J Cryptol (2024)37:18
https://doi.org/10.1007/s00145-024-09493-7

Journal of CRYPTOLOGY

Research Article

Bitcoin as a Transaction Ledger: A Composable Treatment*

Christian Badertscher
Input Output, Zürich, Switzerland
christian.badertscher@iohk.io

Ueli Maurer
ETH Zurich, Zürich, Switzerland
maurer@inf.ethz.ch

Daniel Tschudi
Concordium, Cham, Switzerland
dt@concordium.com

Vassilis Zikas
Purdue University, West Lafayette, USA
vzikas@cs.purdue.edu

Communicated by Manoj Prabhakaran

Received 19 November 2018 / Revised 15 January 2024 / Accepted 15 January 2024
Online publication 4 April 2024

Abstract. Bitcoin is one of the most prominent examples of a distributed cryptographic protocol that is extensively used in reality. Nonetheless, existing security proofs are property-based, and as such they do not support composition. In this work, we put forth a universally composable treatment of the Bitcoin protocol. We specify the goal that Bitcoin aims to achieve as an instance of a parameterizable ledger functionality and present a UC abstraction of the Bitcoin blockchain protocol. Our ideal functionality is weaker than the first proposed candidate by Kiayias, Zhou, and Zikas [EUROCRYPT'16], but unlike the latter suggestion, which is arguably not implementable by the UC Bitcoin protocol, we prove that the one proposed here is securely UC-realized by the protocol assuming access to a global clock, to model time-based executions, a random oracle, to model hash functions, and an idealized network, to model message dissemination. We further show how known property-based approaches can be cast as special instances of our treatment and how their underlying assumptions can be cast in UC as part of the setup functionalities and without restricting the environment or the adversary.

*An earlier version of this work appeared as an extended abstract in the proceedings of the 37th International Cryptology Conference (CRYPTO 2017) available at https://doi.org/10.1007/978-3-319-63688-7_11. C. Badertscher: Lead contributor to this extended article. Earlier version done while the author was at ETH Zurich. D. Tschudi: Work done while the author was at ETH Zurich. V. Zikas: Work done while the author was at RPI

Keywords. Bitcoin, Blockchain, Ledger functionality, Universal composition, Provable security.

1. Introduction

Since Nakamoto first proposed Bitcoin as a decentralized cryptocurrency [41], several early works have focused on analyzing and/or predicting its behavior under different attack scenarios [5,22,23,32,43,47,48]. However, a core question remained unanswered for quite a while:

What security goal does Bitcoin achieve under what assumptions?

An intuitive answer to this question was already given in Nakamoto's original white paper [41]: Bitcoin aims to achieve some form of consensus on a set of valid transactions. The core difference of this consensus mechanism with traditional consensus [37–39,45] is that it does not rely on having a known (permissioned) set of participants, but everyone can join and leave at any point in time. This is often referred to as the *permissionless* model. Consensus in this model is achieved by shifting from the traditional assumptions on the fraction of cheating versus honest participants, to assumptions on the collective computing power of the cheating participants compared to the total computing power of the parties that support the consensus mechanism. The core idea is that in order for a party's action to affect the system's behavior, it needs to prove that it is investing sufficient computing resources. In Bitcoin, these resources are measured by means of solutions to a presumably computation-intensive problem.

Although the above idea is implicit in [41], a formal description of Bitcoin's goal had not been proposed or known to be achieved (and under what assumptions) until the first, seminal works of Garay, Kiayias, and Leonardos [24] and Pass, Seeman, and shelat [44], which mainly influenced this work. In a nutshell, these works set forth models of computation and, in these models, an abstraction of Bitcoin as a distributed protocol and proved that the output of this protocol satisfies certain security properties, for example the *common prefix* [24] or consistency [44] property. This property confirms—under the assumption that not too much of the total computing power of the system is invested in breaking it, where the exact threshold has been the study of recent works [20,27]—a heuristic argument used by the Bitcoin specification: if some block makes it deep enough into the blockchain of an honest party, then it will eventually make it into the blockchain of every honest party and will never be reversed.[1] In addition to the common prefix property, other quality properties of the output of the abstracted blockchain protocol were also defined and proved.

1.1. *Bitcoin as a Service for Cryptographic Protocols*

Evidently, the main use of the Bitcoin protocol is as a decentralized monetary system with a payment mechanism, which is what it was designed for. And although the exact

[1] In the original Bitcoin heuristic "deep enough" is defined as six blocks, whereas in these works it is defined as linear in an appropriate security parameter.

economic forces that guide its sustainability are still being researched, and certain rational models predict it is not a stable solution, it is a fact that Bitcoin has not met any of these pessimistic predictions for several years and it is not clear it ever will do. And even if it does, the research community has produced and is testing several alternative decentralized cryptocurrencies, e.g., [6,12,19,30,31,40,42,46], that are more functional or based on different resource assumptions than Bitcoin, some of which base their analysis on earlier versions of this article [11].

This leads to the natural questions of how one can use this new reality to improve the security and/or efficiency of cryptographic protocols. First answers to this question were given in [1–3,10,28,29,33,35] where it was shown how Bitcoin can be used as a punishment mechanism to incentivize honest behavior in higher-level cryptographic protocols such as fair lotteries, poker, and general multi-party computation.

But in order to formally define and prove the security of the above constructions in a widely accepted cryptographic framework for multi-party protocols, one needs to define what it means for these protocols to be run in a world that gives them access to the Bitcoin network as a resource to improve their security. In other words, the question now becomes:

What functionality can Bitcoin provide to cryptographic protocols?

To address this question, Bentov and Kumaresan [10] introduced a model of computation in which protocols can use a punishment mechanism to incentivize adversaries to adhere to their protocol instructions. As a basis, they use the universal composition framework of Canetti [13], but the proposed modifications do not support composition and it is not clear how standard UC cryptographic protocols can be cast as protocols in that model.

In a different direction, Kiayias, Zhou, and Zikas [36] connected the above question with the original question of Bitcoin's security goal. More concretely, they proposed identifying the resource that Bitcoin (or other decentralized cryptocurrencies) offers to cryptographic protocols as its security goal, and expressing it in a standard language compatible with the existing literature on cryptographic multi-party protocols. More specifically, they modeled the ideal guarantees as a transaction-ledger functionality in the (global) universal composition framework.

In a nutshell, the ledger proposed by [36] corresponds to a trusted third party which keeps a state of blocks of transactions and makes it available, upon request, to any party. Furthermore, it accepts messages/transactions from any party and records them as long as they pass an appropriate validation procedure that depends on the above publicly available state as well as other registered messages. Periodically, this ledger puts the transactions that were recently registered into a block and adds them into the state. The state is available to everyone. As proved in [36], giving multi-party protocols access to such a transaction-ledger functionality allows for formally capturing the mechanism of leveraging security loss with coins. The proposed ledger functionality guarantees in an ideal manner all properties that one could expect from Bitcoin and encompasses the properties in [24,44]. Therefore, it is natural to postulate that it is a candidate for defining the security goal of Bitcoin (and potentially other decentralized cryptocurrencies). However, the ledger functionality proposed by [36] was not accompanied by a security proof that any of the known cryptocurrencies implements it.

However, as we show, despite being a step in the right direction, the ledger proposed in [36] cannot be realized under standard assumptions about the Bitcoin network. On the positive side, we specify a new transaction ledger functionality which still guarantees all properties postulated in [24,44], and prove that a reasonable abstraction of the Bitcoin protocol implements this ledger. In our construction, we describe Bitcoin as a UC protocol which generalizes both the protocols proposed in [24,44]. We leave it as an interesting open problem to integrate more recent analyses [7,17,25,26] in our UC model, where the main changes are expected in formulating the setup assumptions and restrictions along the lines we show in Sect. 8.1 for the initial models. Still, the main goal remains to UC-realize our ledger functionality.

1.2. *Our Contributions*

We put forth the first universally composable (simulation-based) proof of security of Bitcoin. We design a general ledger functionality whose parameters we subsequently concretely instantiate for the Bitcoin setting. We observe that the first attempts in defining such a functionality, notably the ledger proposed by Kiayias et al. [36], are too strong to be implemented by our UC abstraction of Bitcoin, the main reason being that the functionality allows too little interference of the simulator with its state, making it impossible to emulate adversarial attacks that result, e.g., in the adversary inserting only transactions coming from parties it wants or that result in parties holding chains of different lengths. We detail this in Sect. 4.1. Therefore, we propose an alternative ledger functionality in Sect. 4.2 which shares certain design properties with the proposal in [36] but which can be provably implemented by a UC abstraction of the Bitcoin protocol, where our protocol abstraction makes use of hybrid (idealized) functionalities such as the bounded-delay network, the clock to model (lock-step) synchrony, and the random oracle to idealize hash queries.

Our ledger is parametrized by a set of parameters, for example by a generic transaction validation predicate, which enables it to capture decentralized blockchain protocols beyond Bitcoin. Our functionality allows for parties/miners to join and leave the computation and we support adaptive corruptions.

We formally prove for which choice of parameters the proposed ledger functionality is implemented by Bitcoin under the assumption that miners which deviate from the Bitcoin protocol do not control a majority of the total hashing power at any point. The description of concrete parameters is given in Sect. 6, and the UC realization proof appears in Sect. 7. To this end, we first describe in detail an abstraction of the Bitcoin protocol as a UC protocol in Sect. 5. Casting Bitcoin in UC allows to precisely model the protocol assumptions, for example the knowledge of the network delay and the number of hash-function calls per round. We model Bitcoin to work over a network which basically consists of bounded-delay channels. We explain how such a network could be implemented by running the message-diffusion mechanism of the Bitcoin network (which is run over a lower level network of unicast channels). Intuitively, this network is built by every miner, upon joining the system, choosing some existing miners of its choice to use them as relay nodes. Similar to the protocol in [44], the miners are not aware of (an upper bound on) the actual delay that the network induces. As we argue, this is a strictly weaker model assumption than assuming that the network delay is publicly known such as in [24]. We devote Sect. 3 to modeling the UC execution with the appropriate setups.

Our security proof proposes a useful modularization of the Bitcoin protocol. Concretely, we first identify the part of the Bitcoin code which intuitively corresponds to the lottery aspect, provide an ideal UC functionality that reflects this lottery aspect, and prove that this part of the Bitcoin code realizes the proposed functionality. We then analyze the remainder of the protocol in the simpler world where the respective code that implements the lottery aspect is replaced by invocations of the corresponding functionality. Using the UC composition theorem, we can then immediately combine the two parts into a proof of the full protocol. Finally, in Sect. 8, we show how one can cast the theorem's assumptions as part of the setup functionalities of the protocol. We thus obtain a desirable corollary where we do not have to restrict the environment regarding the distribution of hashing power (but where the restriction is enforced by the setup functionalities), which improves the way this protocol can be formally composed with other protocols.

As is the case with the so-called *backbone* protocol from [24], our above UC protocol description of Bitcoin relies only on proofs of work and not on digital signatures. As a result, it implements a somewhat weaker ledger, which does not guarantee that transactions submitted by honest parties will eventually make it into the blockchain.[2] As a last result, we show in Sect. 9 that (similarly to [24]) by incorporating public-key cryptography, i.e., taking signatures into account in the validation predicate, we can implement a stronger ledger that ensures that transactions issued by honest users—i.e., users who do not sign contradicting transactions and who keep their signing keys for themselves—are guaranteed to be eventually included into the blockchain. The fact that our protocol is described in UC makes this a straight-forward, modular construction using the proposed transaction ledger as a hybrid. In particular, we do not need to consider the specifics of the Bitcoin protocol in the proof of this step. This also allows us to identify the maximum (worst-case) delay a user needs to wait before being guaranteed to see its transaction on the blockchain and be assured that it will not be inverted.

Future directions. The presented analysis in UC corresponds to the first analysis of a blockchain protocol and requires a couple of novel modeling concepts to accurately model the execution of such decentralized protocols whose security are based on rate-limiting resources. For Bitcoin in particular, a few interesting extensions to this work are conceivable which all relate to the topic of bringing the model closer to reality. First, it is an interesting open question to what extent the reliance on a local random oracle can be relaxed while still achieving composition. A second line of research would be to model a more realistic network functionality, taking into account limited message omissions or bandwidth constraints. Finally, the full Bitcoin protocol includes adjusting the difficulty of the PoW puzzles per epoch based on the observed historic performance. It appears as a very interesting theoretical question whether the Bitcoin protocol (or possibly a variant of it) can in fact be understood as the modular composition of a component estimating participation and a ledger component as analyzed in this work.

[2] We formulate a weakened guarantee, which we then amplify using digital signatures.

1.3. *Overview of Bitcoin and Related Work*

High-level introduction. At a high level, the Bitcoin protocol works as follows: The parties (also referred to as *miners*) collect and circulate messages (transactions) from users of the network, check that they satisfy some commonly agreed validity property, put the valid transactions into a block, and then try to find appropriate metadata such that the hash of the block-contents and this metadata is of a specific form—concretely that, parsed as a binary string, it has a sufficient number of leading zeros. This is often referred to as a solving a mining puzzle and the intuition behind it is that the best strategy for finding such metadata is supposedly by trial and error. Thus, informally, the probability that some party finds appropriate metadata increases proportional to the number of times some party attempts a hash computation. And the more leading zeros we require from a correct puzzle solution the harder it is to find one, since the solution space of the puzzle is smaller.

Intuitively, a successful solution can be seen as a *proof-of-work* (POW) that testifies to the fact that the miner presenting has in fact tried a large number of hash queries. Once a miner finds such a solution, he puts it into a block and sends it to the other miners. The miners who receive it check that it satisfies some validity property (see below) and if so create new metadata using the hash of this (newly minted) block and put this metadata together with transactions that are still valid into a new block and start working on solving the puzzle induced by this block. Since a block is rendered valid by a miner only if it includes a hash-pointer to a previous valid block in the view of this miner, the view consists of a set of linked lists, namely a sequence of valid blocks each with a hash-pointer to its predecessor in the list. Each such list is called a blockchain or simply chain. All lists have a common starting point which is the so-called genesis block of Bitcoin. Hence, the entire view of a miner could be modeled as a tree, where the root is the genesis block, the nodes are valid blocks, and the hash-pointers correspond to (directed) edges.

The works of Garay, Kiayias, and Leonardos [24] and that of Pass, Seeman, and shelat [44] contain the first formal specifications and security proofs of the Bitcoin protocol. The proved security in these works is property-based. They prove that conditioned on the largest part of the network following the Bitcoin protocol (in fact an abstraction and generalization thereof), the output of this so-called backbone protocol satisfies three properties with overwhelming probability. We only informally describe these properties here. We will meet their formalization when analyzing the Bitcoin protocol in UC. In the following, let $t_1 \leq t_2$ be two points in time during the protocol execution.

- *Common prefix*: Any two valid chains $\mathcal{C}_{t_1}$, $\mathcal{C}_{t_2}$ adopted by some honest parties at times t_1 and t_2, respectively, share a large common prefix. This is typically quantified by specifying a value k (the common-prefix parameter) and the size of the common prefix is required to be at least $|\mathcal{C}_{t_1}| - k$.
- *Chain growth*: For time-intervals $[t_1, t_2]$ of reasonable extent, the increase in number of blocks—measured as the difference between any two valid chains $\mathcal{C}_{t_1}$ and $\mathcal{C}_{t_2}$ adopted by some honest parties at times t_1 and t_2, respectively—is guaranteed to be substantial. The relationship between time and chain-length is typically referred to as the chain-growth coefficient.

- *Chain quality*: For any honest party and its adopted valid chain $\mathcal{C}_t$ at time t, it holds that any consecutive sequence of blocks of reasonable extent in $\mathcal{C}_t$ is guaranteed to contain blocks contributed by honest parties. The proportion of honestly mined blocks is typically refereed to as the chain-quality coefficient.

Chain quality and chain growth are often expressed with respect to the common-prefix parameter k. That is, as the fraction of honestly mined blocks in a consecutive sequence of k blocks, and as the time interval within which an increase of k blocks is guaranteed (except with negligible probability in k).

Network assumptions and random oracle. Both [24] and [44] assume a multicast network—i.e., a network where a party sends messages to arbitrary other parties[3]—and abstract the hash function as a random oracle. Furthermore, they both have an explicit round-based model of execution where parties proceed in rounds. There are some slight differences between the two models. For example, in [24] every party makes q hash-queries (i.e., q RO calls) in each round as opposed to [44] where every party makes one hash-query per round. Second, in [44], the adversary might choose to delay message delivery but the statements are proved assuming no message is delayed by more than Δ rounds—also known as the partial-synchronous setting—while the initial model taken by [24] was more synchronous (and was lifted to the partial synchronous model later). We note that since the number of hash-queries is fixed in both models, this implies that parties know exactly in which round they are, as they could simply count the number of queries made to the random oracle (and by definition of their models no party goes to round $r + 1$ before all parties have finished round r). Note that the partial-synchronous protocol execution model in [44] is a *strictly* weaker setting than a synchronous execution model with a fixed delay of one round.

Property-based vs simulation-based security. Proving that Bitcoin satisfies the above properties has been an essential step into the direction of understanding the security goals of Bitcoin. But as argued above, this does not offer the tool to be able to argue security of cryptographic protocols that use Bitcoin—e.g., to achieve an improved fairness notion [1–3,10,28,29,33,35]—without the need to always look at the Bitcoin specifics. In other words, such property-based security definitions do not support composition. The standard way to allow for such a generic use of blockchain protocols as a cryptographic resource is to prove that it implements an ideal functionality in a composable framework. Intuitively, in such frameworks, a composition theorem states that we can replace calls to a functionality with invocation of a protocol implementing it without worrying about the protocol's internals.

[3] Unlike [24] where this operation is referred to as broadcast, we choose to call it multicast here to avoid confusion with the standard broadcast primitive in the Byzantine agreement literature that offers stronger consistency guarantees.

2. Preliminaries

2.1. *Overview of the UC Framework*

We use the universal composability (UC) framework introduced by Canetti [13,14]. We give a brief introduction into the main notation of this framework.

2.1.1. *Basics*

The goal of the UC framework is to capture what it means for a protocol to securely carry out a task. UC first defines the process of executing a protocol in some environment and in the presence of an adversary, next it defines an ideal process to formalize what securely carrying out the task means, and finally one has to prove that no (efficient) environment can distinguish the real process and the ideal process. The core defining element of the ideal process is the ideal functionality, which can be thought of as an incorruptible party. We briefly describe the main ingredients first and then describe the real and ideal process.

Protocol and protocol instances. Formally, a protocol π is an algorithm for a distributed system and formalized as an interactive Turing machine. An ITM has several tapes, for example an identity tape (read-only), an activation tape, or input/output tapes to pass values to its program and return values back to the caller (e.g., the environment). An ITM also has communication tapes that model messages sent to and received from the network.

While an ITM is a static object, UC defines the notion of an ITM instance (denoted ITI), which is defined by the so-called *extended identity* (eid) of the form (M, id), where M is the description of an ITM and $id = (\mathsf{sid}, \mathsf{pid})$ is an identity string consisting of a session identifier sid and a party identifier pid. Each instance is associated with a configuration, which is as usual the contents of all of its tapes and the heads, and the control state of that ITM.

An instance, also called a session, of a protocol π (represented as an ITM M_π) with respect to a session id sid is defined as a set of ITIs (M_π, id_i) with $id_i = (\mathsf{pid}_i, \mathsf{sid})$.

Network and adversary. The UC model does not give any guarantee for its built-in network. The network is asynchronous without guaranteed delivery or authenticity of the originator. The messages are routed and controlled by the adversary unless a stronger network is available (such as the one we define in this work). The adversary $\mathcal{A}$ is also defined as an ITM. Aside of its capabilities to send and read messages, it can at any time issue special corruption messages to corrupt protocol ITMs. When an ITM is corrupted, the adversary does not only learn the contents of all tapes, but it can also act in the name of this ITM, meaning that whenever this ITM is activated, the adversary gets actually activated and can decide on the next steps. This corruption dynamics is the standard form of corruption and we call such an adversary active and adaptive.

2.1.2. *Real-World Process*

The real-world process for a protocol π is defined as follows. Let $\mathcal{Z}$ be an environment machine and let $\mathcal{A}$ denote the adversary. The execution consists of a sequence of activations, initiated by $\mathcal{Z}$, where in each activation, either $\mathcal{Z}$, $\mathcal{A}$ or some ITI running π is activated. We say that $\mathcal{Z}$ invokes a new ITI $\mathcal{Z}$ if it activates an ITI for the first time (by passing some inputs) upon which this new instance gets created (in the default configuration). All ITIs invoked by $\mathcal{Z}$ need to have unique extended identities and need to have the same session-identifier (which is chosen by $\mathcal{Z}$).

Activations and execution rules. An activated ITI can change its configuration based on its code. By the UC system model (i.e., by the definition of external-write requests), an ITI loses its activation when (1) passing an input value to a (subsidiary) ITI (like a hybrid functionality), or (2) producing an output, i.e., writing to its subroutine output tape, or (3) providing output to the adversary (e.g., by an ideal functionality). The next activated ITI is the ITI that was addressed in the external-write request, or the environment if no external-write request is made.

The environment $\mathcal{Z}$ can pass inputs to and read outputs from the input/output tape of any party, respectively. The environment can thereby emulate all outside processes and how they interact with the (challenge) protocol session. In these inputs, the environment thereby also specifies a source (extended) identity of the input (to which supposedly some output will be returned). We call such identities external. It is convenient to parametrize an environment with a predicate ξ that restricts the set of allowed external identities to use. One natural standard predicate to enforce is the one that disallows $\mathcal{Z}$ to use as an external identity an extended identity of any ITI that it provides input to in the system. Other choices of predicates may be helpful in various scenarios. Clearly, the more relaxed the predicate ξ, the more general the security statement. More restrictive predicates in turn lead to more restrictions on the contexts in which the protocols proved secure with respect to those predicates can be deployed.

The adversary $\mathcal{A}$ can access the so-called backdoor tapes of the ITIs and in the plain network model, thereby deliver messages. Following the external-write rules, if in some activation, the adversary delivers a message to an ITI, then this ITI is activated next. In addition, the adversary can corrupt parties as described above. The environment learns the party id pid of any corrupted ITI via a special corruption-aggregation mechanism.

The UC model also follows some activation rules (specified by the control function). As already stated, the environment is activated first, and upon completion of its actions (entering a special waiting state), the adversary is activated as a second entity. The remaining execution proceeds as described above. As a convention, in addition to the above rules, the UC execution model requires that if an ITI completes without external-write request (for example not sending a message), then the environment is activated next.

Output and transcript. The output of the protocol execution is the output of $\mathcal{Z}$, and we assume that this output is a binary value $v \in \{0, 1\}$. We denote this output by $\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}(k, z, r)$ where k is the security parameter, $z \in \{0, 1\}^*$ is the input to the environment, and randomness r for the entire experiment. Let $\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}(k, z)$ denote the random variable obtained by choosing the randomness r uniformly at random and evaluating $\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}(k, z, r)$. Let $\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the ensemble $\{\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}(k, z)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$. By slight abuse of notation, we denote by $T_{\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}}}$ (k, z, r) the associated transcript of this execution, which is the concatenation of all inputs to $\mathcal{Z}$, all outputs from $\mathcal{Z}$, and all messages exchanged via the communication tapes of the ITIs (also called the joint view). The distribution $T_{\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}}(k, z)$ and ensemble $T_{\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}}$ are defined analogously to above.

