}, \Delta, p_H, p_A))$, and a simulated network $\mathcal{F}_{\text{DIFF}}$. An honest miner P registered to $\vec{\mathcal{G}}^{\vec{\mathbb{B}}}_{\text{LEDGER}}$ is simulated as registered in all simulated functionalities. Moreover the simulator maintains the local state $\vec{\mathtt{st}}_P$ and the buffer of transactions buffer_P of such a party. 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.

General Structure:
The simulator internally runs adversary $\mathcal{A}$ in a black-box way and simulates the interaction between $\mathcal{A}$ and the (emulated) hybrid functionalities. The inputs from $\mathcal{A}$ to the clock are relayed (and the replies given back to $\mathcal{A}$).

Messages from the Clock:

- Upon receiving (CLOCK-UPDATE, cid, P), first check whether the clock for the challenge session has advanced from time τ to $\tau + 1$ due to this clock-update activation. If this is the case then do the following:
  1. If P is the identity of the ledger functionality, then inspect $\vec{\mathcal{I}}_H^T$ (obtained via a read request) and check which miner P has issued the last (MAINTAIN-LEDGER, $sid, minerID$) request. Conclude the final step of (the interruptible computation of) SIMULATEMINING($P_{minerID}, \tau$) for this party. And in case τ is a working mini-round, execute EXTENDLEDGERSTATE before sending the final (CLOCK-UPDATE, cid, P) to the adversary.
  2. If P is not the identity of the ledger functionality and τ is a working mini-round, then execute EXTENDLEDGERSTATE before outputting (CLOCK-UPDATE, cid, P) to $\mathcal{A}$

  If no such clock advancement occurs, then do the following:
  1. If the identity P corresponds to this ledger functionality, then inspect $\vec{\mathcal{I}}_H^T$ (obtained via a read request) and check which miner P has issued the last (MAINTAIN-LEDGER, $sid, minerID$) request. Conclude the final step of (the interruptilble computation of) SIMULATEMINING($P_{minerID}, \tau$) for this party.
  2. If P is not the identity of the ledger functionality, then just output (CLOCK-UPDATE, cid, P) to $\mathcal{A}$

Message from the Ledger:

- Upon any input from the ledger, the simulator first inspects $\vec{\mathcal{I}}_H^T$ (obtained by reading from the ledger functionality) and obtains the time τ and if τ is an update mini-round, it executes, for each party P that had $I = (\text{READ}, sid)$ in this round, the fetch-information step of procedure SIMULATEMINING before proceedings with the specific actions below.
- Upon receiving (SUBMIT, BTX) from $\bar{\mathcal{G}}_{\text{LEDGER}}^{\bar{\mathbb{B}}}$ where $\texttt{BTX} := (\texttt{tx}, txid, \tau, P)$ forward (MULTICAST, sid, tx) to the simulated network $\mathcal{F}_{\text{DIFF}}$ in the name of P. Output the answer of $\mathcal{F}_{\text{DIFF}}$ to the adversary.
- Upon receiving (MAINTAIN-LEDGER, $sid, minerID$) from $\bar{\mathcal{G}}_{\text{LEDGER}}^{\bar{\mathbb{B}}}$, extract from $\vec{\mathcal{I}}_H^T$ (obtained by reading from the ledger functionality) the identity P_i that issued this query. If P_i is already done in this mini-round, then ignore the request. Otherwise, execute (as an interruptible computation) the procedure SIMULATEMINING($P_{minerID}, \tau$) for this party.

Simulation of the State Exchange Functinonality:

- Upon receiving (SET-DELAY, sid, n) from the adversary $\mathcal{A}$, relay the input to the simulated $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$, and return whatever response is received to $\mathcal{A}$
- Upon receiving (SUBMIT-NEW, $sid, \vec{\mathtt{st}}, \mathtt{st}$) from $\mathcal{A}$ on behalf of a corrupted $P \in \mathcal{P}_{stx}$, then relay it to the simulated $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$, and do the following:
  1. When $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$ returns (SUCCESS, B) give this reply to $\mathcal{A}$.
  2. If the current mini-round is an update mini-round, then execute ExtendLedgerState
- Upon receiving (FETCH-NEW, sid) from $\mathcal{A}$ (on behalf of a corrupted P), forward the request to the simulated $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (SEND, sid, s, P') from $\mathcal{A}$ on behalf of some *corrupted* party P, do the following:
  1. Forward the request to the simulated $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$.
  2. If the current mini-round is an update mini-round, then execute ExtendLedgerState.
  3. Return to $\mathcal{A}$ the return value from $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$.
- Upon receiving (SWAP, $sid, \mathsf{mid}, \mathsf{mid}'$) from $\mathcal{A}$, forward the request to the simulated $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$ and return whatever is returned to $\mathcal{A}$
- Upon receiving (DELAY, sid, T, mid) from $\mathcal{A}$, forward the request to the simulated $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$ and do the following:
  1. Query the ledger state state
  2. Execute AdjustView(state)
  3. Return to $\mathcal{A}$ the output of $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{StX}})$