2.1.3. *Ideal-World Process*

Security of protocols is defined via comparing the real-world execution with an ideal-world process that solves the task in an idealistic way. More formally, the ideal process is formulated with respect to an ITM $\mathcal{F}$ which is called an ideal functionality. In the ideal process, the environment $\mathcal{Z}$ interacts with $\mathcal{F}$, an ideal-world adversary (often called the simulator) $\mathcal{S}$ and a set of trivial, i.e., dummy ITMs representing the protocol machines. The dummy ITMs behave as follows: whenever activated with a request x, they forward the request x to $\mathcal{F}$ and output toward $\mathcal{Z}$ whatever they receive in return. $\mathcal{F}$ thereby specify all outputs generated for each party, and the amount of information the ideal-world adversary learn and what its active influence is via its interaction with $\mathcal{F}$. By definition of the corruption mechanism in UC, corruption of parties happens via special corruption messages on the backdoor tape of the ideal functionality (and the party ids pid of all corrupted (dummy) parties can be learned by the environment). We note that an ideal functionality itself, represented as an ITI during the protocol execution, cannot be corrupted by definition.

Based on the above definitions, the ideal-world process proceeds as the real process. It is essentially the real-world process where the ITIs running the protocol are replaced by the dummy ITIs interacting with $\mathcal{F}$ (and only one challenge session ever exists). In this interaction, the same constraints and activation sequence restrictions are enforced by the UC control function. For further details, we refer to [14].

We denote the output of this ideal-world process by $\text{EXEC}_{\mathcal{F},\mathcal{A},\mathcal{Z}}(k, z, r)$ where the inputs are as in the real-world process. Let $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z)$ denote the random variable obtained by choosing the randomness r uniformly at random and evaluating $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ (k, z, r). Let $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}(k, z)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$. The transcript is defined analogously as in the real-world process and denoted $T_{\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}}(k, z, r)$.

2.1.4. *Hybrid Worlds*

To model setup, the UC framework knows so-called hybrid worlds. We discuss two important cases of hybrid worlds that differ in whether the setup, typically called the hybrid functionality, is available only to an instance of a protocol session (standard), or to multiple protocol sessions at the same time (shared). Note that a protocol can assume several setup functionalities of both types.

Standard (local) setup. A standard setup is modeled in UC as an ideal-functionality available in a real-world protocol execution, i.e., as an incorruptible ITI $\mathcal{F}$ that provides certain ideal guarantees to this protocol session. We consider here the natural case that standard setups are available in real-world processes only (note that while the following conventions can be applied to ideal-world-processes as well, it still seems like an uninteresting case to consider standard setups in ideal-processes). So, formally, the $\mathcal{F}$-hybrid-world process is identical to the real-world process with the following additions: The parties can interact with (an a priori unbounded number of) instances of $\mathcal{F}$ by standard interaction (sending messages, passing output to them, or receiving input from them). Each copy of $\mathcal{F}$, i.e., each such incorruptible ITI, is identified via a unique session identifier sid chosen by the protocol that passes in put to it. (This in particular implies a unique identity id of this ITI.) It is stressed that by this definition, the environment can only access $\mathcal{F}$ via calls to parties or via the adversary.

Since a protocol makes explicit which local functionalities it assumes we omit an explicit reference in the formal expressions for simplicity. For example, we just write $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ or $T_{\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}}$ to denote the output or the transcript distribution ensembles in such cases.

Shared (global) setup. Sometimes we want to model that a certain hybrid functionality, say $\mathcal{G}$, to be declared as shared (often also denoted to as global setup). This breaks the isolated-session idea (subroutine respecting property) of standard UC and allows sessions to share state, or more generally a functionality, with other sessions, resp. with the environment. While the exact dynamics of such a global model is beyond the scope of this introduction, the basic idea how to model such global setup in UC is relatively easy and does not need a separate model such as the one originally proposed in [16]. Instead, we follow the modeling in [4]: a generic mapping (or operator akin to the UC composition operator) is defined that takes a protocol π and the to-be-treated-as-global functionality $\mathcal{G}$ (more precisely, the ideal protocol associated with $\mathcal{G}$ as described above) and transforms the protocol (oblivious to the protocol and the functionality) into a standard UC protocol $\mathsf{M}[\pi, \mathcal{G}]$. This protocol has the property that it exposes $\mathcal{G}$ to the environment, while the behavior of π in its interaction with other ITIs and the environment remains unchanged. We recall that the identity-bound predicate ξ is a handy tool to model which external identities must be assumed to have access to the shared functionality when proving the challenge protocol secure. Finally, since global setups are always present in the experiments, the ideal process (that is, the ideal world) can be simply expressed as $\mathsf{M}[\mathcal{F}, \mathcal{G}]$, i.e., where the ideal protocol for $\mathcal{F}$ is considered instead of π. We point out that global setups must satisfy a so-called *regularity* condition, which can be achieved trivially by having any party that wants to interact with $\mathcal{G}$ to register first with the functionality. All our ideal functionalities are of this form. If a shared setup $\mathcal{G}$ is available in the real-world or ideal-world processes, we usually make it explicit in the notation such as $\text{EXEC}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}}$ or $\text{EXEC}^{\mathcal{G}}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$, where we understand the normal UC execution with the above transformation.

2.1.5. *Secure Realization and Composition*

In a nutshell, a protocol securely realizes an ideal functionality $\mathcal{F}$ if the real-world process (where the protocol is executed) is indistinguishable from the ideal-world process (relative to $\mathcal{F}$). If the protocol uses setup, we technically consider the hybrid-world processes instead of the plain real-world or ideal-world processes. We directly state the definitions.

Definition 2.1. Let us denote by $\mathcal{X} = \{X(k, z)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$ and $\mathcal{Y} = \{Y(k, z)\}_{k\in\mathbb{N},z\in\{0,1\}^*}$ two distribution ensembles over $\{0, 1\}$. We say that $\mathcal{X}$ and $\mathcal{Y}$ are indistinguishable if for any $c, d \in \mathbb{N}$ there exists a $k_0 \in \mathbb{N}$ such that $|\Pr[X(k, z) = 1] - \Pr[Y(k, z) = 1]| < k^{-c}$ for all $k > k_0$ and all $z \in \bigcup_{\kappa\leq k^d}\{0, 1\}^\kappa$. We use the shorthand notation $\mathcal{X} \approx \mathcal{Y}$ to denote two indistinguishable ensembles.

Definition 2.2. Let $n \in \mathbb{N}$, let $\mathcal{F}$ be an ideal functionality, and let π be a protocol defined for the real-world, and which potentially makes use of some local setup functionality $\mathcal{H}$ and some global setup $\mathcal{G}$. We say that π securely realizes $\mathcal{F}$ (in the presence of these setup functionalities) if for any (efficient) adversary $\mathcal{A}$ there exists an (efficient) ideal-world adversary (the simulator) $\mathcal{S}$ such that for every (efficient) environment $\mathcal{Z}$ it holds that $\text{EXEC}^{\mathcal{G}}_{\pi,\mathcal{A},\mathcal{Z}} \approx \text{EXEC}^{\mathcal{G}}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

In the literature, the above condition is often referred to as π securely realizing functionality $\mathcal{F}$ in the $(\mathcal{G}, \mathcal{H})$-hybrid world, where the type of setup is inferred by the context.

Composition. The notion of secure realization is composable. We do not give a detailed explanation as it is not important to follow the results in this work. In a nutshell, assume first that a protocol π securely realizes $\mathcal{F}$ in the $\mathcal{H}$-hybrid world, where $\mathcal{H}$ denotes a standard (local) setup functionality. Let further ϕ be a protocol that securely realizes $\mathcal{F}$. Then, the protocol π', where each call to $\mathcal{H}$ is replaced by an invocation of protocol ϕ, securely realizes $\mathcal{F}$. We refer the interested reader to [14] for the general formal statement and on the exact definition of π'. Along similar lines, a composition theorem can be proven (following from the standard UC composition theorem) where local hybrid functionalities are replaced by the protocols securely realizing them, all in the presence of shared setups [4]. Finally, we only note in passing that one can also consider replacing shared functionalities (i.e., the global setups) by suitable protocols. This, however, is a very subtle issue for which we refer the interested reader to [9].

2.2. *Large Deviation Bounds*

We use some known results to derive large deviation bounds in our probabilistic arguments. For proofs and further discussions we refer to [21].

Theorem 2.3. (Chernoff bound) *Let $X_1, \ldots, X_T$ be independent random variables with $\mathbb{E}[X_i] = p_i$ and $X_i \in [0, 1]$. Let $X = \sum_{i=1}^{T} X_i$ and $\mu = \sum_{i=1}^{T} p_i = \mathbb{E}[X]$. Then, for all $\Lambda \geq 0$,*

$$\Pr[X \geq (1 + \Lambda)\mu] \leq e^{-\frac{\Lambda^2}{2+\Lambda}\mu} ;$$

$$\Pr[X \leq (1 - \Lambda)\mu] \leq e^{-\frac{\Lambda^2}{2+\Lambda}\mu} .$$

Theorem 2.4. (Azuma's inequality (Azuma; Hoeffding)) *Let $X_0, \ldots, X_n$ be a sequence of real-valued random variables so that, for all t, $|X_{t+1} - X_t| \leq c$ for some constant c. If $\mathbb{E}[X_{t+1} \mid X_0, \ldots, X_t] \leq X_t$ for all t, then for every $\Lambda \geq 0$*

$$\Pr[X_n - X_0 \geq \Lambda] \leq \exp\left(-\frac{\Lambda^2}{2nc^2}\right).$$

Alternatively, if $\mathbb{E}[X_{t+1} \mid X_0, \ldots, X_t] \geq X_t$ for all t, then for every $\Lambda \geq 0$

$$\Pr[X_n - X_0 \leq -\Lambda] \leq \exp\left(-\frac{\Lambda^2}{2nc^2}\right).$$

3. UC Execution Model for Permissionless PoW Blockchains

In this section, we describe our UC model of execution for the Bitcoin protocol. We remark that providing such a formal model of execution forces us to make explicit all the implicit assumptions from previous works. As we lay down the theoretical framework, we will also discuss these assumptions along with their strengths and differences.

Bitcoin miners are formally represented as ITIs, which we refer to for notational convenience by P_i, i.e, $P_i = (\pi, id_i)$, where $id_i = (\mathsf{pid}_i, \mathsf{sid})$ and where π will be the Bitcoin protocol (running in session sid). We refer to P_i as a party for short. The index i is used to distinguish two identifiers, i.e., $P_i \neq P_j$, and otherwise carries no meaning. Parties interact which each other by exchanging messages over an unauthenticated multicast network with eventual delivery (see below) and might make queries to a common random oracle. We will assume a central adversary $\mathcal{A}$ who gets to corrupt miners and might use them to attempt to break the protocol's security. As is common in UC, the resources available to the parties are described as hybrid functionalities (some of which are treated as shared or global as we discuss later). Before we provide the formal specification of such functionalities, we first discuss a delicate issue that relates to the set of parties (ITIs) that might interact with an ideal functionality.

3.1. *Functionalities with Dynamic Party Sets*

In many UC functionalities, the set of parties is defined upon initiation of the functionality and is not subject to change throughout the lifecycle of the execution. Nonetheless, UC does provide support for a completely dynamic generation of ITIs and thus making the set of parties that might interact with the functionality dynamic. This feature is important when modeling the Bitcoin protocol—where miners come and go at will. In this work, we make this explicit by means of the following mechanism: all the functionalities considered here include the instructions below that allow parties to join or leave the set $\mathcal{P}$ that the functionality interacts with, and inform the adversary about the current set of registered parties. Note that making the set of parties dynamic means that the adversary needs to be informed about which parties are currently in the computation so that it can chose how many (and which) parties to corrupt.

- Upon receiving (REGISTER, sid) from some party P[4] (or from $\mathcal{A}$ on behalf of a corrupted P), the functionality sets $\mathcal{P} = \mathcal{P} \cup \{P\}$ and returns (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 $P \in \mathcal{P}$), the functionality sets $\mathcal{P} := \mathcal{P} \setminus \{P\}$ and returns (DE-REGISTER, sid, P) to the caller.
- Upon receiving (GET-REGISTERED, sid) from $\mathcal{A}$, the functionality returns the response (GET-REGISTERED, sid, $\mathcal{P}$) to $\mathcal{A}$.

[4]Recall that P stands for any ITI, and as such the instruction is also defined for functionalities registering to e.g., global setups.

Finally, our functionalities will only interact with those parties that are effectively registered to it. This makes any functionality (hybrid or realized) in this work *regular* which is important for global UC.

3.2. *Modeling Network Assumptions*

In many situations, one cannot tolerate a complete asynchronous network such as the standard UC communication mechanism. For example, we want to argue about liveness properties of blockchains, which requires communication with eventual delivery guarantees as time goes by (see below how we model time). We describe such a network based on ideas from [8,18,34]. In particular, we capture such communication by a network functionality $\mathcal{F}^{\Delta}_{\text{N-MC}}$ that provides each party or miner $P_s \in \mathcal{P}$ the capability to multicast a message. For every newly sent message, say m, the network functionality creates a unique identifier mid for each triple (P_j, P_j, m), where $P_j \in \mathcal{P}$ is a potential receiver. This handle is needed to succinctly refer to a message circulating in the network in a fine-grained manner. The network does not provide any information to any receiver about who else is using it or where a message originates from. More precisely, messages are buffered but the information of who is the sender is never provided to a receiver.

The adversary—who is informed about both the content of the messages and about the handles—is allowed to delay messages by any finite amount, and allowed to deliver them in an arbitrary out-of-order manner. To ensure that the adversary cannot arbitrarily delay the delivery of messages submitted by honest parties, we use the following idea: The network works in a "fetch message" mode, which means that parties need to actively query for the message (for example, a party can query for messages once in a round). If the adversary wishes to delay the delivery of some message with message ID mid, he needs to submit an integer value T_{mid}—the *delay* for the message-in-transmission with identifier mid. For example, if mid refers to the triple (P_s, P_j, m), this will have the effect that only after the next T_{mid} fetch attempts by P_j, P_j will be able to report the receipt of this particular message m. Importantly, the network does not accept more than Δ accumulative delay for any mid. To allow the adversary freedom in scheduling the delivery of messages, we allow him to input delays more than once, which are added to the current delay amount. If the adversary wants to deliver the message in the next activation, all he needs to do is submit a negative delay. Furthermore, we allow the adversary to schedule more than one messages to be delivered in the same "fetch" command. Finally, to ensure that the adversary is able to re-order such batches of messages arbitrarily, we allow $\mathcal{A}$ to send special (SWAP, mid, mid') commands that have as an effect to change the order of the corresponding messages. Last but not least, the adversary is further allowed to do partial and inconsistent multicasts, i.e., where different messages are sent to different parties. This is the main difference of such a multicast network from a broadcast network. The description appears in Fig. 1.

3.2.1. *Multicast from Unicast*

The above multicast functionality is an ideal abstraction of a large network, where we idealize the network delay and network topology. While the network delay is an explicit parameter in the analysis (and could be estimated from real-world deployed networks),

Functionality $\mathcal{F}_{\text{N-MC}}^{\Delta}$

Initialization:

The functionality initializes the party set $\mathcal{P} \leftarrow \emptyset$ and a list (of messages) $\vec{M} \leftarrow []$.

Registrations:

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

Network Capabilities:

- Upon receiving $(\text{MULTICAST}, \text{sid}, m)$ from some $P_s \in \mathcal{P}$ (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 $D_{\mathsf{mid}_1} := D^{MAX}_{\mathsf{mid}_1} \ldots := D_{\mathsf{mid}_n} := D^{MAX}_{\mathsf{mid}_n} := 1$,
  3. Set $\vec{M} := \vec{M} || (m, \mathsf{mid}_1, D_{\mathsf{mid}_1}, P_1) || \ldots || (m, \mathsf{mid}_n, D_{\mathsf{mid}_n}, P_n)$,
  4. Send $(\text{MULTICAST}, \text{sid}, m, P_s, (P_1, \mathsf{mid}_1), \ldots, (P_n, \mathsf{mid}_n))$ to the adversary.
- Upon receiving $(\text{FETCH}, \text{sid})$ 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 $(\text{MULTICAST}, \text{sid}, (m_{i_1}, P_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell})$ from the adversary with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{P}$, do:
  1. Choose ℓ new unique message-IDs $\mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_\ell}$,
  2. initialize ℓ new variables $D_{\mathsf{mid}_{i_1}} := D^{MAX}_{\mathsf{mid}_{i_1}} := \ldots := D_{\mathsf{mid}_{i_\ell}} := D^{MAX}_{\mathsf{mid}_{i_\ell}} := 1$,
  3. set $\vec{M} := \vec{M} || (m_{i_1}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}, P_{i_1}) || \ldots || (m_{i_\ell}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_\ell}}, P_{i_\ell})$,
  4. send $(\text{MULTICAST}, \text{sid}, (m_{i_1}, P_{i_1}, \mathsf{mid}_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell}, \mathsf{mid}_{i_\ell})$ to the adversary.
- Upon receiving $(\text{DELAYS}, \text{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ from the adversary do the following for each pair $(T_{\mathsf{mid}_{i_j}}, \mathsf{mid}_{i_j})$: If $D^{MAX}_{\mathsf{mid}_{i_j}} + T_{\mathsf{mid}_{i_j}} \leq \Delta$ 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 $(\text{SWAP}, \text{sid}, \mathsf{mid}, \mathsf{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \mathsf{mid}, D_{\mathsf{mid}}, \cdot)$ and $(m, \mathsf{mid}', D_{\mathsf{mid}'}, \cdot)$ in $\vec{M}$. Return $(\text{SWAP}, \text{sid})$ to the adversary.
- Upon receiving $(\text{GET-REGISTERED}, \text{sid})$ from $\mathcal{A}$, the functionality returns the response $(\text{GET-REGISTERED}, \text{sid}, \mathcal{P})$ to $\mathcal{A}$.

Fig. 1. The network functionality with eventual delivery guarantees. Note that for a list $\vec{M}$ we denote by the symbol $||$ the operation which appends a new element to $\vec{M}$.

the topology does not appear in any way. Hence, a natural question is how to get the above multicast network from simpler channels. Note that in Bitcoin, parties/miners communicate over an incomplete network and a standard diffusion mechanism is employed: The sender sends the message it wishes to multicast to all its neighbors who check that a message with the same content was not received before, and if this is the case forward it to their neighbors, who then do the same check, and so on. In fact, a multicast network can be built from unicast channels. That is, one essentially assumes for each miner $P_R \in \mathcal{P}$ a channel functionality $\mathcal{F}_{\text{U-CH}}^{\Delta, P_R}$—which is parameterized by a receiver P_R and an upper bound on the delay Δ—to which any other party $P_i \in \mathcal{P}$ can connect and input messages to be delivered to P_R. A miner connecting to the unicast channel with receiver P_R models the real-world process of looking up P_R (e.g., a public node in the network) and using this party to disseminate future messages. The unicast channel should have some similar properties as the above network, namely:

- They guarantee (reliable) delivery of messages within a delay parameter but are otherwise specified to be of asynchronous nature (see below) and hence no protocol can rely on timings regarding the delivery of messages. The adversary might delay any message sent through such a channel, but at most by Δ. In particular, the adversary cannot block messages. However, he can induce an arbitrary order on the messages sent to some party.
- The receiver gets no information other than the messages themselves. In particular, a receiver cannot link a message to its sender nor can he observe whether or not two messages were sent from the same sender.
- The channel offers no privacy guarantees. The adversary is given read access to all messages sent on the network.

In Appendix A, we provide this channel functionality for completeness and explain how a simple round-based diffusion mechanism can be used to implement a multicast mechanism from unicast channels as long as the corresponding network among honest parties stays strongly connected—where a network graph is called strongly connected if there is a directed path between any two nodes in the network where the unicast channels are the directed edges from senders to receivers.

3.3. *Modeling Time and Clock-Dependent Protocol Execution*

Katz et al. [34] proposed a methodology for casting synchronous protocols in UC by assuming they have access to an ideal functionality $\mathcal{G}_{\text{CLOCK}}$, *the clock*, that allows parties to ensure that they proceed in synchronized rounds. Informally, the idea is that the clock keeps track of a round variable whose value the parties can request by sending it (CLOCK-READ, sid_C). This value is updated only once all honest parties sent the clock a (CLOCK-UPDATE, sid_C) command. We use a variant of their clock as a global setup in this work. The description is given in Fig. 2, where we also make explicit the behavior of the clock-update upon corruption.

Given a clock, the authors of [34] describe how synchronous protocols can maintain their necessary round structure in UC: For every round ρ each party first executes all its round-ρ instructions and then sends the clock a CLOCK-UPDATE command. Subsequently, whenever activated, it sends the clock a CLOCK-READ command and does not advance to round $\rho + 1$ before it sees the clocks variable being updated. This ensures that no honest party will start round $\rho + 1$ before every honest party has completed round ρ.

Idealized progression of time. We know from [34] that if we want to capture the ideal guarantee of eventual-delivery, or more generally speaking, idealized progression of time, an ideal functionality needs to keep track of the number of activations that an honest party gets—so that it knows when to enforce progress in the time-domain. As a general principle, the functionality would then have to issue a clock-update command in the name of a party, once that party is done with its round actions (the overall clock ticks for a session once all honest parties are done with their round actions). We now define a notion in Definition 3.1 that simplifies this bookkeeping, that is, instead of having the

Functionality $\mathcal{G}_{\text{clock}}$

Initialization and state:

The functionality initializes the party set $\mathcal{P} \leftarrow \emptyset$. It further maintains variables d_P for each registered party (see below), and a variable τ_{sid} for each session specified in a registered party.

Registrations:

- Upon receiving $(\text{REGISTER}, \text{sid}_C)$ from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$, $d_P \leftarrow 0$, and if $P = (\cdot, \mathsf{sid}||\cdot)$ specifies a new sid, then initialize $\tau_{\mathsf{sid}} \leftarrow 0$. Return $(\text{REGISTER}, \text{sid}, P)$ to the caller.

- Upon receiving $(\text{DE-REGISTER}, \text{sid}_C)$ 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 $(\text{DE-REGISTER}, \text{sid}, P)$ to the caller.

- Upon receiving $(\text{GET-REGISTERED}, \text{sid}_C)$ from $\mathcal{A}$, the functionality returns the response $(\text{GET-REGISTERED}, \text{sid}, \mathcal{P})$ to $\mathcal{A}$.

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \text{sid}_C)$ from some party $P \in \mathcal{P}$ first verify that the dummy party providing the input encodes P as its PID; otherwise, ignore the request. Set $d_P \leftarrow 1$, execute *Round-Update*, and forward $(\text{CLOCK-UPDATE}, \text{sid}_C, P)$ to $\mathcal{A}$.

- Upon receiving $(\text{CLOCK-READ}, \text{sid}_C)$ from any ITI $P = (\cdot, \mathsf{sid}||\cdot)$, first check that the session identifier sid is a managed session; ignore the request otherwise. Execute *Round-Update* and return $(\text{CLOCK-READ}, \text{sid}_C, \tau_{\mathsf{sid}})$ to the requester

Corruptions:

- Upon receiving $(\text{CORRUPT}, \text{sid}_C, P_i)$ from $\mathcal{A}$ corrupting $P_i \in \mathcal{P}$, mark the party as corrupted and execute *Round-Update*. Return $(\text{CORRUPT}, \text{sid}_C, P_i)$ to $\mathcal{A}$.

Procedure Round-Update: For each managed session sid do: if $d_P = 1$ for all uncorrupted $P = (\cdot, \mathsf{sid}||\cdot) \in \mathcal{P}$, then set $\tau_{\mathsf{sid}} \leftarrow \tau_{\mathsf{sid}} + 1$ and reset $d_P \leftarrow 0$ for all identities $P = (\cdot, \mathsf{sid}||\cdot) \in \mathcal{P}$.

Fig. 2. The clock functionality that we use as a global setup.

functionality manage time-progression per party, it can do it on a more coarse-grained, session level and only contact the clock once per round.

To follow the definition recall the mechanics of activations in UC. In a protocol execution, an ITI gets activated either by receiving an input from the environment, subroutine output from one of its hybrid-functionalities, or an input on the backdoor tape from the adversary. Any activation results in the activated ITI performing some computation on its view of the protocol and its local state and ends with either the party providing output to some of its hybrid functionalities, to the environment, or to the adversary. In either of these cases (formally dubbed external-write requests), the ITI loses the activation.[5]

For any given protocol execution, we define the *honest-input sequence* $\vec{\mathcal{I}}_H$ to consist of all inputs given to a main ITI by the environment and the corruption messages by the adversary (listed in the order that they were given). The inputs are annotated with the identity of the ITI that received the input. For an execution with an environment and adversary, where the honest parties in session sid have received m inputs in total, $\vec{\mathcal{I}}_H$ is a vector of the form $((x_1, id_1), \ldots, (x_m, id_m))$, where x_i is the ith input that was given to machine id_i, or a corruption message with target id_i. We further define the extended *timed honest-input sequence*, denoted as $\vec{\mathcal{I}}_H^T$, to be the honest-input sequence augmented with the respective clock time when an input was given. If the timed honest-

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

input sequence of an execution is $\vec{\mathcal{I}}_H^T = ((x_1, id_1, \tau_1), \ldots, (x_m, id_m, \tau_m))$, this means that for each $i \in [n]$, τ_i is the time of the global clock when input x_i was handed to id_i.

Definition 3.1. A $\mathcal{G}_{\text{CLOCK}}$-hybrid protocol Π has a *predictable synchronization pattern* iff there exist an efficiently computable algorithm $\mathsf{predict\text{-}time}_\Pi(\cdot)$ 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 $i \in [m-1]$:

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

3.3.1. *Using the Clock as a Global Setup/Shared Subroutine*

Treating the clock as a global setup or shared subroutine has the benefit of allowing parties across protocols to have a common denomination of time, and to be able to specify observable time-dependent ideal properties. However, modeling a setup as global also comes with complications [4]: to complete the specification in UC to what extent the usage to coordinate on time across protocols is sound, we have to define the (identity bound) predicate that specifies the applicable context. Recall that this predicate is intended to restrict the set of extended identities that the environment can claim when contacting protocols.[6] We define the following identity bound ξ_{sync} first suggested in [4]: The environment is not able to issue any request to the clock which has as a source ID the ID of a party (i.e., ITI) that already exists in the system, or to spawn any ITI for which it already claimed an external identity before in an interaction with the clock. Furthermore, the environment is not allowed to access the corruption information of the session sid_C from the clock directly for the sake of a well-defined PID-wise corruption model [4].

This identity-bound thus ensures that the clock can be used to model lock-step progression of protocols, and composition is guaranteed in any context that does not trivially break the lock-step execution style. Finally, in order to keep the standard PID-wise corruption model simple, the environment is only allowed to access the actual corruption set through the main session's corruption aggregation machine, not through the shared subroutine's session [4].

3.4. *Modeling Hash Queries*

As usual in cryptographic proofs, the queries to the hash function are modeled by assuming access to a random oracle (functionality) $\mathcal{F}_{\text{RO}}^{\kappa}$. This functionality is specified as follows: upon receiving a query $(\textsc{eval}, \text{sid}, x)$ from a registered party, if x has not been queried before, a value y is chosen uniformly at random from $\{0, 1\}^{\kappa}$ (for security parameter κ) and returned to the party (and the mapping (x, y) is internally stored). If x has been queried before, the corresponding y is returned. The description appears in Fig. 3.

[6] Note that having no bound in place is not an option: if the environment was allowed to issue an external write request with the source-ID corresponding to an active party in the session, then the environment would have the power to simply skip and ignore any party.

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

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).

Registrations:

- Upon receiving $(\textsc{register}, \text{sid})$ from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$ and return $(\textsc{register}, \text{sid}, P)$ to the caller.
- Upon receiving $(\textsc{de-register}, \text{sid})$ 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 $(\textsc{de-register}, \text{sid}, P)$ to the caller.
- Upon receiving $(\textsc{get-registered}, \text{sid})$ from $\mathcal{A}$, the functionality returns the response $(\textsc{get-registered}, \text{sid}, \mathcal{P})$ to $\mathcal{A}$.

RO queries:

- Upon receiving $(\textsc{eval}, \text{sid}, x)$ 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 y uniformly at random from $\{0,1\}^{\kappa}$ and set $H[x] \leftarrow y$.
  2. Return $(\textsc{eval}, \text{sid}, x, H[x])$ to the requestor.

Fig. 3. The random oracle functionality. The functionality makes no changes to the standard corruption mode.

3.5. *Which Setup Functionalities as Shared Subroutines?*

The choice which setup functionalities should be treated as local and which one as shared is an important one and is steered by several considerations. First, from the point of view of an outsider party, the question is how much of the inner workings of a session are indeed relevant to observe. Second, modeling a resource as shared comes with complications and must justify the insights. For example, if the network was modeled as a shared network, then the protocol's effect on the network must be replicated in the ideal world (it does not just exist in the simulator's head). This complication might be justified when one studies the problem of protocols competing for bandwidth. Third, to enable simulation, the simulator needs some edge over the real-world adversary which often comes from the fact that the inner workings of a session are not publicly verifiable and thus the simulator can be in charge of creating for example a local CRS or program a random oracle.

In our work, we have three setups: the network, the clock, and the RO, where only the clock is a shared setup. The network is a local resource to simplify the proofs. Making it global would only make the ideal world more complex without providing more insights. Finally, abstracting hash-queries as calls to a global random oracle (GRO) runs into intrinsic problems in the PoW-setting because of two reasons. First, the model needs a reasonably elegant way to achieve some closure on the amount of work invested into a PoW blockchain, and thus it seems natural to say all work invested in Bitcoin are the RO queries made in that session. Second, at the more technical level, a global random oracle would force the simulator to create blocks that indeed carry sufficient work due to the lack of programmability. Since the simulator needs to also simulate the hash queries of honest parties, this would only be feasible if it had a much larger query budget than the real-world adversary (and not having to do this amount of work is the simulator's edge since it can program the RO). Clearly, one could consider a global random oracle that supports programmability [15], but such a more complicated model does not appear to offer more insights than a local RO, as it basically is a shared subroutine that offers a per-session (and hence a local) advantage to the simulator. Additionally, the programmability feature comes at the price of a technical condition via which an adversary can always make a protocol abort, which is unrealistic.

3.6. *Assumptions as UC-Functionality Wrappers*

In order to prove statements about cryptographic protocols, one often makes assumptions about what the environment (or the adversary) can or cannot do. For example, a standard assumption in [24,44] is that in each round the adversary cannot do more calls to the random oracle than what the honest parties (collectively) can do. This can be captured by assuming a restricted environment and adversary which balances the amount of times that the adversary queries the random oracle. In a property-based treatment such as [24,44], this assumption is typically acceptable. Also in a composable model such restrictions can be formulated. However, restricting the environment is not compliant with a general composition theorem.

Therefore, instead of restricting the class of environments/adversaries, we present an alternative approach to capture the fact that the adversary's access to real-world resource is restricted. The general methodology is to capture restrictions by means of a functionality wrapper that wraps the hybrid resources and enforces the restrictions on the adversary by limiting its access to the resource. Such restrictions can become quite complex and we show concrete examples in Sect. 8 to cast the assumptions and derive the equivalent composable statements.

A toy example. To illustrate the general methodology, consider the example of limiting the rate of RO queries of an adversary over time. We can capture this assumption by means of a functionality wrapper in a $\mathcal{G}_{\text{CLOCK}}$ hybrid world that wraps the RO functionality and enforces a bound on the adversary, for example by assigning to each corrupted party at most q activations per clock-tick for some parameter q. For completeness the wrapped random-oracle functionality $\mathcal{W}^q(\mathcal{F}_{\text{RO}}^{\kappa})$ is given in Fig. 4.

4. The Basic Transaction-Ledger Functionality

The purpose of this section is to describe the basic structure of a ledger functionality $\mathcal{G}_{\text{LEDGER}}$. The presented functionality is very generic in the sense that it is parameterizable by several elements. The idea is that concrete blockchain protocols yield concrete instances of these parameters, while the basic structure, as presented here, remains the same and can be seen as the greatest common divisor of any such blockchain protocol proposal.

4.1. *Introduction and Overview*

Our ledger is parametrized by certain algorithms/predicates that allow us to capture a more general version of a ledger which can be instantiated by various cryptocurrencies. Since our abstraction of the Bitcoin protocol is in the synchronous model of computation

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

Initialization:

The functionality manages the variable $\texttt{counter}$ (initially 0) and the set of corrupted parties $\mathcal{P}'$ in the session. For each party $P \in \mathcal{P}'$ it manages variables $\texttt{count}_P$.

Initially, $\mathcal{P}' = \emptyset$ and $\texttt{counter} = 0$.

General:

- 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 $(\text{EVAL}, \text{sid}, x)$ from $\mathcal{A}$ on behalf of a corrupted party $P \in \mathcal{P}'$, then first execute *Round Reset*. Then, set $\texttt{count}_P \leftarrow \texttt{count}_P + 1$ and only if $\texttt{count}_P \leq q$ forward the request to $\mathcal{F}_{\text{RO}}^{\kappa}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{RO}}^{\kappa}$ 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 $(\text{CORRUPT}, \text{sid}, P)$ from the adversary, set $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{P\}$. If P has already issued $t > 0$ random oracle queries in this round, set $\texttt{count}_P \leftarrow t$. Otherwise set $\texttt{count}_P \leftarrow 0$.

Procedure Round-Reset:
Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and receive $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ from $\mathcal{G}_{\text{CLOCK}}$. If $|\tau - \texttt{counter}| > 0$, then set $\texttt{count}_P \leftarrow 0$ for each participant $P \in \mathcal{P}'$ and set $\texttt{counter} \leftarrow \tau$.

Fig. 4. The wrapped random oracle.

(this is consistent with known approaches in the cryptographic literature), our ledger is also designed for this synchronous model. Nonetheless, several of our modeling choices are made with the foresight of removing or limiting the use of the clock and leaving room for less synchrony.

At a high level, our ledger $\mathcal{G}_{\text{LEDGER}}$ has a similar structure as the ledger proposed in [36]. Concretely, anyone (whether an honest miner or the adversary) might submit a transaction which is validated by means of a predicate $\mathsf{Validate}$, and if it is found valid it is added to a buffer $\texttt{buffer}$. The adversary $\mathcal{A}$ is informed that the transaction was received and is given its contents.[7] Informally, this buffer also contains transactions that, although validated, are not yet deep enough in the blockchain to be considered out-of-reach for an adversary.[8] Periodically, $\mathcal{G}_{\text{LEDGER}}$ fetches some of the transactions in the buffer, and using an algorithm $\mathsf{Blockify}$ creates a block including these transactions and adds this block to its permanent state $\texttt{state}$, which is a data structure that includes the part of the blockchain the adversary can no longer change. This corresponds to the *common prefix* in [24,44]. Any miner or the adversary is allowed to request a read of the contents of the state.

This sketched specification is simple, but in order to have a ledger that can be implemented by existing blockchain protocols, we need to relax this functionality by giving the adversary more power to interfere with it and influence its behavior. Before sketching

[7] This is inevitable since we assume non-private communication, where the adversary sees any message as soon as it is sent, even if the sender and receiver are honest.

[8] E.g., in [36] the adversary is allowed to permute the contents of the buffer.

the necessary relaxations we discuss the need for a new ledger definition and its potential use as a global setup.

Impossibility to realize the ledger of [36]. The main reasons why the ledger functionality in [36] is not realizable by known protocols under reasonable assumptions are as follows: first, their ledger guarantees that parties always obtain the same common state. Even with strong synchrony assumptions, this is not realizable since an adversary, who just mined a new block, is not forced to inform each party instantaneously (or at all) and thus could, for example, make parties observe different lengths of the same prefix. Second, the adversarial influence is restricted to permuting the buffer. This is too optimistic, as in reality the adversary can try to mine a new block and possibly exclude certain transactions. Also, this excludes any possibility to quantify quality. Third, letting the update rate be fixed does not adequately reflect the probabilistic nature of Nakamoto-style blockchain protocols.

On the sound usage of a ledger as a global setup.

As already pointed out in [11], one has to be extra careful when replacing a global setup by its implementation, e.g., in the case of $\mathcal{G}_{\text{LEDGER}}$ by the UC Bitcoin protocol. Indeed, such a replacement does not, in general, preserve a realization proof of some ideal functionality $\mathcal{F}$ that is conducted in ledger-hybrid world where the ledger is treated as a *shared subroutine*, because the simulator in that proof might rely on specific capabilities that are not available any more after replacement (as the global setup is also replaced in the ideal world). A recent follow-up work by Badertscher et al. [9] explores the facets of this question and gives conditions when a replacement is sound.

As this work focuses on the realization of ledger functionalities per se, this complication is not relevant to this work. We know from [9] that the distinction on whether a functionality is "global/shared" or "local" is decision of how the functionality is being *used* by a protocol (which stands in sharp contrast to prior global UC models that assign it a new type, which can be problematic [4]). Therefore, the functionality $\mathcal{G}_{\text{LEDGER}}$ is a standard UC functionality in our realization proofs.

4.2. *The General Ledger Functionality*

We present here the formal description of the ledger functionality. An overview of its parameters and state variables is given in Table 1 and a in-depth explanation follows in the next section.

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

Parameters: Integers `windowSize`, `Delay`; Algorithms Validate, ExtendPolicy, Blockify, predict-time (cf. Table 1).

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 (CLOCK-READ, sid_C) to $\mathcal{G}_{\text{CLOCK}}$ to receive the answer (CLOCK-READ, sid_C, τ) and sets $\tau_L := \tau$ and then proceeds with the remaining actions.

Variables and initialization: The functionality initializes $\texttt{state}, s_{ep}, \texttt{NxtBC}, \vec{\mathcal{I}}_H^T \leftarrow \varepsilon, \texttt{buffer} \leftarrow \emptyset$ as well as party sets $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS} \leftarrow \emptyset$ (cf. Table 1).

Party Management:

- Upon receiving (REGISTER, sid) from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$, initialize $\texttt{pt}_P \leftarrow 1, \texttt{state}_P \leftarrow \varepsilon$, and $\tau_P^{\text{reg}} \leftarrow \tau_L$. If P is an honest party and if $\mathcal{H} = \emptyset$ send (REGISTER, sid_C)

to $\mathcal{G}_{\text{CLOCK}}$. If P is honest then update $\vec{\mathcal{I}}_H^T$ and set $\mathcal{H} \leftarrow \mathcal{H} \cup \{P\}$ and if additionally $\tau_P^{\text{reg}} > 0$ 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 $\mathcal{P} \leftarrow \mathcal{P} \setminus \{P\}$, $\mathcal{H} \leftarrow \mathcal{H} \setminus \{P\}$, and $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \setminus \{P\}$. If $\mathcal{H} = \emptyset$, send (DE-REGISTER, sid$_C$) to $\mathcal{G}_{\text{CLOCK}}$. If P is honest then update $\vec{\mathcal{I}}_H^T$. Return (DE-REGISTER, sid, P) to the caller.

Ledger Operation:

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 $\vec{\mathcal{I}}_H^T \leftarrow \vec{\mathcal{I}}_H^T || (I, P_i, \tau_L)$. If a party P_i gets corrupted, additionally update $\mathcal{H} \leftarrow \mathcal{H} \setminus \{P_i\}$ and $\mathcal{P}_{DS} \leftarrow \mathcal{P}_{DS} \setminus \{P_i\}$.
2. Let $\widehat{\mathcal{P}} := \{P \in \mathcal{P}_{DS} \mid \tau_P^{\text{reg}} < \tau_L - \texttt{Delay}\}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \widehat{\mathcal{P}}$.
3. If $P_i \in \mathcal{H}$ then additionally take the following steps:
   (a) $(\vec{N}, s') \leftarrow \mathsf{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \texttt{state}, \texttt{NxtBC}, \texttt{buffer}; s_{ep})$. Reset $\texttt{NxtBC} \leftarrow \varepsilon$ and store $s_{ep} \leftarrow s'$.
   (b) If $\vec{N} \neq \varepsilon$ then parse $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell)$ and update $\texttt{state} \leftarrow \texttt{state}||\mathsf{Blockify}(\vec{N}_1)|| \ldots ||\mathsf{Blockify}(\vec{N}_\ell)$.
   (c) For each $\texttt{BTX} \in \texttt{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 0$ then $\texttt{buffer} \leftarrow \texttt{buffer} \setminus \{\texttt{BTX}\}$.
   (d) If $\exists P \in \mathcal{H} \setminus \mathcal{P}_{DS}$ s.t. $\texttt{pt}_P \notin [|\texttt{state}| - \texttt{windowSize} + 1, |\texttt{state}|]$, then set $\texttt{pt}_{P_k} \leftarrow |\texttt{state}|$ for all $P_k \in \mathcal{H} \setminus \mathcal{P}_{DS}$.
4. 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:
   - *Submiting a transaction:*
     If $I = (\text{SUBMIT}, \text{sid}, \texttt{tx})$ do the following:
     (a) Choose a unique transaction ID txid and set $\texttt{BTX} \leftarrow (\texttt{tx}, \text{txid}, \tau_L, P_i)$
     (b) If $\mathsf{Validate}(\texttt{BTX}, \texttt{state}, \texttt{buffer}) = 1$, then $\texttt{buffer} \leftarrow \texttt{buffer} \cup \{\texttt{BTX}\}$.
     (c) Output (SUBMIT, $\texttt{BTX}$) to $\mathcal{A}$.
   - *Reading the state:*
     If $I = (\text{READ}, \text{sid})$ then do the following: if $P_i \in \mathcal{H} \setminus \mathcal{P}_{DS}$ then set $\texttt{state}_i := \texttt{state}|_{\min\{\texttt{pt}_i, |\texttt{state}|\}}$. Return (READ, sid, $\texttt{state}_i$) to the caller.
   - *Maintaining the ledger state:*
     If $I = (\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ and $P_i \in \mathcal{H}$ and $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) > \tau_L$ then send (CLOCK-UPDATE, sid$_C$) to $\mathcal{G}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.
5. If the input I is an additional adversarial capability (received on the backdoor tape from $\mathcal{A}$) execute the respective code:
   - *The adversary reading the state:*
     If $I = (\text{READ}, \text{sid})$, then return $(\texttt{state}, \texttt{buffer}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.
   - *The adversary proposing the next block:*
     If $I = (\text{NEXT-BLOCK}, (\text{txid}_1, \ldots, \text{txid}_\ell))$, update $\texttt{NxtBC}$ as follows:
     (a) Set listOfTxid $\leftarrow \epsilon$
     (b) For $i = 1, \ldots, \ell$ do: if there exists a $\texttt{BTX} = (\texttt{tx}, \text{txid}, \text{minerID}, \tau_L, P_i) \in \texttt{buffer}$ with ID txid $=$ txid$_i$ then set listOfTxid := listOfTxid||txid$_i$.
     (c) Finally, set $\texttt{NxtBC} := \texttt{NxtBC}||$listOfTxid and output (NEXT-BLOCK, *ok*) to $\mathcal{A}$.
   - *The adversary setting state-slackness:*
     If $I = (\text{SET-SLACK}, (P_{i_1}, \widehat{\texttt{pt}}_{i_1}), \ldots, (P_{i_\ell}, \widehat{\texttt{pt}}_{i_\ell}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{H}$ then do the following:
     If for all $P_{i_j} \in \mathcal{H} \setminus \mathcal{P}_{DS}$, $j \in [\ell]$: $|\texttt{state}| - \widehat{\texttt{pt}}_{i_j} < \texttt{windowSize}$ and $\widehat{\texttt{pt}}_{i_j} \geq |\texttt{state}_{i_j}|$, then update $\texttt{pt}_{i_1} := \widehat{\texttt{pt}}_{i_1}$ for every $j \in [\ell]$. Return (SET-SLACK, *ok*) to $\mathcal{A}$.
   - *The adversary setting the state for desychronized parties:*
     If $I = (\text{DESYNC-STATE}, (P_{i_1}, \texttt{state}'_{i_1}), \ldots, (P_{i_\ell}, \texttt{state}'_{i_\ell}))$, with $\{P_{i_1}, \ldots, P_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ then set $\texttt{state}_{i_j} := \texttt{state}'_{i_j}$ for each $j \in [\ell]$ and return (DESYNC-STATE, *ok*) to $\mathcal{A}$.
   - *The adversary obtaining the set of registered parties:*
     If $I = (\text{GET-REGISTERED}, \text{sid})$, then return (GET-REGISTERED, sid, $\mathcal{P}$) to $\mathcal{A}$.
   - *The adversary corrupting a party (additional steps to* Item 1*):*
     If $I = (\text{CORRUPT}, \text{sid}, P_i)$ and $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T) > \tau_L$ then send (CLOCK-UPDATE, sid$_C$) to $\mathcal{G}_{\text{CLOCK}}$. Else return I to $\mathcal{A}$.

Table 1. Overview of main ledger elements including its parameters and state variables.