Simulation of the Network *(over which transactions are sent) :*

- Upon receiving (MULTICAST, $sid, (m_{i_1}, P_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell})$) with list of transactions from $\mathcal{A}$ on behalf of some *corrupted* $P \in \mathcal{P}_{net}$, then do the following:
  1. Submit the transactions to the ledger on behalf of this corrupted party, and receive the transaction id $\mathtt{txid}$ for each transaction
  2. Forward the request to the internally simulated $\mathcal{F}_{\text{DIFF}}$, which replies for each message with a message-ID mid
  3. Remember the association between each mid and the corresponding $\mathtt{txid}$
  4. Provide $\mathcal{A}$ with whatever the network outputs.
- Upon receiving (an ordinary input) (MULTICAST, sid, m) from $\mathcal{A}$ on behalf of some *corrupted* $P \in \mathcal{P}_{net}$, then execute the corresponding steps 1. to 4. as above.
- Upon receiving FETCH-NEW, sid) from $\mathcal{A}$ on behalf of some *corrupted* $P \in \mathcal{P}_{net}$, forward the request to the simulated $\mathcal{F}_{\text{DIFF}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (DELAY, $sid, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell})$) from $\mathcal{A}$, forward the request to the simulated $\mathcal{F}_{\text{DIFF}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (SWAP, $sid, \mathsf{mid}, \mathsf{mid}'$) from $\mathcal{A}$, forward the request to the simulated $\mathcal{F}_{\text{DIFF}}$ and return whatever is returned to $\mathcal{A}$.

Simulation of Corruptions:

– Upon corruption of a party $P \in \mathcal{P}$, corrupt the party in all hybrid functionalities and the clock, and remember this party as corrupted. If the corruption leads to a clock advancement, then execute the sane steps as above upon a (CLOCK-UPDATE, cid, P) from $\bar{\mathcal{G}}_{\text{CLOCK}}$

procedure SIMULATEMINING(P, τ)

Simulate the (interruptible) mining procedure of P of the ledger protocol:
if time-tick τ corresponds to a working mini-round and P is not done yet **then**
Execute Step 2 of the mining protocol. This includes:
-Define the next state block $\mathtt{st}$ using the transaction set buffer_P
-Send (SUBMIT-NEW, $sid, \vec{\mathtt{st}}, \mathtt{st}$) to simulated functionality $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\text{STX}})$
-If successful, store $((\vec{\mathtt{st}}_P \parallel \mathtt{st}), (\vec{T} \parallel T), (\tau \parallel \tau_s))$ as the new $\vec{\mathtt{st}}_P, \vec{T}, \vec{\tau}$
-If successful, distribute the new state via $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\text{STX}})$
-If done with all actions, the last action is outputting (CLOCK-UPDATE, cid, P) to $\mathcal{A}$
else if time-tick τ corresponds to an update sub-round and P is not done yet **then**
Execute Step 3 of the mining protocol. This means that if the new information has not been fetched in this round already, then the following is executed:
-Fetch transactions $(\mathtt{tx}_1, \ldots, \mathtt{tx}_u)$ (on behalf of P) from simulated $\mathcal{F}_{\text{DIFF}}$ and add them to buffer_P
-Fetch states $(\vec{\mathtt{st}}_1, T_1, \tau_1) \ldots, (\vec{\mathtt{st}}_s, T_1, \tau_1)$ (on behalf of P) from the simulated $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\text{STX}})$ and update $\vec{\mathtt{st}}_P$
-If done with all actions, the last action is outputting (CLOCK-UPDATE, cid, P) to $\mathcal{A}$
end if