	Description
Ledger parameter	
$\mathsf{Validate}$	Decides on the validity of a transaction with respect to the current state. Used to clean the buffer of transactions
$\mathsf{ExtendPolicy}$	The (stateful) function that specifies the ledger's guarantees in extending the ledger state (e.g., speed, content etc.)
$\mathsf{predict\text{-}time}$	The function to predict the real-world time advancement
$\mathsf{Blockify}$	The function to format the ledger state output
$\texttt{windowSize}$	A positive integer that describes the window size (number of blocks) of the sliding window
$\texttt{Delay}$	A positive integer that describes a general delay parameter for the time it takes for a newly joining (after the onset of the computation) miner to become synchronized
Ledger variables	
$\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$	The party sets and categories: Registered, honest, and honest-but-desynchronized, respectively.
$\vec{\mathcal{I}}_H^T$	The timed honest-input sequence
$\texttt{state}$	The ledger state, i.e., a sequence of blocks containing the content
$\texttt{buffer}$	The buffer of submitted input values
$\texttt{pt}_P, \texttt{state}_P, \tau_P^{\text{reg}}$	The pointer of party P into state $\texttt{state}$. This prefix is denoted $\texttt{state}_P$ for brevity. The time variable τ_P^{reg} records the time when party P registered to the execution most recently
s_{ep}	The state of the extend-policy algorithm
τ_L	The current time as reported by the clock
$\texttt{NxtBC}$	Stores the current adversarial suggestion for extending the ledger state

4.3. *On the Defining Features*

We explain several of the features of the ledger functionality and give an overview of the relevant parameters and state variables in Table 1.

4.3.1. *State-Buffer Validation*

The first relaxation is with respect to the invariant that is enforced by the validation predicate $\mathsf{Validate}$. Concretely, in [36] it is assumed that the validation predicate enforces that the buffer does not include conflicting transactions, i.e., upon receipt of a transaction, $\mathsf{Validate}$ checks that it is not in conflict with the state and the buffer, and if so the transaction is added to the buffer. However, in reality we do not know how to implement such a strong filter, as different miners might be working on different, potentially conflicting sets of transactions.[9] The only time when it becomes clear which of these conflicting transactions will make it into the state is once one of them has been inserted into a block which has made it deep enough into the blockchain (i.e., has become part of $\texttt{state}$). Hence, given that the buffer includes all transactions that might end up in the state, it might at some point include both conflicting transactions.

[9]This will be the case for transactions submitted by the adversary even when signatures are used to authenticate transactions.

To enable us for a provably implementable ledger, in this work we take a different approach. The validate predicate will be less restrictive as to which transactions make it into the buffer. Concretely, at the very least, Validate will enforce the invariant that no single transaction in the buffer contradicts the state `state`, while different transactions in `buffer` might contradict each other. Looking ahead, a stronger version that is achievable by employing digital signatures (presented in Sect. 9) could enforce that no submitted transaction contradicts other submitted transactions. As in [36], whenever a new transaction `tx` is submitted to $\mathcal{G}_{\text{LEDGER}}$, it is passed to Validate which takes as input a transaction and the current state and decides if `tx` should be added to the buffer. Additionally, as `buffer` might include conflicts, whenever a new block is added to the state, the buffer (i.e., every single transaction in `buffer`) is re-validated using Validate and invalid transactions in `buffer` are removed. To allow for this re-validation to be generic, transactions that are added to the buffer are accompanied by certain metadata, i.e., the identity of the submitter, a unique transaction ID txid,[10] or the time τ when `tx` was received.

4.3.2. *State Update Policy and Security Guarantees*

The second relaxation is with respect to the rate and the form and/or origin of transactions that make it into a block. Concretely, instead of assuming that the state is extended in fixed time intervals, we allow the adversary to define when this update occurs. This is done by allowing the adversary, at any point, to propose what we refer to as the next-block candidate `NxtBC`. This is a data structure containing the contents of the next block that $\mathcal{A}$ wants to have inserted into the state. Leaving `NxtBC` empty can be interpreted as the adversary signaling that it does not want the state to be updated in the current clock tick.

Of course allowing the adversary to always decide what makes it into the state `state`, or if anything ever does, yields a very weak ledger. Intuitively, this would be a ledger that only guarantees the common prefix property [24] but no liveness or chain quality. Therefore, to enable us to capture also stronger properties of blockchain protocols we parameterize the ledger by an algorithm ExtendPolicy that, informally, enforces a state-update policy restricting the freedom of the adversary to choose the next block and implementing an appropriate compliance-enforcing mechanism in case the adversary does not follow the policy. This enforcing mechanism simply returns a default policy-complying block using the current contents of the buffer. We point out that a good simulator for realizing the ledger will avoid triggering this compliance-enforcing mechanism, as this could result in an uncontrolled update of the state which would yield a potential distinguishing advantage. In other words, a good simulator, i.e., ideal-world adversary, always complies with the policy.

In a nutshell, ExtendPolicy is a possibly stateful algorithm that takes the ledger state, the current contents of the buffer `buffer`, along with the adversary's recommendation `NxtBC`. The output of ExtendPolicy is a vector including the blocks to be appended to the state (where again, ExtendPolicy outputting an empty vector is a signal to not

[10] In Bitcoin, the value txid would be the hash-pointer corresponding to this transaction. Note that the generic ledger can capture explicit guarantees on the ability or disability to link transactions, as this crucially depends on the concrete choice of an ID mechanism.

extend), together with its update internal state. To ensure that ExtendPolicy can also enforce properties that depend on who inserted how many (or which) blocks into the state—e.g., the so-called *chain quality* property from [24]—we also pass to it the timed honest-input sequence $\vec{\mathcal{I}}_H^T$ (cf. Sect. 3).

Some examples of how ExtendPolicy allows us to define ways that the protocol might restrict the adversary's interference in the state-update include the following properties from [24]:

- *Liveness* corresponds to ExtendPolicy enforcing the following policy: If the state has not been extended for more that a certain number of rounds and the simulator keeps recommending an empty `NxtBC`, ExtendPolicy can choose some of the transactions in the buffer (e.g., those that have been in the buffer for a long time) and add them to the next block. Note that a good simulator or ideal-world adversary will never allow for this automatic update to happen and will make sure that he keeps the state extend rate within the right amount.
- *Chain quality* corresponds to ExtendPolicy enforcing the following policy: Every block proposal made by the simulator is examined as to whether it is maximally filled with valid transactions. Such blocks must appear frequently. If this is not the case, the ledger will define and add a default block to the state. We point out that unlike the original chain-quality property from [24], this policy does not enforce which miner should receive the reward for honest blocks and it is up to the simulator to do so (via the so-called coinbase transaction).[11]

We note that ExtendPolicy is a general concept capable of formulating various properties of blockchain protocols. For example, we can capture that honest (and non-conflicting) transactions eventually make it into the state. Another property could be to formalize that transactions with higher rewards make it into a block faster than others (which we do not consider in this work).

In Sect. 6, we provide one possible specification of ExtendPolicy that can be guaranteed for the UC Bitcoin protocol.

4.3.3. *Output Slackness and Sliding Window of State Blocks*

The common prefix property guarantees that blocks that are sufficiently deep in the blockchain of an honest miner will eventually be included in the blockchain of every honest miner. Stated differently, if an honest miner receives as output from the ledger a state `state`, every honest miner will eventually receive `state` as its output. However, in reality we cannot guarantee that at any given point in time all honest miners see exactly the same blockchain length; this is especially the case when network delays are incorporated into the model, but it is also true in the zero-delay model of [24]. Thus, it is unclear how `state` can be defined so that at any point all parties have the same view on it.

[11] Note that while good blocks are created and circulated in the network by an honest miner, this does not mean that this miner is still honest when the block makes it into the ledger state unless one considers static corruptions only (in which case one could more simply argue about the fraction of honest originators in the ledger state). To make this difference is crucial to explicitly see the impact due to adaptive corruptions.

Therefore, to have a ledger implementable by standard assumptions we make the following relaxation: We interpret $\texttt{state}$ as the view of the state of the miner with the longest blockchain. And we allow the adversary to define for every honest miner P_i a subchain $\texttt{state}_i$ of $\texttt{state}$ of length $|\texttt{state}_i| = \texttt{pt}_i$ that corresponds to what P_i gets as a response when he reads the state of the ledger (formally, the adversary can fix a pointer $\texttt{pt}_i$). For convenience, we denote by $\texttt{state}|_{\texttt{pt}_i}$ the subchain of $\texttt{state}$ that finishes in the $\texttt{pt}_i$th block. Once again, to avoid over-relaxing the functionality to an unuseful setup, our ledger allows the adversary to only move the pointers forward and it forbids the adversary to define pointers for honest miners that are too far apart, i.e., more than $\texttt{windowSize}$ state blocks. The parameter $\texttt{windowSize} \in \mathbb{N}$ denotes a core parameter of the ledger. In particular, the parameter $\texttt{windowSize}$ reflects the similarity of the blockchain to the dynamics of a so-called *sliding window*, where the window of size $\texttt{windowSize}$ contains the possible views of honest miners onto $\texttt{state}$ and where the head of the window advances with the head of the $\texttt{state}$. In addition, it is convenient to express security properties of concrete blockchain protocols, including the properties discussed above, as assertions that hold within such a sliding window (for any point in time).

4.3.4. *Synchrony Aspects and De-Synchronized Parties*

In order to keep the ideal execution indistinguishable from the real execution, the progression of time must be the same. Since the protocol advances the clock as an effect of executing the protocol, the ledger needs to ensure this in the ideal world (note that we model that the protocol can make advancement without the adversary being in the loop to capture liveness). To simplify clock-progression management, recall Definition 3.1, where we introduce $\mathsf{predict\text{-}time}(\vec{\mathcal{I}}_H^T)$, to enable a modular view how the clock proceeds of an entire session. Thus, instead of managing each party individually, the ledger simply registers itself to the clock, records the timed honest-input sequence $\vec{\mathcal{I}}_H^T$ of its session, and signals the clock when the session is ready to advance to the next round. Observe that the ledger can infer all protocol-relevant inputs to honest parties and thus keep track of the honest inputs sequence $\vec{\mathcal{I}}_H^T$. As the other functions explained above, we make the function $\mathsf{predict\text{-}time}$ a parameter of the (general) ledger functionality that needs to be instantiated when realizing a specific ledger such as the Bitcoin ledger (which is the topic of Sect. 6).

A final observation is with respect to guarantees that the protocol (and therefore also the ledger) can give to recently registered honest parties. We introduce an additional party set, $\mathcal{P}_{DS}$, which consist of honest parties for which we are not able to give the full guarantees yet because they are *de-synchronized*. The ledger parameter $\texttt{Delay}$ describes the time (in number of clock ticks) it takes for a newly joining party, that joins later than at the onset of the execution, to become fully synchronized.

To provide more intuition why we need such a set, consider the following scenario: An honest party registers as miner in round r and waits to receive from honest parties the transactions to mine and the current longest blockchain. In Bitcoin, upon joining, the miner sends out a special request on the network—we denote this here as a special NEW-MINER-message—and as soon as any party receives it, it responds with the set of transactions and longest blockchain it knows. Due to the network delay, it can take a full round-trip time before the longest chain arrives to the newcomer. However, because we do not make any assumption on honest parties knowing Δ they start mining as soon as they see network traffic. But now the adversary, in the worst case, can make these parties mine on any block he wants and have them accept any valid chain he wants as the current state while they wait for the network's response (by maximally delaying everything sent to these parties by other honest parties, and instead immediately deliver what he wants them to work on). However, after a constant number of rounds, this effect will be resolved and the parties will be synchronized with the longest chain.

5. Bitcoin as a UC Protocol

5.1. *Basics*

A *blockchain* $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ is a (finite) sequence of blocks where a *block* $\mathbf{B}_i = \langle \mathtt{s}_i, \mathtt{st}_i, \mathtt{n}_i \rangle$ is a triple consisting of the *pointer* $\mathtt{s}_i$ (identifying the predecessor block via its hash), the *state block* $\mathtt{st}_i$, and the *nonce* $\mathtt{n}_i$. The *head* of chain $\mathcal{C}$ is the block $\mathtt{head}(\mathcal{C}) := \mathbf{B}_n$ and the *length* $\mathtt{length}(\mathcal{C})$ of the chain is the number of blocks, i.e., $\mathtt{length}(\mathcal{C}) = n$. The chain $\mathcal{C}^{\lceil k}$ is the (potentially empty) sequence of the first $\mathtt{length}(\mathcal{C}) - k$ blocks of $\mathcal{C}$. A special block is the *genesis block* $\mathbf{G} = \langle \perp, \mathtt{gen}, \perp \rangle$ which contains the genesis state $\mathtt{gen} := \varepsilon$ and, as we will see later, is required to be the first block in the sequence.

The *state* $\vec{\mathtt{st}}$ *encoded* in $\mathcal{C}$ is defined as a sequence of the corresponding state blocks, i.e., $\vec{\mathtt{st}} := \mathtt{st}_1 || \ldots || \mathtt{st}_n$. In other words, one should think of the blockchain $\mathcal{C}$ as an encoding of its underlying state $\vec{\mathtt{st}}$; such an encoding might, e.g., organize $\mathcal{C}$ is an efficient searchable data structure as is the case in the Bitcoin protocol where a blockchain is a linked list implemented with hash-pointers. In the protocol, the blockchain is the data structure storing a sequence of entries, often referred to as transactions. Furthermore, as in [36], in order to capture a range of blockchains with syntactically different state encoding, we assume a generic algorithm $\mathsf{blockify}_{\text{B}}$ to map a vector of transactions into a state block. Thus, each block $\mathtt{st} \in \vec{\mathtt{st}}$ (except the genesis state) of the state encoded in the blockchain has the form $\mathtt{st} = \mathsf{blockify}_{\text{B}}(\vec{N})$ where $\vec{N}$ is a vector of elements that we simply call transactions, although our treatment is generic and does not fix the type of data the ledger is carrying.

5.1.1. *Validity and Longest Valid Chains*

The validity of a blockchain $\mathcal{C}$ depends on two aspects: *chain-level* validity, also referred to as syntactic validity, and a *state-level* validity also referred to as semantic validity.

Syntactic validity. This is defined with respect to a difficulty parameter $\mathtt{D} \in [2^\kappa]$, where κ is the security parameter, and a given hash function $H : \{0, 1\}^* \to \{0, 1\}^\kappa$; it requires that, for each $i > 1$, the value $\mathtt{s}_i$ contained in $\mathbf{B}_i$ satisfies $\mathtt{s}_i = H[\mathbf{B}_{i-1}]$ and that $H[\mathbf{B}_i] < \mathtt{D}$ holds (for non-genesis blocks), where the output of the hash-function is understood as an integer. Note that for notational simplicity, we omit the hash function as an explicit superscript.

Algorithm $\mathsf{validStruct}^{\mathtt{D}}_{\text{B}}(\mathcal{C})$

res $\leftarrow$ true
if $(\mathtt{length}(\mathcal{C}) = 0)$ **or** $(H[\mathtt{head}(\mathcal{C})] \geq \mathtt{D})$ **then**
  res $\leftarrow$ false
else if $\mathtt{length}(\mathcal{C}) = 1$ **then**
  res $\leftarrow (\mathcal{C} = \mathbf{G})$
else ▷ In this case, the chain is non-trivial and the most recent block is a valid proof-of-work.
  $\mathcal{C}' \leftarrow \mathcal{C}$
  $\langle \mathtt{s}', \cdot, \cdot \rangle \leftarrow \mathtt{head}(\mathcal{C}')$
  repeat
    $\mathcal{C}' \leftarrow \mathcal{C}'^{\lceil 1}$ ▷ Chop off the head of $\mathcal{C}'$.
    $\mathbf{B} := \langle \mathtt{s}, \mathtt{st}, \mathtt{n} \rangle \leftarrow \mathtt{head}(\mathcal{C}')$
    if $(H[\mathbf{B}] \neq \mathtt{s}')$ **or** $(\mathtt{length}(\mathcal{C}') > 1$ **and** $H[\mathtt{head}(\mathcal{C})] \geq \mathtt{D})$ **or** $(\mathtt{length}(\mathcal{C}') = 1$ **and** $\mathbf{B} \neq \mathbf{G})$ **then**
      res $\leftarrow$ false
    else
      $\mathtt{s}' \leftarrow \mathtt{s}$
  until res $=$ false **or** $\mathtt{length}(\mathcal{C}') = 1$
return res

Semantic validity. This is defined on the state $\vec{\mathtt{st}}$ encoded in the blockchain $\mathcal{C}$ and specifies whether this content is valid (which might depend on a particular application). We go with a generic semantic validity check of the blockchain defined by algorithm $\mathsf{isvalidstate}_{\text{B}}$ below. We assume a generic validation predicate for single transactions that we refer to by $\mathsf{ValidTx}_{\text{B}}$ (and which is an algorithm that takes a state and the transaction that is being validated as inputs and outputs a bit). For the sake of generality, this validity predicate is completely generic and looking ahead, our main theorem holds for any choice of this predicate, whenever the ledger parameter $\mathsf{Validate}$ is chosen accordingly as we show in Sect. 6.

The pseudo-code of the algorithm $\mathsf{isvalidstate}_{\text{B}}$ which builds upon $\mathsf{ValidTx}_{\text{B}}$ is provided below. In a nutshell, the algorithm checks that a given blockchain state can be built in an iterative manner, such that each contained transaction is considered valid according to $\mathsf{ValidTx}_{\text{B}}$ upon insertion. It further ensures that the state starts with the genesis state and that state blocks contain a special *coin-base* transaction $\mathtt{tx}^{\text{coin-base}}_{\text{minerID}}$ which assigns them to a miner.

Algorithm $\mathsf{isvalidstate}_{\text{B}}(\vec{\mathtt{st}})$

Let $\vec{\mathtt{st}} := \mathtt{st}_1 || \ldots || \mathtt{st}_n$
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 $\mathtt{st}_i$
$\vec{\mathtt{st}}' \leftarrow \mathtt{gen}$ ▷ Initialize the genesis state
for $i = 1$ to n **do**
  if the first transaction in $\vec{\mathtt{tx}}_i$ is not a coin-base transaction **return** false
  $\vec{N}_i \leftarrow \mathtt{tx}_{i,1}$
  for $j = 2$ to $|\vec{\mathtt{tx}}_i|$ **do**
    $\mathtt{st} \leftarrow \mathsf{blockify}_{\text{B}}(\vec{N}_i)$
    if $\mathsf{ValidTx}_{\text{B}}(\mathtt{tx}_{i,j}, \vec{\mathtt{st}}' || \mathtt{st}) = 0$ **return** false
    $\vec{N}_i \leftarrow \vec{N}_i || \mathtt{tx}_{i,j}$
  $\vec{\mathtt{st}}' \leftarrow \vec{\mathtt{st}}' || \mathtt{st}_i$
return true

Definition 5.1. A chain $\mathcal{C}$ is valid if it satisfies syntactic and semantic validity, i.e., if, for the chain and its encoded state $\vec{\mathtt{st}}$, the following predicate evaluates to true:

$$\mathsf{isvalidchain}^{\mathrm{D}}_{\text{฿}}(\mathcal{C}) := \mathsf{validStruct}^{\mathrm{D}}_{\text{฿}}(\mathcal{C}) \wedge \mathsf{isvalidstate}_{\text{฿}}(\vec{\mathtt{st}}).$$

Longest valid chain. In the Bitcoin protocol, the notion of the *longest valid chain* is very crucial. The reason is that the party defines the ledger state at a certain time as a prefix of the state encoded in the longest valid chain it knows at that time. We stick to the nomenclature of [24] and call the function $\mathsf{maxvalid}_{\text{฿}}(\mathcal{C}_1, \ldots, \mathcal{C}_k)$.

Algorithm $\mathsf{maxvalid}^{\mathrm{D}}_{\text{฿}}(\mathcal{C}_1, \ldots, \mathcal{C}_k)$

$\mathcal{C}_{temp} \leftarrow \varepsilon$
for $i = 1$ to k **do**
  if $\mathsf{isvalidchain}^{\mathrm{D}}_{\text{฿}}(\mathcal{C}_i)$ **and** $(\mathtt{length}(\mathcal{C}_i) > \mathtt{length}(\mathcal{C}_{temp}))$ **then**
    $\mathcal{C}_{temp} \leftarrow \mathcal{C}_i$
return $\mathcal{C}_{temp}$

5.1.2. *Extending Chains and Proofs-of-Work*

A core step in Bitcoin is to extend a given chain $\mathcal{C}$ by a new block $\mathbf{B}$ (with certain state content) to yield a longer chain $\mathcal{C}||\mathbf{B}$. As presented in [24] this can be captured by an algorithm $\mathsf{extendchain}_{\mathrm{D}}(\cdot)$ that takes a chain $\mathcal{C}$, a state block $\mathtt{st}$ and the number of attempts q as inputs. It tries to find a proof-of-work which allows to extend the $\mathcal{C}$ by a block which encodes $\mathtt{st}$.

Algorithm $\mathsf{extendchain}_{\mathrm{D}}(\mathcal{C}, \mathtt{st}, q)$

Input: Chain $\mathcal{C}$ is valid with state $\vec{\mathtt{st}}$. The state $\vec{\mathtt{st}}||\mathtt{st}$ is valid.
  Set $\mathbf{B} \leftarrow \perp$
  $\mathtt{s} \leftarrow H[\mathtt{head}(\mathcal{C})]$ ▷ Compute the pointer $\mathtt{s}$ of the new block
  for $i \in \{1, \ldots, q\}$ **do**
    Choose nonce $\mathtt{n}$ uniformly at random from $\{0, 1\}^{\kappa}$ and set $\mathbf{B} \leftarrow \langle \mathtt{s}, \mathtt{st}, \mathtt{n}\rangle$.
    if $H[\mathbf{B}] < \mathrm{D}$ **then**
      break
  if $\mathbf{B} \neq \perp$ **then**
    $\mathcal{C} \leftarrow \mathcal{C}||\mathbf{B}$
  return $\mathcal{C}$

5.2. *Overview and Modeling Decisions*

In Bitcoin, each party maintains a local blockchain which initially consists of the genesis block. The chains of honest parties might differ (but as we will prove, it will have a common prefix which will define the ledger state). New transactions are added in a 'mining process'. First, a party collects valid transactions (according to $\mathsf{ValidTx}_{\text{฿}}$) and creates a new state block $\mathtt{st}$ using $\mathsf{blockify}_{\text{฿}}$. Next, the party attempts to mine a new block by solving a puzzle (and hence finding a proof-of-work) which upon success could then be validly added to their local blockchain. After each mining attempt parties will multicast their current chain. A party will replace its local chain if it obtains or receives a longer valid chain. When queried to output the state of the ledger, a party reports a prefix of the state encoded in its longest valid chain—obtained by ignoring (or chopping-off) the most recent T blocks (a party outputs ε if the state has less than T blocks). This behavior will ensure that all honest parties output a consistent ledger state. T is a crucial parameter of the Bitcoin protocol and typically, the guarantees of the security statements depend on T (and in addition on the usual security parameter κ).

5.2.1. *The Round Structure*

As already mentioned in the introduction, we model Bitcoin as a lock-step (sometimes dubbed semi-) synchronous protocol: The protocol can proceed in rounds—enabled by having access to a global synchronization clock $\mathcal{G}_{\text{CLOCK}}$—but is not aware of the actual delay of the network. In each round, two logical tasks have to be executed: an *updating* or information-fetching step (where new messages from the network are processed) and a *working* or mining-step, where each party tries to extend its local chain.

To simplify the UC activation handling in the analysis, we divide each logical round into two sub-rounds (where each sub-round corresponds to a logical task; see below for more details). This means that each logical round correspond to two actual clock-ticks (also known as mini-rounds in the MPC literature). We say that a protocol is in round r if the current time of the clock is $\tau \in \{2r, 2r + 1\}$.

Having two clock-ticks per round is a standard way to model in synchronous UC that messages (e.g., a block) sent within a round are delivered at the beginning of the next round. In our case, each round is divided into two mini-rounds, where each mini-round corresponds to a clock tick. We treat the first mini-round as the *updating mini-round* (fetch messages from the network to obtain messages sent previous rounds) and the second mini-round as the *working mini-round* (solving the puzzle and multicasting solutions).

5.2.2. *Handling Interrupts*

A protocol command might consists of a sequence of operations. However, certain operations, such as sending a message to another party, result in the protocol machine losing the activation token. We briefly describe a standard way to formalize that a party that loses an activation in the middle of a multi-step command is able to resume and complete the command following the implicit proposal of [34]. Their mechanism can be made explicit by introducing an anchor a that stores a pointer to the current operation; the protocol associates each anchor with such a multiple command and an input I, so that when such an input is received it directly jumps to the stored anchor, executes the next operation(s) and updates (increases) the anchor before releasing the activation. We refer to such an execution as being *I-interruptible*.

As an example, consider a protocol that requires that upon receiving input I, the party should run a command that consists of m steps Step 1, Step 2, ..., Step m, but some of these steps might result in the party losing its activation. Running this command in an *I-interruptible* manner means executing the following code: Upon receiving input I if $a < m$ go to Step a and increase $a = a + 1$ before executing the first operation that releases the activation; otherwise go to Step 1 and set $a = 2$ before executing any operation that releases the activation.

5.3. *The Formal Protocol Description*

We can now formally define our blockchain protocol $\mathsf{Ledger\text{-}Protocol}_{q,\mathtt{D},T}$ (we usually omit the parameters when clear from the context). The protocol allows an arbitrary number of parties/miners to communicate by means of a multicast network $\mathcal{F}^{\Delta}_{\text{N-MC}}$. Note that this means that the adversary can send different messages to different parties. New miners might dynamically join or leave the protocol by means of the registration/de-registration commands: when they join they register with all associated functionalities and when they leave they deregister. The pseudo-code of this UC blockchain protocol is given in the remainder of this section. For the general structure of our UC blockchain model, we refer to Fig. 5.

The Bitcoin ledger protocol assumes as hybrids a random oracle $\mathcal{F}^{\kappa}_{\text{RO}}$, a network $\mathcal{F}^{\Delta,\mathsf{bc}}_{\text{N-MC}}$ for blockchains, a network $\mathcal{F}^{\Delta,\mathsf{tx}}_{\text{N-MC}}$ for transactions, and clock $\mathcal{G}_{\text{CLOCK}}$. Note that the two networks are simply (named) instances of $\mathcal{F}^{\Delta}_{\text{N-MC}}$ and can be realized from a single network $\mathcal{F}^{\Delta}_{\text{N-MC}}$ using different message-IDs. The protocol is parametrized by q, $\mathtt{D}$, T where q is the number of mining attempts per round, $\mathtt{D}$ is the difficulty of the proof-of-work, and T is the number of blocks chopped off to obtain the ledger state.

5.3.1. *Registration, De-Registration and Initialization*

Recall from Sects. 2.1.4 and 3 that we model explicit registrations to make all our functionalities, and in particular the ledger functionality, regular. And thus this has to be reflected in the real-world protocol. However, registration and de-registration can be seen as explicit commands to start the operation of a protocol machine, and to stop the operation of a protocol machine, and to have this feature explicitly exposed to the higher-level protocol.

The formal registration process in the protocol works as follows. If a party receives (REGISTER, sid) from the environment it registers to all hybrid functionalities. Once registration has succeeded the party returns activation to the environment. Upon the next activation to maintain the ledger (MAINTAIN-LEDGER), the party initializes its local variables, multicasts a special NEW-PARTY message over the network, and executes the main maintenance sub-protocol (in an interruptible manner as further explained below). De-registering from the ledger (via a query (DE-REGISTER, sid)) from the environment) works analogously, upon which the party erases all its state and becomes idle until its is freshly invoked with a REGISTER-query.

Recall that the notion of de-synchronized parties is strongly connected to these registrations: if an active honest party is not registered to all hybrids for long enough after joining the protocol execution at some time $\tau > 0$, it is considered de-synchronized (and otherwise the party is synchronized). In particular, honest parties that register at the onset of the protocol execution are synchronized (until they get corrupted or de-registered).

5.3.2. *Ledger-Specific Queries*

Ledger-specific queries are the specific features that one wishes to implement. Our very basic ledger supports three operations (after registration):

Protocol Ledger-Protocol$_{q,\mathtt{D},T}(P)$

Variables and Initial Values:

- The protocol stores a local (working) chain $\mathcal{C}_{loc}$ which initially contains the genesis block, i.e., $\mathcal{C}_{loc} \leftarrow (\mathbf{G})$.
- It additionally manages a separate chain $\mathcal{C}_{\text{exp}}$ 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 a indication was received that a new party joined the network (initially WELCOME $= 0$).
- Two variables `doneWork` and `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 $(\text{REGISTER}, \text{sid})$, this party sends $(\text{REGISTER}, \text{sid})$ to $\mathcal{F}^{\Delta,\mathsf{bc}}_{\text{N-MC}}$, $\mathcal{F}^{\Delta,\mathsf{tx}}_{\text{N-MC}}$, $\mathcal{F}^{\kappa}_{\text{RO}}$, and $\mathcal{G}_{\text{CLOCK}}$ and outputs $(\text{REGISTER}, \text{sid}, P)$.
- Upon receiving $(\text{DE-REGISTER}, \text{sid})$, send $\mathcal{F}^{\Delta,\mathsf{bc}}_{\text{N-MC}}$, $\mathcal{F}^{\Delta,\mathsf{tx}}_{\text{N-MC}}$, $\mathcal{F}^{\kappa}_{\text{RO}}$. Set all variables back to their initial values and return $(\text{DE-REGISTER}, \text{sid}, P)$.

Ledger-Queries:
Ledger queries are only answered once registered.

- Upon receiving $(\text{SUBMIT}, \text{sid}, \mathtt{tx})$, set $\mathtt{buffer} \leftarrow \mathtt{buffer}||\mathtt{tx}$, and send $(\text{MULTICAST}, \text{sid}, \mathtt{tx})$ to $\mathcal{F}^{\Delta,\mathsf{tx}}_{\text{N-MC}}$.
- Upon receiving $(\text{READ}, \text{sid})$ send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$, receive as answer $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ and proceed as follows:
  if τ corresponds to an update mini-round and `isInit` and $\neg$`doneUpdate` **then**
  Execute sub-protocol **FetchInformation** and set `doneUpdate` $\leftarrow$ true.
  Let $\vec{\mathtt{st}}$ be the encoded state in $\mathcal{C}_{\text{exp}}$
  Return $(\text{READ}, \text{sid}, \vec{\mathtt{st}}^{\lceil T})$.
- Upon receiving $(\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$ execute in a $(\text{MAINTAIN-LEDGER}, \text{sid}, \text{minerID})$-interruptible manner the following:
  1. If `isInit` $=$ false, then set all variables to their initial values, set `isInit` $\leftarrow$ true and output $(\text{MULTICAST}, \text{sid}, \text{NEW-PARTY})$ to $\mathcal{F}^{\Delta,\mathsf{tx}}_{\text{N-MC}}$.
  2. Execute sub-protocol **Ledger-Maintenance**

Fig. 5. The main structure of the UC blockchain protocol.

Submitting a transaction. This one is very simple: when given a transaction a party multicasts the transaction.

Ledger maintenance. Ledger maintenance refers to activating the main mining procedure of Bitcoin and is given in Fig. 6. Since ledger maintenance consists of several complex steps that in particular lose activations, the execution proceeds in an interruptible manner as explained in Sect. 5.2.2. The main structure of maintenance enforces the mini-round structure: in a working mini-round, the protocol tries to obtain the solution to a proof-of-work puzzle for a newly generated state block. The core sub-protocol thereby is:

Sub-Protocol ExtendState($\mathtt{st}$)

$\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\mathrm{D}}(\mathcal{C}_{loc}, \mathtt{st}, q)$
if $\mathcal{C}_{\text{new}} \neq \mathcal{C}_{loc}$ **then**
  Update the local chain, i.e., $\mathcal{C}_{loc} \leftarrow \mathcal{C}_{\text{new}}$.
Send (MULTICAST, sid, $\mathcal{C}_{loc}$) to $\mathcal{F}^{\Delta,\mathsf{bc}}_{\text{N-MC}}$ ▷ Multicast current chain

It then enters an idle mode for maintenance queries until the clock advances and enters an update mini-round where new information is fetched from the network.

Sub-Protocol FetchInformation

Send (FETCH, sid) to $\mathcal{F}^{\Delta,\mathsf{bc}}_{\text{N-MC}}$; denote the response from $\mathcal{F}^{\Delta,\mathsf{bc}}_{\text{N-MC}}$ by (FETCH, sid, b).
Extract chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from b.
$\mathcal{C}_{loc}, \mathcal{C}_{\text{exp}} \leftarrow \mathsf{maxvalid}^{\mathrm{D}}_{\mathcal{B}}(\mathcal{C}_{loc}, \mathcal{C}_{\text{exp}}, \mathcal{C}_1, \ldots, \mathcal{C}_k)$
Send (FETCH, sid) to $\mathcal{F}^{\Delta,\mathsf{tx}}_{\text{N-MC}}$; denote the response from $\mathcal{F}^{\Delta,\mathsf{tx}}_{\text{N-MC}}$ by (FETCH, sid, b).
Extract received transactions $(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$ from b.
Set $\mathtt{buffer} \leftarrow \mathtt{buffer}||(\mathtt{tx}_1, \ldots, \mathtt{tx}_k)$.
If a NEW-PARTY message was received, set WELCOME $\leftarrow 1$. Otherwise, set WELCOME $\leftarrow 0$.
Remove all transactions from $\mathtt{buffer}$ which are invalid with respect to (the state of) $\mathcal{C}^{\lceil T}_{loc}$

Again the protocol is idle for maintenance queries until the clock advances.

Reading the state. When asked to report the current ledger state, the protocol outputs the prefix of the exported state, i.e., a prefix of the state encoded in $\mathcal{C}_{\text{exp}}$. By the mini-round structure, the exported state is updated exactly once in every update mini-rounds (after initialization is complete).

5.3.3. *Predictable Synchronization Pattern*

We now show that the ledger protocol has a predictable synchronization pattern according to Definition 3.1.

Lemma 5.2. *The protocol* $\mathsf{Ledger\text{-}Protocol}_{q,D,T}$ *satisfies Definition* 3.1. *More specifically, there is a predicate* $\mathsf{predict\text{-}time}_{BC}$ *that predicts the synchronization pattern of the UC Bitcoin protocol as required by Definition* 3.1.

Proof Sketch. This is follows by inspection of our ledger protocol (and all protocols that share the same structure as we will see later in all the respective hybrid worlds they are executed in). The predicate $\mathsf{predict\text{-}time}$ can be implemented as follows: browse through the entire sequence $\vec{\mathcal{I}}^T_H$ and determine how many times the clock advances. The clock advances for the first time, when all miners got sufficient maintain commands to complete their mini-round operation. By definition of $\mathsf{Ledger\text{-}Protocol}$, this implies that each party has sent a clock-update to the clock and hence the clock advances. By an inductive argument, whenever the clock has ticked, the check when the clock advances the next time is checked exactly the same way. Overall, this allows to check whether the next activation of an honest party, given the history of activations will provoke a clock update (and knowing which parties are corrupted). □

Sub-Protocol Ledger-Maintenance

This sub-protocol is executed in a (MAINTAIN-LEDGER, *sid*, *minerID*)*-interruptible manner*

Step 1: Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$, receive as answer $(\text{CLOCK-READ}, \text{sid}_C, \tau)$, and proceed according to the following case distinction.

Step 2: **if** τ corresponds to a working mini-round **then**
- **if** $\neg$`doneWork` **then**
  - Let $\vec{\texttt{st}}$ be the encoded state in $\mathcal{C}_{loc}$
  - Set `buffer'` $\leftarrow$ `buffer`
  - Parse `buffer'` as sequence $(\texttt{tx}_1, \dots, \texttt{tx}_n)$
  - Set $\vec{N} \leftarrow \texttt{tx}^{\text{coin-base}}_{\text{minerID}}$
  - Set $\texttt{st} \leftarrow \mathsf{blockify}_{฿}(\vec{N})$
  - **repeat**
    - Let $(\texttt{tx}_1, \dots, \texttt{tx}_n)$ be the current list of (remaining) transactions in `buffer'`
    - **for** $i = 1$ to n **do**
      - **if** $\mathsf{ValidTx}_{฿}(\texttt{tx}_i, \vec{\texttt{st}}||\texttt{st}) = 1$ **then**
        - $\vec{N} \leftarrow \vec{N}||\texttt{tx}_i$
        - Remove `tx` from `buffer'`
        - Set $\texttt{st} \leftarrow \mathsf{blockify}_{฿}(\vec{N})$
  - **until** $\vec{N}$ does not increase anymore
  - Execute **ExtendState(st)**
  - If the flag WELCOME $= 1$, send $(\text{MULTICAST}, \text{sid}, \texttt{buffer})$ to $\mathcal{F}^{\text{tx}}_{\text{N-MC}}$. Otherwise, give up activation.
- Set `doneWork` $\leftarrow$ true, `doneUpdate` $\leftarrow$ false, and send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$

Step 3: **if** τ corresponds to an update mini-round **then**
- **if** $\neg$`doneUpdate` **then**
  - Execute **FetchInformation**
- Set `doneUpdate` $\leftarrow$ true, `doneWork` $\leftarrow$ false, and send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$

Fig. 6. The maintenance procedure of the UC Bitcoin protocol.

6. The Bitcoin Ledger

We next show how to instantiate the ledger functionality from Sect. 4 with appropriate parameters so that it is implemented by protocol Ledger-Protocol. The proof of this appears in the next section. To define this Bitcoin ledger $\mathcal{G}^{฿}_{\text{LEDGER}}$, we give the specific instantiations of the relevant functions Validate, Blockify, ExtendPolicy, and predict-time.

Synchrony pattern. First, predict-time is defined to be $\mathsf{predict\text{-}time}_{BC}$ to reflect the synchronization pattern of the UC Bitcoin protocol as described in the proof of Lemma 5.2. This shows the dependency of the realized ledger from the protocol that achieves it.

State-buffer-validation. Similarly, in case of Validate we use the same predicate as the protocol uses to validate the states: For a given transaction `tx` and a given state `state`, the predicate decides whether this transaction is valid with respect to `state`. Given such a validation predicate, the ledger validation predicate takes a specific simple form which, excludes dependency on anything other than the transaction `tx` and the state `state`, i.e., for any values of txid, τ_L, P_i, and `buffer`:

$$\mathsf{Validate}((\texttt{tx}, \text{txid}, \tau_L, P_i), \texttt{state}, \texttt{buffer}) := \mathsf{ValidTx}_{฿}(\texttt{tx}, \texttt{state}).$$

Ledger-output format. As with the above parameters, the function Blockify is defined to be $\mathsf{blockify}_{฿}$, i.e., the function used in the UC Bitcoin protocol. In principle, any formatting function can be used and the security proof goes through (as long as the same function is used in the protocol Ledger-Protocol and functionality $\mathcal{G}_{\text{LEDGER}}^{B}$). However, as we observe below in Definition 6.1, a meaningful Blockify should be in certain relation with the ledger's Validate predicate. This relation is satisfied by the Bitcoin protocol.

The ledger policy. Finally, we define ExtendPolicy. At a high level, upon receiving a list of possible candidate blocks which should go into the state of the ledger, ExtendPolicy does the following: for each block it first verifies that the blocks are valid with respect to the state they extend. Only valid blocks might be added to the state. In particular, ExtendPolicy is parameterized by three parameters—two positive integers $\texttt{maxTime}_{\texttt{window}}$, $\texttt{minTime}_{\texttt{window}}$, and a positive fraction η—and ensures the following property:

1. The speed of the ledger is not too slow. This is implemented by defining an upper limit $\texttt{maxTime}_{\texttt{window}}$ on the time (number of clock-ticks) it takes to add $\texttt{windowSize}$ state blocks. The enforced minimal ledger growth rate is expressed as the fraction $\frac{\texttt{windowSize}}{\texttt{maxTime}_{\texttt{window}}}$.
2. The speed of the ledger is not too fast. This is implemented by defining a lower bound $\texttt{minTime}_{\texttt{window}}$ on the time it takes to add $\texttt{windowSize}$ state blocks. The enforced maximal ledger growth rate is expressed as $\frac{\texttt{windowSize}}{\texttt{minTime}_{\texttt{window}}}$.
3. The adversary cannot create too many blocks with arbitrary (but valid) contents. This is formally enforced by defining an upper bound η on the ratio these so-called adversarial blocks within any sequence of $\texttt{windowSize}$ (or more) state blocks. This is known as chain quality. Formally, this is enforced by requiring that a certain fraction of blocks need to satisfy higher-quality standards (to model blocks that are honestly generated).
4. Last but not least, ExtendPolicy guarantees that if a transaction is "old enough", and still valid with respect to the actual state, then it is included into the state. This is a weak form of guaranteeing that a transaction will make it into the state unless it is in conflict. As we show in Sect. 9, this guarantee can be amplified by using digital signatures.

In order to enforce these policies, ExtendPolicy first defines alternative blocks which satisfy all of the above criteria in an ideal way, and whenever it catches the adversary in trying to propose blocks that do not obey the policies, it punishes the adversary by proposing its own generated blocks. In particular, if the adversary violates the policy regarding minimal chain-growth, the ExtendPolicy will directly propose a sequence of complying blocks and hence ensure liveness in a strong sense. The precise formal description of the extend policy (as pseudo-code) for $\mathcal{G}_{\text{LEDGER}}^{B}$ is given in Appendix B for completeness.

On the relation between Blockify and Validate. As already discussed above, ExtendPolicy guarantees that the adversary cannot block the extension of the state indefinitely, and that occasionally an honest miner will create a block. These are implications of the chain-growth and chain-quality properties from [24]. However, our generic ExtendPolicy makes explicit that a priori, we cannot exclude that the chain always extends with blocks that include, for example, only a coin-base transaction, i.e., any submitted transaction is ignored and never inserted into a new block. This issue is an orthogonal one to ensuring that honest transactions are not invalidated by adversarial interaction—which, as argued in [24], is achieved by adding digital signatures.

To see where this could be problematic in general, consider a blockify that, at a certain point, creates a block that renders all possible future transactions invalid. Observe that this does not mean that our protocol is insecure and that this is as well possible for the protocols of [24,44]; indeed, our proof shows that the protocol will give exactly the same guarantees as an $\mathcal{G}_{\text{LEDGER}}$ parametrized with such an algorithm $\mathsf{Blockify}$.

Nonetheless, a look in reality indicates that this situation never occurs with Bitcoin. To capture that this is the case, $\mathsf{Validate}$ and $\mathsf{Blockify}$ need to be in a certain relation with each other. Informally, this relation should ensure that the above sketched situation does not occur, i.e., $\mathsf{Blockify}$ should "not affect" the "true validity" of a transaction. A way to ensure this, which is already implemented by the Bitcoin protocol, is by restricting $\mathsf{Blockify}$ to only make an invertible manipulation of the blocks when they are inserted into the state—e.g., be an encoding function—and define $\mathsf{Validate}$ to depend on the inverse of $\mathsf{Blockify}$. This is captured in the following definition.

Definition 6.1. A co-design of $\mathsf{Blockify}$ and $\mathsf{Validate}$ is *non-self-disqualifying* if there exists an efficiently computable function Dec mapping outputs of $\mathsf{Blockify}$ to vectors $\vec{N}$ such that there exists a validate predicate $\mathsf{Validate}'$ for which the following properties hold for any possible state $\mathtt{state} = \mathtt{st}_1 || \cdots || \mathtt{st}_\ell$, buffer $\mathtt{buffer}$ vectors $\vec{N} := (\mathtt{tx}_1, \ldots, \mathtt{tx}_m)$, and transaction $\mathtt{tx}$:

1. $\mathsf{Validate}(\mathtt{tx}, \mathtt{state}, \mathtt{buffer}) = \mathsf{Validate}'(\mathtt{tx}, \mathsf{Dec}(\mathtt{st}_1) || \cdots || \mathsf{Dec}(\mathtt{st}_\ell), \mathtt{buffer})$
2. $\mathsf{Validate}(\mathtt{tx}, \mathtt{state} || \mathsf{Blockify}(\vec{N}), \mathtt{buffer}) = \mathsf{Validate}'(\mathtt{tx}, \mathsf{Dec}(\mathtt{st}_1) || \ldots || \mathsf{Dec}(\mathtt{st}_\ell) || \vec{N}, \mathtt{buffer})$

We remark that the actual validation of Bitcoin does satisfy the above definition, since a transaction is only rendered invalid with respect to the state if the coins it is trying to spend have already been spent, and this only depends on the transactions in the state and not the metadata added by $\mathsf{Blockify}$. Hence, in the following, we assume that $\mathsf{ValidTx}_{\text{₿}}$ and $\mathsf{blockify}_{\text{₿}}$ satisfy the relation in Definition 6.1.

7. Security Analysis

7.1. *Overview*

In this section, we prove our main theorem, namely that, under appropriate assumptions, Bitcoin realizes the instantiation of the ledger functionality from the previous section. We prove our main theorem which can be described informally as follows:

Theorem. *(Informal). For the security parameter κ and assuming* $\mathtt{windowSize} = \omega(\log \kappa)$*, then the protocol* $\mathsf{Ledger\text{-}Protocol}$ *securely realizes the concrete ledger functionality* $\mathcal{G}^{\text{₿}}_{\text{LEDGER}}$ *defined in the previous section. The assumptions on network delays and* *mining power, where mining power is roughly understood as the ability to find proofs of work via queries to the random oracle (and will be formally defined later), are as follows:*

- *In any round of the protocol execution, the collective mining power of the adversary, contributed by corrupted and temporarily de-synchronized miners, does not exceed the mining power of honest (and synchronized) parties. The exact relation additionally captures the (negative) impact of network delays on the coordination of mining power of honest parties.*
- *No message can be delayed in the network by more than* $\Delta = O(1)$ *rounds.*

We prove the above theorem via what we believe is a useful modularization of the Bitcoin protocol (cf. Fig. 7). Informally, this modularization distills out from the protocol a reactive *state-extend* subprocess which captures the lottery that decides which miner gets to advance the blockchain next and additionally the process of propagating this state to other miners. In Lemma 7.2, we show that the state-extend-and-exchange module/-subprocess implements an appropriate reactive UC functionality $\mathcal{F}_{\text{STX}}$. We can then use the UC composition theorem which allows us to argue security of Ledger-Protocol in a simpler hybrid world where, instead of using this subprocess, parties make calls to the functionality $\mathcal{F}_{\text{STX}}$, which then leads us to our main analysis in Theorem 7.9.