procedure EXTENDLEDGERSTATE

Consider all honest and snychronized players P:
-Let $\vec{\mathtt{st}}$ be the longest state among all states $\vec{\mathtt{st}}_P$ or states contained in a receiver buffer $\vec{M}_P$ with delay 1 (and hence is a potential output in the next round)
Compare $\vec{\mathtt{st}}^{\lceil \mathtt{windowSize}}$ with the current state state of the ledger
if $|\mathsf{state}| > |\vec{\mathtt{st}}^{\lceil \mathtt{windowSize}}|$ **then** ‘
Execute ADJUSTVIEW(state)
end if
if state is not a prefix of $\vec{\mathtt{st}}^{\lceil \mathtt{windowSize}}$ **then**
Abort the simulation (due to inconsistency)
end if
Define the difference diff to be the block sequence s.t $\mathsf{state} \parallel \mathsf{diff} = \vec{\mathtt{st}}^{\lceil \mathtt{windowSize}}$
Let $n \leftarrow |\mathsf{diff}|$
for each block diff_j, $j = 1$ to n **do**
Map each transaction $\mathtt{tx}$ in this block to its unique transaction ID txid
If a transaction does not yet have a txid, then submit it to the ledger and receive the corresponding txid from $\bar{\mathcal{G}}^{\mathsf{B}}_{\text{LEDGER}}$

Let $\mathsf{list}_j = (txid_{j,1}, \ldots, txid_{j,\ell_j})$ be the corresponding list for this block.

Output (NEXT-BLOCK, list_j) to $\bar{\mathcal{G}}_{\text{LEDGER}}^{\vec{\mathbb{B}}}$ (receiving (NEXT-BLOCK, ok) as an immediate answer)

end for

Execute ADJUSTVIEW($\mathsf{state} \parallel \mathsf{diff}$)

procedure ADJUSTVIEW(state)

$\mathsf{pointers} \leftarrow \epsilon$

for each honest and synchronized party P_i **do**

Using the simulated functionality $\mathcal{F}_{\mathtt{STX}}$ do the following:

-Let $\vec{\mathtt{st}}$ be the longest state among $\vec{\mathtt{st}}_{P_i}$ and those contained in the receiver buffer $\vec{M}_{P_i}$ with delay 1

Determine the pointer $\mathtt{pt}_i$ s.t $\vec{\mathtt{st}}^{\lceil \mathtt{windowSize}} = \mathsf{state}|_{\mathtt{pt}_i}$

if such a pointer value does not exist **then**

Abort simulation (due to inconsistency)

end if

if Party P_i has not executed step 3 of the mining protocol in this current mini-round **then**

$\mathsf{pointers} \leftarrow \mathsf{pointers} \parallel (P_i, \mathtt{pt}_i)$

end if

end for

Output (SET-SLACK, $\mathsf{pointers}$) to $\bar{\mathcal{G}}_{\text{LEDGER}}^{\vec{\mathbb{B}}}$

$\mathsf{pointers} \leftarrow \epsilon$

$\mathsf{desyncStates} \leftarrow \epsilon$

for each honest and de-synchronized party P_i **do**

Using the simulated functionality $\mathcal{W}^{\mathsf{stx\text{-}params}}(\mathcal{F}_{\mathtt{STX}})$ do the following:

-Let $\vec{\mathtt{st}}$ be the longest state among $\vec{\mathtt{st}}_{P_i}$ and those contained in the receiver buffer $\vec{M}_{P_i}$ with delay 1

if Party P_i has not executed step 4 of the mining protocol in this current mini-round **then**

Set the pointer $\mathtt{pt}_i$ to be $\vec{\mathtt{st}}^{\lceil \mathtt{windowSize}} = \mathsf{state}|_{\mathtt{pt}_i}$

$\mathsf{pointers} \leftarrow \mathsf{pointers} \parallel (P_i, \mathtt{pt}_i)$

$\mathsf{desyncStates} \leftarrow \mathsf{desyncStates} \parallel (P_i, \vec{\mathtt{st}}^{\lceil \mathtt{windowSize}})$

end if

end for

Output (SET-SLACK, $\mathsf{pointers}$) to $\bar{\mathcal{G}}_{\text{LEDGER}}^{\vec{\mathbb{B}}}$

Output (DESYNC-STATE, $\mathsf{desyncStates}$) to $\bar{\mathcal{G}}_{\text{LEDGER}}^{\vec{\mathbb{B}}}$

□