7.2. *First Proof Step*

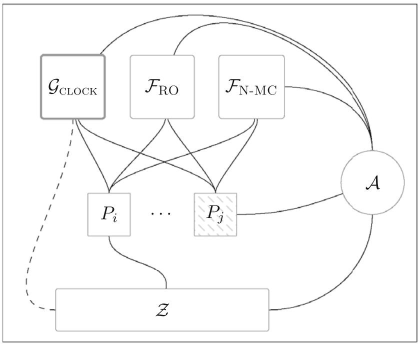
In a first step, we distill out from the protocol Ledger-Protocol a state-extend module/-subprocess, denoted as StateExchange-Protocol, and show, using a "game-hopping" argument, that a modular description of the Ledger-Protocol in which every party makes invocations of this subprocess, yields an equivalent protocol. We abstract the service provided by this subprocess by a new lottery-functionality denoted $\mathcal{F}_{\text{STX}}$. The modularized protocol, defined for the $\mathcal{F}_{\text{STX}}$-hybrid world, is denoted by Modular-Ledger-Protocol.

As we prove, the subprocess StateExchange-Protocol UC-realizes $\mathcal{F}_{\text{STX}}$ and hence the original protocol Ledger-Protocol and the modularized protocol Modular-Ledger-Protocol are in fact indistinguishable. This final step is a direct consequence of the universal composition theorem: Ledger-Protocol UC emulates Modular-Ledger-Protocol where invocations of StateExchange-Protocol are replaced by invocations of $\mathcal{F}_{\text{STX}}$ (for appropriate parameters as precisely defined below).

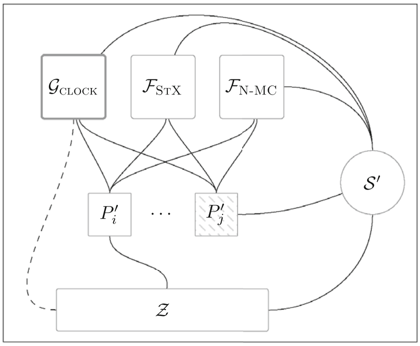
Looking ahead, in the next section, we can hence focus on analyzing the simpler protocol Modular-Ledger-Protocol in order to show that the UC Bitcoin protocol realizes the Bitcoin Ledger of Sect. 6—again by invoking the composition theorem.

7.2.1. *The State-Exchange Functionality*

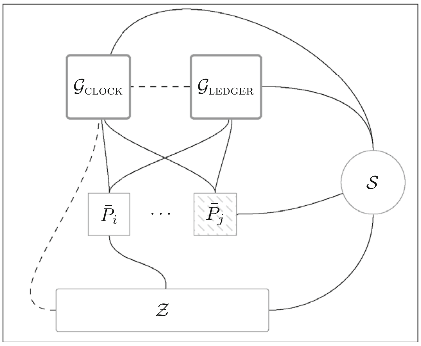
The state-exchange functionality $\mathcal{F}_{\text{STX}}^{\Delta,p_H,p_A}$ allows parties to submit ledger states which are accepted with a certain probability. Accepted states are then multicast to all parties. Informally, it can be seen as lottery on which (valid) states are exchanged among the participants. Note that for simplicity of notation we do not write the parameters when clear from the context.



(a) In the real world parties have access to the global clock $\mathcal{G}_{\text{CLOCK}}$, the random oracle $\mathcal{F}_{\text{RO}}^{\kappa}$, and network $\mathcal{F}_{\text{N-MC}}$. Here, parties execute the Bitcoin protocol Ledger-Protocol



(b) In the hybrid world parties have access to the state-exchange functionality $\mathcal{F}_{\text{STX}}$ (instead of the random oracle). Here, parties execute the modularized protocol Modular-Ledger-Protocol



(c) In the ideal world, dummy parties have access to the global clock $\mathcal{G}_{\text{CLOCK}}$ and the ledger $\mathcal{G}_{\text{LEDGER}}$

Fig. 7. Modularization of the Bitcoin protocol.

Parties can use $\mathcal{F}_{\text{STX}}$ to multicast a valid state, but instead of accepting any submitted state and sending it to all (registered) parties, $\mathcal{F}_{\text{STX}}$ keeps track of all states that it ever saw, and implements the following mechanism upon submission of a state $\vec{\mathtt{st}}$ and a new block $\mathtt{st}$ from any party: If $\vec{\mathtt{st}}$ was previously accepted by $\mathcal{F}_{\text{STX}}$ and $\vec{\mathtt{st}}||\mathtt{st}$ is a valid new state, then $\mathcal{F}_{\text{STX}}$ accepts $\vec{\mathtt{st}}||\mathtt{st}$ with probability p_H (resp. p_A for dishonest parties) and sends it to registered parties. Each submission is evaluated independently. The formal specification is found in Fig. 8.

7.2.2. *Realizing the State-Exchange Functionality*

The state-exchange functionality is realized by the protocol given below. It is obtained by identifying the relevant instructions from the UC-ledger protocol. More precisely, protocol StateExchange-Protocol UC-realizes the $\mathcal{F}_{\text{STX}}$ functionality in the ($\mathcal{F}_{\text{RO}}^{\kappa}$, $\mathcal{F}_{\text{N-MC}}^{\text{bc}}$)-hybrid world. Note that $\mathcal{F}_{\text{N-MC}}^{\text{bc}}$ is a (named) instance of the $\mathcal{F}_{\text{N-MC}}^{\Delta}$ functionality. The protocol is parametrized by q and $\mathtt{D}$ where q is the number of mining attempts per submission attempt and $\mathtt{D}$ is the difficulty of the proof-of-work.

Functionality $\mathcal{F}_{\mathsf{StX}}^{\Delta,p_H,p_A}$

Initialization:

The functionality initializes the party set $\mathcal{P} \leftarrow \emptyset$ and 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 $\mathbf{N}_{\text{net}}$ of adverbially injected chunk messages (that might not correspond to valid states).

Registrations:

- Upon receiving $(\text{REGISTER}, \text{sid})$ from some party P (or from $\mathcal{A}$ on behalf of a corrupted P), set $\mathcal{P} = \mathcal{P} \cup \{P\}$, and initialize the tree $\mathcal{T}_P \leftarrow \texttt{gen}$ where each rooted path corresponds to a valid state the party has received. Return $(\text{REGISTER}, \text{sid}, P)$ to the caller.

- Upon receiving $(\text{DE-REGISTER}, \text{sid})$ 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 $(\text{DE-REGISTER}, \text{sid}, P)$ to the caller.

Submit/receive new states:

– Upon receiving $(\text{SUBMIT-NEW}, \text{sid}, \vec{\texttt{st}}, \texttt{st})$ from some participant $P_s \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted P_s), if $\mathsf{isvalidstate}_{\mathfrak{B}}(\vec{\texttt{st}}||\texttt{st}) = 1$ and $\vec{\texttt{st}} \in \mathcal{T}_P$ do the following:

  1. If P_s is honest, sample B according to a Bernoulli-Distribution with parameter p_H. Otherwise, sample B with parameter p_A.
  2. If $B = 1$, set $\vec{\texttt{st}}_{new} \leftarrow \vec{\texttt{st}}||\texttt{st}$ and add $\vec{\texttt{st}}_{new}$ to $\mathcal{T}_{P_s}$. Else set $\vec{\texttt{st}}_{new} \leftarrow \vec{\texttt{st}}$.
  3. Output $(\text{SUCCESS}, \text{sid}, B)$ to P_s.
  4. On response $(\text{CONTINUE}, \text{sid})$ where $\mathcal{P} = \{P_1, \ldots, P_n\}$ choose n new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$, initialize n new variables $D_{\mathsf{mid}_1} := D_{\mathsf{mid}_1}^{MAX} := \ldots := D_{\mathsf{mid}_n} := D_{\mathsf{mid}_n}^{MAX} := 1$ set $\vec{M} := \vec{M}||(\vec{\texttt{st}}_{new}, \mathsf{mid}_1, D_{\mathsf{mid}_1}, P_1)||\ldots||(\vec{\texttt{st}}_{new}, \mathsf{mid}_n, D_{\mathsf{mid}_n}, P_n)$, and send $(\text{SUBMIT-NEW}, \text{sid}, \vec{\texttt{st}}_{new}, P_s, (P_1, \mathsf{mid}_1), \ldots, (P_n, \mathsf{mid}_n))$ to the adversary.

– Upon receiving $(\text{FETCH-NEW}, \text{sid})$ from a party $P \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted P), do the following:

  1. For all tuples $(\vec{\texttt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P) \in \vec{M}, \mathbf{N}_{\text{net}}$ update the value $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.
  2. Let $\vec{M}_0^P$ denote the subvector of $\vec{M}$ including all tuples of the form $(\vec{\texttt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P)$ where $D_{\mathsf{mid}} \leq 0$ (in the same order as they appear in $\vec{M}$). For each tuple $(\vec{\texttt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P) \in \vec{M}_0^P$ add $\vec{\texttt{st}}$ 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 $\mathbf{N}_{\text{net}}$ to the adversary.

Further adversarial influence:

– Upon receiving $(\text{SEND}, \text{sid}, \vec{\texttt{st}}, P')$ from $\mathcal{A}$ on behalf of some *corrupted* $P \in \mathcal{P}$, if $P' \in \mathcal{P}$ and $\vec{\texttt{st}} \in \mathcal{T}_P$, choose a new unique message-ID mid, initialize $D := 1$, add $(\vec{\texttt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P')$ to $\vec{M}$, and return $(\text{SEND}, \text{sid}, \vec{\texttt{st}}, P', \mathsf{mid})$ to $\mathcal{A}$. If $\vec{\texttt{st}} \notin \mathcal{T}_P$, then conduct the same steps except that $(\vec{\texttt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P')$ is added to $\mathbf{N}_{\text{net}}$.

– Upon receiving $(\text{SWAP}, \text{sid}, \mathsf{mid}, \mathsf{mid}')$ from $\mathcal{A}$, if mid and mid' are message-IDs both registered in the same message buffers, swap the corresponding tuples in that buffer. Return $(\text{SWAP}, \text{sid})$ to $\mathcal{A}$.

– Upon receiving $(\text{DELAY}, \text{sid}, T, \mathsf{mid})$ from $\mathcal{A}$, if T is a valid delay, mid is a message-ID for a tuple $(\vec{\texttt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, P)$ in the current message buffers and $D_{\mathsf{mid}}^{MAX} + T \leq \Delta$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} + T$ and set $D_{\mathsf{mid}}^{MAX} := D_{\mathsf{mid}}^{MAX} + T$.

– Upon receiving $(\text{GET-REGISTERED}, \text{sid})$ from $\mathcal{A}$, the functionality returns the response $(\text{GET-REGISTERED}, \text{sid}, \mathcal{P})$ to $\mathcal{A}$.

Fig. 8. The state exchange functionality. Parameters are the delay Δ and the success probabilities p_H and p_A for honest and adversarial submissions.

Protocol StateExchange-Protocol$_{q,\mathrm{D}}(P)$

Initialization:

The protocol maintains a tree $\mathcal{T}$ of all valid chains. Initially it contains the genesis chain (**G**).

Registration/De-Registration:

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

Exchange: *Exchange queries are only answered once registered.*

- Upon receiving (SUBMIT-NEW, sid, $\vec{\mathtt{st}}$, $\mathtt{st}$) do

  if isvalidstate$_{\mathfrak{B}}(\vec{\mathtt{st}}||\mathtt{st}) = 1$ **then** ▷ Check if there exists a chain in $\mathcal{T}$ which contains the state $\vec{\mathtt{st}}$
  if there exists $\mathcal{C} \in \mathcal{T}$ with $\vec{\mathtt{st}}$ **then**
  $\mathcal{C}_{\text{new}} \leftarrow$ extendchain$_{\mathrm{D}}(\mathcal{C}, \mathtt{st}, q)$ ▷ Try to extend the chain
  if $\mathcal{C}_{\text{new}} \neq \mathcal{C}$ **then**
  Update the local tree, i.e., add $\mathcal{C}_{\text{new}}$ to $\mathcal{T}$
  Output (SUCCESS, sid, 1) to P.
  else
  Output (SUCCESS, sid, 0) to P.
  On response (CONTINUE, sid) send (MULTICAST, sid, $\mathcal{C}_{\text{new}}$) to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$. ▷ Broadcast current chain

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

  Send (FETCH, sid) to $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ 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.

Lemma 7.1. *Let $p := \frac{D}{2^\kappa}$. The protocol* StateExchange-Protocol$_{q,D}$ *UC-realizes functionality $\mathcal{F}^{\Delta,p_H,p_A}_{\text{STX}}$ in the $(\mathcal{F}^{\kappa}_{\text{RO}}, \mathcal{F}^{\Delta}_{\text{N-MC}})$-hybrid model where $p_A := p$ and $p_H := 1-(1-p)^q$.*

Proof. We consider the following simulator:

Simulator $\mathcal{S}_{\text{stx}}$

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 random 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 $\mathcal{F}^{\kappa}_{\text{RO}}$ from $\mathcal{A}$ on behalf of corrupted $P \in \mathcal{P}^a$ do the following.
  1. If $H[v]$ is already defined, output (EVAL, sid, v, $H[v]$).
  2. If v is of the form $(\mathtt{s}, \mathtt{st}, \mathtt{n})$ and there exists[b] a chain $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ such that $H[\mathbf{B}_n] = \mathtt{s}$ 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 (SUBMIT-NEW, sid, $\vec{\mathtt{st}}$, $\mathtt{st}$) to $\mathcal{F}_{\text{STX}}$ and denote by (SUCCESS, B) the output of $\mathcal{F}_{\text{STX}}$. If $B = 1$ set $H[v]$ to a uniform random value in $\{0,1\}^\kappa$ strictly smaller[c] than D. Add $\mathcal{C}||v$ to $\mathcal{T}$. Otherwise set $H[v]$ to a uniform random value in $\{0,1\}^\kappa$ larger than D. Output (EVAL, sid, v, $H[v]$).
  3. Otherwise set v to a uniform random value in $\{0,1\}^\kappa$ and output (EVAL, sid, v, $H[v]$).

Simulating the Network:

- Upon receiving (MULTICAST, sid, $(m_{i_1}, P_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell})$) for $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ 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 (SEND, sid, $\vec{\mathtt{st}}$, P_{i_j}) to $\mathcal{F}_{\mathrm{STX}}$. Store the message-ID $\widehat{\mathsf{mid}_{i_j}}$ returned by $\mathcal{F}_{\mathrm{STX}}$ 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 $\mathcal{F}_{\mathrm{STX}}$ to obtain their mid.
3. Denote the obtained message-IDs by $\mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_\ell}$, initialize ℓ new variables $D_{\mathsf{mid}_{i_1}} := \ldots := D_{\mathsf{mid}_{i_\ell}} := 1$, set $\vec{M} := \vec{M}||(m_{i_1}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}, P_{i_1})|| \ldots ||(m_{i_\ell}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_\ell}}, P_\ell)$.
4. Output (MULTICAST, sid, $(m_{i_1}, P_{i_1}, \mathsf{mid}_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell}, \mathsf{mid}_{i_\ell})$) to $\mathcal{A}$.

- Upon receiving (FETCH, sid) for $\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}$ from $\mathcal{A}$ on behalf of corrupted $P \in \mathcal{P}_{net}$ proceed as follows.
  1. Fetch in the name of party P from $\mathcal{F}_{\mathrm{STX}}$ 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 (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 $(T_{\mathsf{mid}}, \mathsf{mid})$ 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 $D_{\mathsf{mid}} := \max\{1, D_{\mathsf{mid}} + T_{\mathsf{mid}}\}$; otherwise, ignore this tuple.
  2. If the simulator knows a corresponding $\mathcal{F}_{\mathrm{STX}}$-message-ID $\widehat{\mathsf{mid}}$ for mid send (DELAY, sid, $T_{\mathsf{mid}}, \widehat{\mathsf{mid}}$) to $\mathcal{F}_{\mathrm{STX}}$.
- Upon receiving a message (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 $\mathcal{F}_{\mathrm{STX}}$-message-IDs $\widehat{\mathsf{mid}_1}$ and $\widehat{\mathsf{mid}_2}$ send (SWAP, sid, $\widehat{\mathsf{mid}_1}, \widehat{\mathsf{mid}_2}$) to $\mathcal{F}_{\mathrm{STX}}$.
  3. Output (SWAP, sid) to $\mathcal{A}$.

Interaction with the State Exchange Functionality :

- Upon receiving (SUBMIT-NEW, sid, $\vec{\mathtt{st}}, P_s, (P_1, \widehat{\mathsf{mid}_1}), \ldots, (P_n, \widehat{\mathsf{mid}_n})$) from $\mathcal{F}_{\mathrm{STX}}$ 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 $\vec{M}||(\mathcal{C}, \mathsf{mid}_{i1}, D_{\mathsf{mid}_1}, P_1)|| \ldots ||(\mathcal{C}, \mathsf{mid}_n, D_{\mathsf{mid}_n}, P_n)$, and store the message-IDs $\widehat{\mathsf{mid}_i}$ along the message-IDs mid_i.
     Output (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 $\mathtt{st}_1, \ldots, \mathtt{st}^d_{k-1}$. Choose a random nonce $\mathtt{n}$ and set $\mathbf{B}_k = (H[\mathbf{B}_{k-1}], \mathtt{st}_k, \mathtt{n})$ and set $H[\mathbf{B}_k]$ to a uniform random value in $\{0, 1\}^\kappa$ strictly smaller than $\mathtt{D}$. Add the chain $\mathcal{C} = \mathcal{C}'||\mathbf{B}_k$ to $\mathcal{T}$.
     Generate new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$, initialize $D_1 := \cdots := D_n = 1$, set $\vec{M}||(\mathcal{C}, \mathsf{mid}_{i1}, D_{\mathsf{mid}_1}, P_1)|| \ldots ||(\mathcal{C}, \mathsf{mid}_n, D_{\mathsf{mid}_n}, P_n)$, and store the message-IDs $\widehat{\mathsf{mid}_i}$ along the message-IDs mid_i. Output (MULTICAST, sid, $\mathcal{C}, P_s, (P_1, \mathsf{mid}_1), \ldots, (P_n, \mathsf{mid}_n)$) to the adversary.

[a] We do not write explicitly the instruction via which the simulator obtains $\mathcal{P}$ from $\mathcal{F}_{\mathrm{STX}}$.
[b] This can be checked efficiently using H under the assumption that there are no collisions.
[c] Can be done efficiently using rejection sampling.
[d] Such a chain must exist as $\mathtt{st}_1, \ldots, \mathtt{st}_{k-1}$ is a successfully submitted state in $\mathcal{F}_{\mathrm{STX}}$ in which case the simulator knows a corresponding chain.

The proof works similar as the one for Lemma 5.1 in [44]. Recall the notation from Sect. 2.1 and introduce the shorthand notation $T_{\text{real}} := T_{\text{EXEC}_{\mathsf{StateExchange\text{-}Protocol}, \mathcal{A}, \mathcal{Z}}}(\kappa, z)$ which is the (distribution of the) joint view of all parties in the execution of StateExchange-Protocol for adversary $\mathcal{A}$ and environment $\mathcal{Z}$ (upon some input z). Denote by $T_{\text{ideal}} := T_{\text{EXEC}_{\mathcal{F}_{\mathrm{STX}}, \mathcal{S}_{\text{stx}}, \mathcal{Z}}}(\kappa, z)$ the joint view of all parties for $\mathcal{F}_{\mathrm{STX}}$ with simulator $\mathcal{S}_{\text{stx}}$. In the following, we treat the arguments κ and z as implicit.

Define a new hybrid world, via the following random experiment: the experiment is defined as the real-world execution except that the random oracle aborts on collisions with COLLISION-ERROR and that adversarial oracle queries are emulated as in $\mathcal{S}_{\text{stx}}$. We use the shorthand $\text{HYB}_{\mathcal{A},\mathcal{Z}}$ to refer to this hybrid world (defined analogously to $\text{EXEC}_{\cdot,\cdot,\mathcal{Z}}$). The only difference is thus that in the hybrid world we may abort with COLLISION-ERROR or TREE-ERROR as in the ideal execution. Let T_{hyb} be the associated distribution of the joint view.

Let $\mathsf{event1}$ be the event that some parties query two different values v, v' such that $H[v] = H[v']$, i.e. the event that a hash-collision occurs (this event is a condition on the realized transcript tr in the support of T_{real} or T_{hyb}, respectively). For any two queries the probability that they return the same hash value is $2^{-\kappa}$. By a union bound over all queries we have that $\mathsf{event1}$ happens with probability at most $\mathsf{poly}(\kappa) \cdot 2^{-\kappa}$ in both worlds. Note that if $\mathsf{event1}$ does not happen the hybrid random experiment does not abort with COLLISION-ERROR.

Let $\mathsf{event2}$ be the event that some party makes a query $H[(\mathtt{s}, \cdot, \cdot)]$ where no v exists such that $H[v] = \mathtt{s}$, but later some party makes a query v' such that $H[v'] = \mathtt{s}$. The probability that any query $H[(\mathtt{s}, \cdot, \cdot)]$ a later query returns $\mathtt{s}$ is $2^{-\kappa}$ in both worlds By a union bound over all queries we have that $\mathsf{event2}$ happens with probability at most $\mathsf{poly}(\kappa) \cdot 2^{-\kappa}$ in both worlds.

Next, we show that the TREE-ERROR abort does not occur in the hybrid world execution conditioned under $\mathsf{event1}$ and $\mathsf{event2}$ not happening. Assume for contradiction that $\text{HYB}_{\mathcal{A},\mathcal{Z}}$ aborts with TREE-ERROR with $\mathsf{event1}$ and $\mathsf{event2}$ not happening. Let $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ be the shortest valid chain created in the experiment $\text{HYB}_{\mathcal{A},\mathcal{Z}}$ such that $\mathbf{B}_1, \ldots, \mathbf{B}_{n-1} \in \mathcal{T}$ but $\mathbf{B}_1, \ldots, \mathbf{B}_n \notin \mathcal{T}$. Let $\mathbf{B}_i = (\mathtt{s}_i, \mathtt{st}_i, \mathtt{n}_i)$. Since $\mathcal{C}$ is a valid chain we have $H[(\mathtt{s}_n, \mathtt{st}_n \mathtt{n}_n)] < D$. But at the time $\mathbf{B}_n$ was added to H no valid chain existed where the last block has hash value $\mathtt{s}_n$ (otherwise $\mathcal{C}$ would be in $\mathcal{T}$). This implies that no earlier query to H could have returned $\mathtt{s}_n$, since if the query was $\mathbf{B}_{n-1}$ $\mathcal{C}$ would not be the shortest chain with the above property and if the query was not $\mathbf{B}_{n-1}$ the event $\mathsf{event1}$ must have happened. This implies that $\mathsf{event2}$ must have happened, which is a contradiction.

This implies that conditioned under $\mathsf{event1}$ and $\mathsf{event2}$ not happening, the hybrid-world execution proceeds the same as the real-world execution and hence the two worlds are statistically close with respect to efficient environments $\mathcal{Z}$, i.e., $\text{EXEC}_{\mathsf{StateExchange\text{-}Protocol},\mathcal{A},\mathcal{Z}} \approx \text{HYB}_{\mathcal{A},\mathcal{Z}}$.

Now we compare $\text{HYB}_{\mathcal{A},\mathcal{Z}}$ and $\text{EXEC}_{\mathcal{F}_{\text{STX}},\mathcal{S}_{\text{stx}},\mathcal{Z}}$. Consider the event where a honest miner queries a block $(\mathtt{s}, \mathtt{st}, \mathtt{n})$ and fails, i.e., where $H[(\mathtt{s}, \mathtt{st}, \mathtt{n})] > \mathtt{D}$. In the hybrid execution, this query is stored in the random oracle table while the simulator in the ideal world does not store the query in the random oracle table. Under the condition that such failed queries are not repeated, the hybrid-world execution and the ideal-world execution proceed in identical ways (note that the network simulation in $\mathcal{S}_{\text{stx}}$ perfectly mimics the real and the hybrid worlds).

Note that the nonce $\mathtt{n}$ in a 'failed' query $(\mathtt{s}, \mathtt{st}, \mathtt{n})$ is chosen uniformly at random from $\{0, 1\}^{\kappa}$ by honest parties. This implies that with probability $\mathsf{poly}(\kappa) \cdot 2^{-\kappa}$ it was never queried before. As honest miner discard 'failed' queries (and failed queries do not leave the ITI and hence are hidden from the adversary) it also follows that except with probability $\mathsf{poly}(\kappa) \cdot 2^{-\kappa}$ the query will not be queried again (by any honest or corrupted

Protocol Modular-Ledger-Protocol$_T(P)$

Variables and Initial Values:

- The same as in the original protocol, cf. Fig. 5, except replace:

  The protocol stores a local (working) chain $\mathcal{C}_{loc}$ which initially contains the genesis block, i.e., $\mathcal{C}_{loc} \leftarrow (\mathbf{G})$.

  by

  The protocol manages the exported ledger state $\vec{st}_{\mathrm{exp}}$ which initially is the genesis state, i.e. , $\vec{st} \leftarrow (gen)$. It also manages a local (working) state $\vec{st}_{\mathrm{loc}}$ (initially also the genesis state).

Registration/De-Registration:

- As in the original protocol, cf. Fig. 5, but where the two local setup functionalities $(\mathcal{F}^{\mathsf{bc}}_{\text{N-MC}}, \mathcal{F}^{\kappa}_{\text{RO}})$ are subsumed by one local functionality $\mathcal{F}_{\text{STX}}$.

Ledger-Queries:
Ledger queries are only answered once registered.

- As in the original protocol, cf. Fig. 5 with different implementations of the subroutines ExtendState($\mathtt{st}$) and FetchInformation($\mathtt{st}$) as defined in Sect. 7.3.

Fig. 9. The modular ledger protocol (differences to original protocol shown).

party) unless the nonce of that failed query would be successfully guessed. By a union bound over all failed queries we have that failed queries are never queried twice except with probability $\mathsf{poly}(\kappa) \cdot 2^{-\kappa}$. Thus, $\text{EXEC}_{\mathcal{F}_{\text{STX}}, \mathcal{S}_{\text{stx}}, \mathcal{Z}} \approx \text{HYB}_{\mathcal{A}, \mathcal{Z}}$.

This concludes the proof. □

7.3. *Modularizing the Ledger-Protocol*

From the ledger protocol Ledger-Protocol$_{q,\text{D},T}$ we can derive what we denote Modular-Ledger-Protocol$_T$, which uses the state-exchange functionality to extend and exchange blockchain states. This is defined in Fig. 9, where the only non-trivial modifications (aside of some minor structural changes) are the replaced implementations of algorithms ExtendState($\mathtt{st}$) and FetchInformation. The new implementations are as follows:

Sub-Protocol ExtendState($\mathtt{st}$)

Send (SUBMIT-NEW, sid, $\vec{\mathtt{st}}_{\mathrm{loc}}$, $\mathtt{st}$) to $\mathcal{F}_{\text{STX}}$.
Denote the response by (SUCCESS, sid, B) of $\mathcal{F}_{\text{STX}}$.
if $B = 1$ **then**
  Update the local state, i.e., $\vec{\mathtt{st}}_{\mathrm{loc}} \leftarrow \vec{\mathtt{st}}_{\mathrm{loc}} || \mathtt{st}$.
Send (CONTINUE, sid) to $\mathcal{F}_{\text{STX}}$ ▷ Broadcast current state using $\mathcal{F}_{\text{STX}}$.

and

Sub-Protocol FetchInformation

Send (FETCH-NEW, sid) to $\mathcal{F}_{\text{STX}}$.
Denote the response from $\mathcal{F}_{\text{STX}}$ by (FETCH-NEW, sid, $(\vec{\texttt{st}}_1, \ldots, \vec{\texttt{st}}_k)$).
Set both $\vec{\texttt{st}}_{\text{loc}}, \vec{\texttt{st}}_{\text{exp}}$ to the longest state in $\vec{\texttt{st}}_{\text{loc}}, \vec{\texttt{st}}_{\text{exp}}, \vec{\texttt{st}}_1, \ldots, \vec{\texttt{st}}_k$ (to resolve ties the ordering decides).
Send (FETCH, sid) to $\mathcal{F}^{\text{tx}}_{\text{N-MC}}$; denote the response from $\mathcal{F}^{\text{tx}}_{\text{N-MC}}$ by (FETCH, sid, b).
Extract received transactions $(\texttt{tx}_1, \ldots, \texttt{tx}_k)$ from b.
Set $\texttt{buffer} \leftarrow \texttt{buffer}||(\texttt{tx}_1, \ldots, \texttt{tx}_k)$.
If a NEW-PARTY message was received, set WELCOME $\leftarrow$ 1. Otherwise, set WELCOME $\leftarrow$ 0.
Remove all transactions from buffer which are invalid with respect to $\vec{\texttt{st}}^{\lceil T}_{\text{loc}}$

We prove the soundness of this decomposition in the following lemma, which involves a sequence of hybrid steps to convert the original protocol into the suitable modular form, and finally by invoking Lemma,7.1.

Lemma 7.2. *The UC Bitcoin protocol* $\mathsf{Ledger\text{-}Protocol}_{q,D,T}$ *UC emulates* $\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T$ *that runs in a hybrid world with access to the functionality* $\mathcal{F}^{\Delta,p_H,p_A}_{\text{STX}}$ *with* $p_A := \frac{D}{2^\kappa}$ *and* $p_H = 1 - (1 - p_A)^q$*, and where* Δ *denotes the upper bound on the network delay.*

Proof. We first provide the sequence of modifications, morphing from the original protocol to the modularized protocol in a "game-hopping" style:

We start with the original Ledger-Protocol and consider the protocol part below where will alter the protocol step by step.

Fragments of Original Protocol Part

Initialization:

The protocol stores a local (working) chain $\mathcal{C}_{loc}$ which initially contains the genesis block, i.e., $\mathcal{C}_{loc} \leftarrow (\mathbf{G})$.
[...]

ExtendState(st):

$\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\text{D}}(\mathcal{C}_{loc}, \texttt{st}, q)$
if $\mathcal{C}_{\text{new}} \neq \mathcal{C}_{loc}$ **then**
  Update the local chain, i.e., $\mathcal{C}_{loc} \leftarrow \mathcal{C}_{\text{new}}$.
Send (MULTICAST, sid, $\mathcal{C}_{loc}$) to $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$ ▷ Multicast current chain

FetchInformation:

▷ Update the local state
Send (FETCH, sid) to $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$; denote the response from $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$ by (FETCH, sid, b).
Extract valid chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from b.
Set both $\mathcal{C}_{loc}, \mathcal{C}_{\text{exp}}$ to the longest valid chain in $\mathcal{C}_{loc}, \mathcal{C}_{\text{exp}}, \mathcal{C}_1, \ldots, \mathcal{C}_k$ (to resolve ties the ordering decides).
[...]

Modification 1. The first modification of the protocol (see below) proceeds as Ledger-Protocol except (a) it stores a history of all valid chains in a tree $\mathcal{T}$ and (b) in the **ExtendState(st)** procedure it checks that $\vec{\texttt{st}}||\texttt{st}$ is a valid state and that there exists a chain in $\mathcal{T}$ which encodes the state $\vec{\texttt{st}}$. We observe that the protocol calls **Extend-State(st)** only with $\texttt{st}$ where $\vec{\texttt{st}}||\texttt{st}$ is a valid state. This implies that the first check is always satisfied. Moreover, note that the current local chain $\mathcal{C}_{loc}$ which encodes state $\vec{\texttt{st}}$ is at any time stored in the tree $\mathcal{T}$. We therefore call the state encoded in $\mathcal{C}_{loc}$ by $\vec{\texttt{st}}_{\text{loc}}$

and see that the second check is therefore also always satisfied. Hence, the modified protocol has the same input/output behavior as Ledger-Protocol.

Fragments, Modification 1

Initialization:

The protocol stores a local (working) chain $\mathcal{C}_{loc}$ which initially contains the genesis block, i.e., $\mathcal{C}_{loc} \leftarrow (\mathbf{G})$.
[...]
The protocol additionally maintains a tree $\mathcal{T}$ of valid chains which initially contains the (genesis) chain $(\mathbf{G})$.

ExtendState(st):

if $\mathsf{isvalidstate}_{\mathbb{B}}(\vec{\mathtt{st}}_{\text{loc}}||\mathtt{st}) = 1$ **then**
  if there exists $\mathcal{C} \in \mathcal{T}$ which encodes $\vec{\mathtt{st}}_{\text{loc}}$ **then**
    $\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\mathrm{D}}(\mathcal{C}_{loc}, \mathtt{st}, q)$
    if $\mathcal{C}_{\text{new}} \neq \mathcal{C}_{loc}$ **then**
      Update the local chain, i.e., $\mathcal{C}_{loc} \leftarrow \mathcal{C}_{\text{new}}$.
      Add $\mathcal{C}_{loc}$ to $\mathcal{T}$
    Send (MULTICAST, sid, $\mathcal{C}_{\text{loc}}$) to $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$ ▷ Multicast current chain

FetchInformation:

Send (FETCH, sid) to $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$; denote the response from $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$ by (FETCH, sid, b).
Extract all valid chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from b and add them to $\mathcal{T}$.
Set both $\mathcal{C}_{loc}, \mathcal{C}_{\text{exp}}$ to the longest valid chain in $\mathcal{C}_{loc}, \mathcal{C}_{\text{exp}}, \mathcal{C}_1, \ldots, \mathcal{C}_k$ (to resolve ties the ordering decides).
[…]

Modification 2. In Modification 2 (see below) the local state $\vec{\mathtt{st}}_{\text{loc}}$ is stored directly instead of being encoded in chain $\mathcal{C}_{loc}$. The procedures **ExtendState(st)** and **FetchInformation** are modified to accommodate this change. Note that the $\mathcal{C}_{loc}$ is stored in $\mathcal{T}$ as we have seen in the first modification. This implies that the behavior of **ExtendState(st)** remains the same as in the first modification.

Fragments, Modification 2

Initialization:

The protocol manages [...] a local (working) state $\vec{\mathtt{st}}_{\text{loc}}$ (initially also the genesis state).[...]
The protocol additionally maintains a tree $\mathcal{T}$ of valid chains which initially contains the genesis chain $(\mathbf{G})$.

ExtendState(st):

if $\mathsf{isvalidstate}_{\mathbb{B}}(\vec{\mathtt{st}}_{\text{loc}}||\mathtt{st}) = 1$ **then**
  if there exists $\mathcal{C} \in \mathcal{T}$ which encodes $\vec{\mathtt{st}}_{\text{loc}}$ **then**
    $\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\mathrm{D}}(\mathcal{C}, \mathtt{st}, q)$
    if $\mathcal{C}_{\text{new}} \neq \mathcal{C}$ **then**
      Add $\mathcal{C}$ to $\mathcal{T}$
      Update the local state, i.e., $\vec{\mathtt{st}}_{\text{loc}} \leftarrow \vec{\mathtt{st}}_{\text{loc}}||\mathtt{st}$.
    Send (MULTICAST, sid, $\mathcal{C}_{\text{loc}}$) to $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$ ▷ Multicast current chain

FetchInformation:

Send (FETCH, sid) to $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$; denote the response from $\mathcal{F}^{\text{bc}}_{\text{N-MC}}$ by (FETCH, sid, b).
Extract all valid chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from b and add them to $\mathcal{T}$.
Extract all state $\vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k$ from chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$.
Set both $\vec{\mathtt{st}}_{\text{loc}}, \vec{\mathtt{st}}_{\text{exp}}$ to the longest state in $\vec{\mathtt{st}}_{\text{loc}}, \vec{\mathtt{st}}_{\text{exp}}, \vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k$ (to resolve ties the ordering decides).
[…]

Modification 3. In Modification 3 (see below) parts of the procedures **ExtendState(st)** and **FetchInformation** are split off into separate sub-procedures. Otherwise the protocol remains the same. As there are no changes to the program logic, the protocol still has the same behavior as the original protocol.

Fragments, Modification 3

Initialization:

The protocol manages [...] a local (working) state $\vec{\mathtt{st}}_{\mathrm{loc}}$ (initially also the genesis state).[...]
The protocol additionally maintains a tree $\mathcal{T}$ of valid chains which initially contains the (genesis) chain $(\mathbf{G})$.

ExtendState($\mathtt{st}$):

$B \leftarrow$ SUBMIT-NEW($\vec{\mathtt{st}}_{\mathrm{loc}}, \mathtt{st}$)
if $B = 1$ **then**
  Update the local state, i.e., $\vec{\mathtt{st}}_{\mathrm{loc}} \leftarrow \vec{\mathtt{st}}_{\mathrm{loc}} || \mathtt{st}$.
Execute CONTINUE. ▷ Broadcast current chain

Procedure SUBMIT-NEW($\vec{\mathtt{st}}, \mathtt{st}$):

if $\mathsf{isvalidstate}_{\vec{\mathsf{B}}}(\vec{\mathtt{st}}||\mathtt{st}) = 1$ **then**
  if there exists $\mathcal{C}' \in \mathcal{T}$ which encodes $\vec{\mathtt{st}}$ **then**
    Set $\mathcal{C} \leftarrow \mathcal{C}'$. ▷ $\mathcal{C}$ is assumed to be a global variable
    $\mathcal{C}_{\mathrm{new}} \leftarrow \mathsf{extendchain}_{\mathrm{D}}(\mathcal{C}, \mathtt{st}, q)$
    if $\mathcal{C}_{\mathrm{new}} \neq \mathcal{C}$ **then**
      Add $\mathcal{C}$ to $\mathcal{T}$
      return 1
    return 0

Procedure CONTINUE:

Send (MULTICAST, sid, $\mathcal{C}$) to $\mathcal{F}^{\mathsf{bc}}_{\mathrm{N\text{-}MC}}$

FetchInformation:

$(\vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k) \leftarrow$ FETCH-NEW
Set both $\vec{\mathtt{st}}_{\mathrm{loc}}, \vec{\mathtt{st}}_{\mathrm{exp}}$ to the longest state in $\vec{\mathtt{st}}_{\mathrm{loc}}, \vec{\mathtt{st}}_{\mathrm{exp}}, \vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k$ (to resolve ties the ordering decides).
[...]

Procedure FETCH-NEW:

Send (FETCH, sid) to $\mathcal{F}^{\mathsf{bc}}_{\mathrm{N\text{-}MC}}$; denote the response from $\mathcal{F}^{\mathsf{bc}}_{\mathrm{N\text{-}MC}}$ 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 $\vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_s$ from $\mathcal{C}_1, \ldots, \mathcal{C}_k$ and output them.

Final Considerations. We identify that Modification 3 above, in particular procedures SUBMIT-NEW, CONTINUE, and FETCH-NEW, are as defined in StateExchange-Protocol, hence invocations of this protocol. Consider the part of Modular-Ledger-Protocol below that we observe is the same as Modification 3 except that the handling of chains (variable $\mathcal{T}$) and the calls to sub-procedures SUBMIT-NEW, CONTINUE, and FETCH-NEW are now handled managed by the functionality $\mathcal{F}_{\mathrm{STX}}$ that is invoked at the respective places:

Modular-Ledger-Protocol (relevant part)

Initialization:

The protocol manages [...] a local (working) state $\vec{\mathtt{st}}_{\mathrm{loc}}$ (initially also the genesis state). [...]

ExtendState(st):

Send (SUBMIT-NEW, sid, $\vec{\mathtt{st}}_{\mathrm{loc}}$, $\mathtt{st}$) to $\mathcal{F}_{\mathrm{STX}}$.
Denote the response by (SUCCESS, sid, B) of $\mathcal{F}_{\mathrm{STX}}$.
if $B = 1$ **then**
  Update the local state, i.e., $\vec{\mathtt{st}}_{\mathrm{loc}} \leftarrow \vec{\mathtt{st}}_{\mathrm{loc}} || \mathtt{st}$.
Send (CONTINUE, sid) to $\mathcal{F}_{\mathrm{STX}}$ ▷ Broadcast current state using $\mathcal{F}_{\mathrm{STX}}$.

FetchInformation:

Send (FETCH-NEW, sid) to $\mathcal{F}_{\mathrm{STX}}$.
Denote the response from $\mathcal{F}_{\mathrm{STX}}$ by (FETCH-NEW, sid, ($\vec{\mathtt{st}}_1, \dots, \vec{\mathtt{st}}_k$)).
Set $\vec{\mathtt{st}}_{\mathrm{loc}}$, $\vec{\mathtt{st}}_{\mathrm{exp}}$ to the longest state in $\vec{\mathtt{st}}_{\mathrm{loc}}, \vec{\mathtt{st}}_{\mathrm{exp}}, \vec{\mathtt{st}}_1, \dots, \vec{\mathtt{st}}_k$ (to resolve ties the ordering decides).
[…]

By Lemma 7.1, we know that StateExchange-Protocol UC-realizes $\mathcal{F}_{\mathrm{STX}}$, therefore replacing calls to StateExchange-Protocol by calls to the ideal process $\mathcal{F}_{\mathrm{STX}}$ yields an indistinguishable protocol to Ledger-Protocol. This concludes the proof of the lemma. □

7.4. *Second Proof Step*

We now proof that if honest parties have some advantage over the dishonest parties in winning the lottery, then the UC Bitcoin protocol Modular-Ledger-Protocol$_T$ realizes the ledger functionality. By the composition theorem, we can directly conclude that Ledger-Protocol$_{q,\mathrm{D},T}$ realizes the Bitcoin ledger functionality.

7.4.1. *Relevant Quantities of the Analysis*

The main theorem will require a condition on the power of the adversary, and it is useful to describe here the random variables induced by a pair $(\mathcal{Z}, \mathcal{A})$.

Recall from Sects. 5.2.1 and 5.3.1 that a party is honest-and-synchronized if it either joined at the onset of the execution or it joined a sufficient number of rounds ago (depending on the delay). Furthermore, recall that a logical round consists of two clock-ticks. In the following, we denote the round number by r (which consists of two mini-rounds).

Definition 7.3. (Query Power) We define for the real-world execution of Modular-Ledger-Protocol$_T$ with respect to the pair $(\mathcal{Z}, \mathcal{A})$ the sequence of random variables $Q_H^{(r)}$ to measure the number of distinct honest-and-synchronized parties that are activated in the working mini-round of round r to submit a query to $\mathcal{F}_{\mathrm{STX}}^{\Delta, p_H, p_A}$. Analogously, denote by $Q_A^{(r)}$ the number of submit-queries to $\mathcal{F}_{\mathrm{STX}}^{\Delta, p_H, p_A}$ from corrupted parties in round r, and by $Q_{H,DS}^{(r)}$ the number submit-queries by honest-but-desynchronized parties in the working mini-round of round r.

Definition 7.4. (Mining Power.) We define mining power as simple functions of the query-power. Note that, in our analysis, p_A and p_H are constants. We have:

- The total mining power $\mathtt{T}_{mp}^{(r)} := Q_A^{(r)} \cdot p_A + (Q_H^{(r)} + Q_{H,DS}^{(r)}) \cdot p_H$.
- The adversarial mining power $\beta^{(r)} := p_A \cdot Q_A^{(r)} + p_H \cdot Q_{H,DS}^{(r)}$.
- The honest mining power $\alpha^{(r)} := 1 - (1 - p_H)^{Q_H^{(r)}}$.

It might be useful to recall that from Bernoulli's inequality we have $\alpha^{(r)} \leq p_H \cdot Q_H^{(r)}$. For small values of p_H (as usual in Bitcoin), this upper bound is a good approximation of $\alpha^{(r)}$.

Note that $\alpha^{(r)}$, $\beta^{(r)}$, and $\mathtt{T}_{mp}^{(r)}$ are random variables (on integer domains). For example. $\alpha^{(r)}$ maps the number of honest-and-synchronized submit-queries to the probability that at least one is a successful query. More formally, conditioned on $Q_H^{(r)} = q$, the random variable $\alpha^{(r)}$ is the probability of at least one success among q queries and the expected value of $\alpha^{(r)}$ corresponds to the probability of at least one successful state-extension in round r of the execution. The reason is that $\mathcal{F}_{\mathrm{STX}}^{\Delta, p_H, p_A}$ treats each submit-query independently at random. This is the main motivation to introduce this intermediate step.

7.4.2. *The Analysis*

In the analysis of Bitcoin, conditions are needed that allow to reasonably lower and upper bound expected values of the above random variables (and their variances). As we will quickly recap below, it is shown in [44] that if the involved query power exceeds any limits in the constant-difficulty case, then no security guarantees can be obtained. We start with the following definition.

Definition 7.5. (Query and Mining Pattern) We say that the pair $(\mathcal{Z}, \mathcal{A})$, running for R rounds (referred to by numbers $0, \ldots, R-1$) obeys the query pattern $(\vec{h}, \vec{a}, \vec{d})$ if, for any round r, we have

$$Q_H^{(r)} \geq h_r, \quad Q_A^{(r)} \leq a_r, \quad Q_{H,DS}^{(r)} \leq d_r$$

where $\vec{h} = (h_0, \ldots, h_{R-1})$, $\vec{a} = (a_0, \ldots, a_{R-1})$, $\vec{d} = (d_0, \ldots, d_{R-1})$ are vectors consisting of positive integers. Consequently, the pair $(\mathcal{Z}, \mathcal{A})$ obeys the associated mining pattern denoted by $(\vec{\alpha}, \vec{\beta})$, where vectors $\vec{\alpha} = (\alpha_0, \ldots, \alpha_{R-1})$ and $\vec{\beta} = (\beta_0, \ldots, \beta_{R-1})$ are defined by the mapping

$$\alpha^{(r)} \geq 1 - (1 - p_H)^{h_r} =: \alpha_r$$
$$\beta^{(r)} \leq p_A \cdot a_r + p_H \cdot d_r =: \beta_r.$$

Technically, these definitions imply lower and upper bounds on the expectations of the random variables $\alpha^{(r)}$ and $\beta^{(r)}$, respectively, which is what will be eventually needed.

Definition 7.6. (Power Limits) The pair $(\mathcal{Z}, \mathcal{A})$ is said to be q_{tot}-query-limited if $Q_H^{(r)} + Q_A^{(r)} + Q_{H,DS}^{(r)} \leq q_{tot}$. The pair $(\mathcal{Z}, \mathcal{A})$ is said to be $\mathtt{T}_{mp}$-mining limited if for all r,

$$\mathtt{T}_{mp}^{(r)} \leq \mathtt{T}_{mp}.$$

The bounds in the theorem will depend on several worst-case quantities that we introduce below.

Definition 7.7. For mining patterns $(\vec{\alpha}, \vec{\beta})$, we use the shorthand notation

$$\alpha_{min} := \min\{\alpha_r\}_{r\in[0,R-1]} \quad \text{and} \quad \alpha_{max} := \max\{\alpha_r\}_{r\in[0,R-1]};$$
$$\beta_{min} := \min\{\beta_r\}_{r\in[0,R-1]} \quad \text{and} \quad \beta_{max} := \max\{\beta_r\}_{r\in[0,R-1]}.$$

For a (non-empty) subset $S \subseteq \{0, \dots, R-1\}$ of rounds, we define the corresponding averages by

$$\overline{\alpha}_S := \frac{1}{|S|} \cdot \sum_{r\in S} \alpha_r \quad \text{and} \quad \overline{\beta}_S := \frac{1}{|S|} \cdot \sum_{r\in S} \beta_r.$$

For $\mathtt{T}_{mp}$-mining limited pairs $(\mathcal{Z}, \mathcal{A})$, we define the relative-power fractions

$$\rho_h := \frac{\alpha_{min}}{\mathtt{T}_{mp}} \quad \text{and} \quad \rho_a := \frac{\beta_{min}}{\mathtt{T}_{mp}}.$$

We call a subset S of rounds an interval if it consists of consecutive round numbers $r, \dots, r+t$ for some integers $r, t \geq 0$.

Following [44], the theorem will take into account that the network delay Δ decreases the effectiveness of the actual honest mining power:

Definition 7.8. (Discount function.) We define the function $\gamma(\alpha, \Delta) := \frac{\alpha}{1+\alpha\Delta}$ for $\alpha, \Delta > 0$.

We are now ready to state and prove the main theorem which assures that we can realize the ledger for a given range of parameters.[12]

Theorem 7.9. *Let $p \in (0, 1)$, integer $q \geq 1$, $p_H = 1 - (1-p)^q$, and $p_A = p$. Let $\Delta \geq 1$ be the upper bound on the network delay, let κ be the security parameter, and let $T = \omega(\log \kappa)$ be the main protocol parameter of the ledger protocol. For all pairs $(\mathcal{Z}, \mathcal{A})$ of PPT environments $\mathcal{Z}$ (w.r.t. identity bound ξ_{sync}) and PPT adversaries $\mathcal{A}$ running for R rounds which obey the $(\vec{\alpha}, \vec{\beta})$ mining pattern as of Definition 7.5 and which are $\mathtt{T}_{mp}$-limited as of Definition 7.6, the real-world execution of protocol*

[12] Recall from Sect. 3.3.1 the formal definition of the identity bound ξ_{sync} to model an admissible (lock-step) synchronous execution environment.

Modular-Ledger-Protocol$_T$ *(in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{F}_{\text{STX}}^{\Delta, p_H, p_A}, \mathcal{F}_{\text{N-MC}}^{\Delta})$*-hybrid world) is indistinguishable from the ideal-world execution with ledger functionality* $\mathcal{G}_{\text{LEDGER}}^{B}$ *(and the simulator defined in the proof), if for some* $\lambda > 1$*, it holds that for any interval* S *of rounds of size* $t \geq 1$ *and any* $S' \subseteq S$ *of size* $t' \in [\max\{1, t \cdot (1 - \Delta\alpha_{max})\}, \ldots, t]$ *the relation*

$$\overline{\alpha}_{S'} \cdot (1 - 2 \cdot (\Delta + 1) \cdot T_{mp}) \geq \lambda \cdot \overline{\beta}_S \tag{1}$$

holds, and if the parameters of $\mathcal{G}_{\text{LEDGER}}^{B}$ *fulfill the relations*

$$\texttt{windowSize} = T \quad \text{and} \quad \texttt{Delay} = 4\Delta,$$
$$\frac{(1-\delta)}{2} \cdot \gamma_{min} \geq \frac{\texttt{windowSize}}{\texttt{maxTime}_{\texttt{window}}} \quad \text{and} \quad \frac{(1+\delta)}{2} \cdot \mathrm{T}_{mp} \leq \frac{\texttt{windowSize}}{\texttt{minTime}_{\texttt{window}}},$$
$$\eta \geq \min\{(1+\delta) \cdot \frac{\beta_{max}}{\gamma_{min}},\ 1\},$$

where the quantities are defined as in Definition 7.7 and where $\gamma_{min} := \gamma(\alpha_{min}, \Delta)$*, and* $\delta > 0$ *is an arbitrary constant. In particular, the distinguishing advantage is bounded by* $R \cdot negl(T)$*, where* $negl(T)$ *denotes a negligible function in* T*.*

Remark. Before proving the theorem, it is instructive to recall the flat model of Bitcoin and to see how the above quantities appear there. By the above definitions and theorem statement, we see that we only make statements if the honest mining power is not too small, the dishonest mining power is not too large (and stands in a certain relation to the honest mining power) and if the respective mining power values are in a reasonable range to the overall mining power. In particular, the theorem expresses a condition that the average honest mining power dominates the average mining power of the adversary, even if the honest average is taken over slightly smaller intervals (note that in particular, for each singleton set S, we obtain that the familiar condition that α_r should dominate β_r).

Note that β_{min} is the most restrictive restriction (but not a lower-bound) on the adversary (similarly, α_{max} is the best guaranteed lower-bound for honest-and-synchronous mining power). In general, the adversary (and hence the environment) is free to activate as many ITIs unless it would exceed T_{mp} if the environment is T_{mp}-bounded, and no more than what is allowed by $\vec{\beta}$. This is a more general setting in the fixed-difficulty setting compared to previous works in the same setting. Furthermore, we show in the next subsection how to get a better bound for chain-quality.

Looking ahead, for example in [44], the overall number of parties is fixed to be some number n and there is an upper bound on the number of dishonest parties ρn (and de-synchronized parties are not allowed by definition). Assume for simplicity that $p_H = p_A = p$ for a very small value $p > 0$. We then obtain $\alpha_{min} \approx (1 - \rho) \cdot n \cdot p$ and $\beta_{max} \approx \rho_H \cdot n \cdot p$. By $\mathrm{T}_{mp} = n \cdot p$ and since the mining pattern as defined above is flat in flat models (cf. Sect. 8.1), the correspondence $\rho_a = \rho$ and $\rho_h = (1 - \rho)$ follows.

Also, as pointed out by [44], for too large values of p in a range that would yield $\mathtt{T}_{mp} = n \cdot p > \frac{1}{\Delta}$ (where Δ is the network delay), there is an attack against the protocol, even if one assumes an honest majority. This indicates that the main condition of the theorem in Eq. (1) is also necessary up to a constant factor, and recent works have revealed the exact threshold for security [20,27].

We now prove our main theorem.

Proof of Theorem 7.9. We start with an overview followed by a sequence of claims.

Overview. We prove the theorem using the formalism of [4] to be able to model shared functionalities. In more detail, we specify the simulator $\mathcal{S}_{\text{ledg}}$ as pseudo-code in Appendix C to prove that $\text{EXEC}^{\mathcal{G}_{\text{CLOCK}}}_{\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T,\mathcal{A},\mathcal{Z}} \approx \text{EXEC}^{\mathcal{G}_{\text{CLOCK}}}_{\mathcal{G}_{\text{LEDGER}},\mathcal{S}_{\text{ledg}},\mathcal{Z}}$. Recall from Sect. 2.1.4 that this means that in the real world, parties are running $\mathsf{M}[\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T, \mathcal{G}_{\text{CLOCK}}]$, and in the ideal world the parties are running $\mathsf{M}[\mathcal{G}^{B}_{\text{LEDGER}}, \mathcal{G}_{\text{CLOCK}}]$, where the operator obliviously transforms the defined protocols into standard UC protocols without changing their behavior, and the indistinguishability notion is the standard UC-emulation notion.

Let us explain the general structure of the simulator and the proof: the simulator internally runs the round-based mining procedure of every honest party. Whenever a working mini-round is over, i.e., whenever the real world parties have issued their queries to $\mathcal{F}_{\text{STX}}$, then the simulator will assemble the views of its simulated honest-and-synchronized miners and determine their common prefix of states, which is the longest state stored or received by each simulated party when chopping off T blocks. The adversary will then propose a new block candidate, i.e., a list of transactions, to the ledger to announce that the common prefix has increased (procedure EXTENDLEDGERSTATE). The ledger will apply the Blockify on this list of transactions and add it to the state. Note that since Blockify does not depend on time, the current time of the ledger has no influence on this output. To reflect that not all parties have the same view on this common prefix, the simulator can adjust the state pointers accordingly (procedure ADJUSTVIEW). The simulation inside the simulator is perfect and is simply the emulation of real-world processes. What restricts a perfect simulation is the requirement of a consistent prefix and the restrictions imposed by ExtendPolicy. In order to show that these restrictions are not forbidding a proper simulation, we have to justify, why the choice of the parameters in the theorem are sufficient to guarantee that (except with negligible probability). To this end, we analyze the real-world execution to bound the corresponding bad events that prevent a perfect simulation.

We basically follow the proof ideas of Pass, Seeman, and shelat [44] to bound the bad events and adapt their observations to our setting. The analysis is divided into several different claims about the real-world execution. They include properties such as a lower-bound on the chain growth, the chain quality, or an upper-bound on the chain growth. These claims prove that our simulator can simulate the real-world view perfectly, since the restrictions imposed by the ledger prohibit that only with negligible probability, where the distinguishing advantage is upper bounded by $R \cdot \text{negl}(T)$, where R denotes the number of rounds the protocol is running and $\text{negl}(\cdot)$ denotes a negligible function in the parameter T.

Recall that each round consists of two time-ticks. Hence, if a statement is expressed with respect to a certain number t of rounds, it can equivalently be expressed with respect to $2t$ clock-ticks. Recall that the ledger parameters have to be given with respect to the clock, since the clock is the formal reference point of time. However, for the analysis, it is easier to think in rounds. In the following sections, if we refer to an interval $r, \ldots, r+t$, this refers to t *full* rounds, i.e., the time window when the clock first switched to the value $\tau = 2r$ up to any point where the clock value satisfies $\tau \in \{2(r+t), 2(r+t)+1\}$.

Chain dissemination. We first state an obvious useful fact about the protocol's operation.

Lemma 7.10. (State dissemination) *Let P_i and P_j be miners, and let $r \geq 0$. Assume P_i is honest in round r, and its adopted state has length ℓ. For any honest miner P_j in round $r + \Delta$ who registered to the network before round r, it holds that its adopted state must have at least length ℓ.*

Proof. By assumption, all messages, and in particular transmitted states of honest miners, are delayed maximally by Δ rounds. Thus, if such a miner receives a state of length ℓ, then any other honest miner will receive this state within the next Δ rounds since the protocol relays its adopted state. Additionally, if an honest miner successfully extends a ledger-state in round r, the new state is fetched by other honest miner at latest after Δ rounds if they were registered before round r. Hence by then, they will have adopted a chain of length at least ℓ. $\square$

Probably the most useful corollary which is used in the sequel, is to apply the above lemma to the sub-class of honest-and-synchronized miners. Note that if P_j in the above lemma is honest-and-synchronized at round $r + \Delta$ it must have been registered to the network not later than at round $\max\{0, r - \Delta\}$ and hence the statement applies.

Analyzing chain growth. We now state the relation between time (measured in number of rounds) and guaranteed number of new state blocks.

Lemma 7.11. (Chain growth) *Consider the real-world execution (under the conditions of the theorem). Let P_i be a miner, and let $r \geq 0$. Assume P_i is honest-and-synchronized in round r, and the (longest) state adopted by P_i in round r has length ℓ. Then, in round $r + t$, it holds that for any $\delta > 0$, except with probability $R \cdot negl(T)$, the length of the (longest) state adopted of any honest-and-synchronized miner P_j in that round has length at least $\ell + T$ if $t \geq \frac{T}{(1-\delta)\cdot\gamma_{min}}$.*

More generally, for an interval of rounds $r, \ldots, r+t$, we can guarantee a length increase of $\gamma \cdot t$ with $\gamma := \frac{\tau}{1+\tau\Delta}$ if for all possible subsets S of rounds of size $t' = t(1-\gamma\Delta)$ of this interval we have $\overline{\alpha}_S \geq \tau$. The guarantee holds except with probability $\exp(-\Omega(t\gamma))$.

Proof. We first prove that for any real-world adversary $\mathcal{A}$, there is an adversary $\mathcal{A}'$ that, starting at the given round r, maximally delays messages and prove that in a real-world execution with $\mathcal{A}'$ the expected state length of an honest-and-synchronized miner in round $r + t$, where the expectation is taken over the randomness of the adversarial strategy, is no larger than with adversary $\mathcal{A}$ in round $r + t$. Given adversary $\mathcal{A}$, the adversary $\mathcal{A}'$ works as follows. It internally runs $\mathcal{A}$ until and including round r without any modifications. After round r, $\mathcal{A}'$ first delays all current messages in the network to the maximally possible delay. Also, after round r, whenever an honest-and-synchronized party sends a message containing a state, $\mathcal{A}'$ sets the maximal delay Δ for this message. Message delays defined by $\mathcal{A}$ for messages that contain valid states of honest parties are ignored. The adversary further ignores any message sent by $\mathcal{A}$ on behalf of corrupted parties after round r.

We define the following "hybrid world", which equals the real world execution, but with fixed randomness as follows: for random strings σ, σ', we define $\text{HYB}_{\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{A}(\sigma),\mathcal{Z}}$ to be defined analogously to $\text{EXEC}_{\cdot,\cdot,\mathcal{Z}}$ but where the internal coins of $\mathcal{A}$ and $\mathcal{F}_{\text{STX}}$ are fixed to σ and σ' respectively (note that both are poly-bounded by the run-time restrictions of UC). Let $T^{\text{hyb}}_{\mathcal{A}(\sigma),\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{Z}}$ be the associated distribution of the joint view (induced by the random coins of $\mathcal{Z}$). Let $\text{Len}_i^r(T)$ be the function that maps a transcript T (of real-world and hybrid-world executions) to the length of the (longest) adopted chain by (honest-and-synchronized) miner i in round r.

We first give an inductive proof to show that for any $r > 0$, and all strings σ, σ',

$$\Pr_{\sigma_Z \in_R \{0,1\}^{\mathsf{poly}(\kappa)}}[\text{Len}_i^{r+t}(T^{\text{hyb}}_{\mathcal{A}(\sigma),\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{Z}(\sigma_Z)}) \geq \text{Len}_i^{r+t}(T^{\text{hyb}}_{\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{Z}(\sigma_Z)})] = 1.$$

$\square$

Base Case(s): We give the base cases $t = 0$ and $= 1$ to already include the arguments for the general case below. We argue for any fixed σ_Z and show that the condition in the event cannot be violated. Since adversary $\mathcal{A}$ and $\mathcal{A}'$ behave identical up to and including round r, the length of the longest state known or received by any party is the same. The reason is that $\mathcal{A}'$ and $\mathcal{A}$ play exactly the same strategy when the randomness is fixed, since $\mathcal{A}'$ itself does not use additional random coins and thus case $t = 0$ follows. Furthermore, when the randomness σ' of $\mathcal{F}_{\text{STX}}$ is fixed, a miner i in any round r' is successful, if and only if it is successful in round r' with adversary $\mathcal{A}'$. Thus, the condition for $t = 1$ would only be violated if player i receives a longer state in round $r + 1$. However, since $\mathcal{A}'$ maximally delays messages, if any state arrives in round $r + 1$ in the real execution with $\mathcal{A}'$, then it arrives no later than $r + 1$ in the real execution with $\mathcal{A}$. This concludes the base cases.

Induction Step: $t \to t + 1$: By the induction hypothesis, we have that the condition

$$\text{Len}_i^{r+t}(T^{\text{hyb}}_{\mathcal{A}(\sigma),\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{Z}(\sigma_Z)}) \geq \text{Len}_i^{r+t}(T^{\text{hyb}}_{\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{Z}(\sigma_Z)})$$

holds with probability one. We argue that $\text{Len}_i^{r+t+1}(\cdot) \geq \text{Len}_i^{r+t+1}(\cdot)$ holds as well (on the above arguments) with probability one. Assume this was not the case, then following the above reasoning, it can only be due to miner i receiving a state in round $r + t + 1$ that would increase the value of $\text{Len}_i^{r+t+1}(T^{\text{hyb}}_{\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{Z}(\sigma_Z)})$ but not the value of $\text{Len}_i^{r+t+1}(T^{\text{hyb}}_{\mathcal{A}(\sigma),\mathcal{F}_{\text{STX}}(\sigma'),\mathcal{Z}(\sigma_Z)})$ (since the success of miner i in round $r + t + 1$ is fixed given σ'). By the same reasoning as above, since $\mathcal{A}'$ maximally delays delivery of new states, if any state arrives in round r in the real execution with $\mathcal{A}'$, then it arrives no later than r in the real execution with $\mathcal{A}$. This concludes the induction proof.

We note that the hybrid world, if we sample σ, σ' this yields the distribution $T_{\mathrm{EXEC}_{\pi,\mathcal{A}',\mathcal{Z}}}(\kappa, z)$ (for any fixed input z to the environment). Let us abbreviate this by $T_{\mathrm{real},\mathcal{A}'}$ to save on notation (and assuming the input z is hard-coded in the environment). Similarly, let us denote $T_{\mathrm{real},\mathcal{A}}$ the distribution in an execution with $\mathcal{A}$.

By taking the expectation over σ, σ' (and by the law of total probability), we immediately get from the above arguments that for any positive integer c and any round r:

$$\begin{aligned}&\Pr[\mathrm{Len}_i^{r+t}(T_{\mathrm{real},\mathcal{A}}) \leq \mathrm{Len}_i^{r}(T_{\mathrm{real},\mathcal{A}}) + c]\\ &\leq \Pr[\mathrm{Len}_i^{r+t}(T_{\mathrm{real},\mathcal{A}'}) \leq \mathrm{Len}_i^{r}(T_{\mathrm{real},\mathcal{A}'}) + c]\end{aligned}$$

where we also used that for $t = 0$, the length distributions induced by $\mathcal{A}$ and $\mathcal{A}'$ are identical. Hence, chain growth can be analyzed w.r.t. adversary $\mathcal{A}'$ to yield a useful statement for any adversary $\mathcal{A}$.

Let us use the following terminology: We say a round r' is *uniform* if $\mathrm{Len}_i^{r'}(tr) = \mathrm{Len}_j^{r'}(tr)$ holds (where tr is a transcript), for all honest-and-synchronized miners i and j. Recall that adversary $\mathcal{A}'$ does not broadcast adversarially generated states and any new state is delayed by exactly Δ rounds. The slowest progress of the overall maximal state length known to an honest-and-synchronized party occurs in case uniform rounds are the only successful rounds (if at all). Otherwise, the honest miner with the longest state could be successful and broadcast a longer state at round r', which would be guaranteed to arrive to any other honest miner in $r + \Delta$. Furthermore, by a standard coupling argument, the probability of success of any honest-and-synchronized party in some round r' is minimized by an environment $\mathcal{Z}$ that activates just enough parties to obey the mining pattern $\alpha_{r'}$. The coupling with any other environment can be obtained by letting the activation results be the same up to the point where enough parties have been activated to satisfy the mining pattern. Further activations honest-and-synchronized participants can only induce more successful state extension than what $\mathcal{Z}$ obtained.

We are thus left with analyzing growth w.r.t. a simple adversary and an environment $\mathcal{Z}$ with a fixed activation pattern per round to match the mining pattern.

Obtaining a tail bound depending on number of blocks. Now, fix some round r. If in round $s = r + t$, the length increase of the overall longest state of an honest-and-synchronized miner is less than c blocks, then at most $c \cdot \Delta$ non-uniform rounds occurred. According to above, we can associate to each round i a random variable X_i which is 1 if at least one honest-and-synchronized miner successfully extended the state by a query to $\mathcal{F}_{\mathrm{STX}}$. The X_i's are independent by construction and there must be at least $t - c \cdot \Delta$ uniform rounds. On the other hand, for any concrete sub-sequence of rounds $S \subset (r, \ldots, r + t)$ of size t', the Chernoff-Hoeffding bound in Theorem 2.3 implies for our setting (of independent heterogeneous variables) that

$$\Pr\left[\sum_{i \in S} X_i \leq (1 - \delta) \cdot \overline{\alpha}_S \cdot t'\right] \leq \exp(-\Omega(\overline{\alpha}_S \cdot t')), \tag{2}$$

where $\overline{\alpha}_S := \frac{1}{t'} \sum_{i \in S} \alpha_i$.

We conclude that if for the sub-sequence S of rounds in the interval from r to s, the relations $c = \mathbb{E}\left[\sum_{i \in S} X_i\right] = \overline{\alpha}_S \cdot t'$ and $|S| =: t' = t - c\Delta$ hold, we can derive a tail-estimate depending on the number of blocks. We can define

$$c_S := \frac{\overline{\alpha}_S t}{1 + \overline{\alpha}_S \Delta}$$

and assign a corresponding growth coefficient

$$\gamma_S := \frac{\overline{\alpha}_S}{1 + \overline{\alpha}_S \Delta}.$$

and thus except with exponentially small probability in $t\gamma_S = c_S$, the length-increase is at least c_S for this particular interval.

For the first part of the statement, observe that $\overline{\alpha}_S \geq \alpha_{min}$, for all subsets S, and that the function $\frac{x}{1+kx}$, where k is a positive integer and $x \in (0, 1)$, is monotone in x. We get the guaranteed minimal growth by $t \cdot \gamma_{min}$ in any interval of size t rounds for an honest-and-synchronized party except with negligible probability in $t \cdot \gamma_{min}$ by taking the union bound overall all rounds r. What remains to prove is that this bound applies also to the growth of the state if one compares any two honest-and-synchronized miners which we do below (still following the proof steps of [44]).

For the second part of the statement, we generalize the above observation: if we have a guaranteed lower bound τ on the average $\overline{\alpha}_S$ (better than α_{min} as used before) with respect to *any* subset of the required size within the given interval $r, \ldots, r + t$ (note that indeed we only have a bound for the size of S in our experiments but no guarantee that a particularly "good" one is chosen), the second part of the statement follows.

Bound for any honest-and-synchronized party. By Lemma 7.10, we know that if an honest-and-synchronized miner knows some state, then within Δ rounds, every other honest miner will be aware of that state. A similar calculation shows that the lower bound on the time to have a state increase by T blocks by all honest-and-synchronized parties follows the same law (and hence the perceived ledger speed is the same). By requiring $s = r + t - \Delta$ above, and thus considering $t' := t - \Delta - c \cdot \Delta = t - (c + 1)\Delta$ does not change the asymptotic behavior since $\gamma_S t - 1 < \gamma_S t - \gamma_S \Delta < \gamma_S t$ for all t and S since $\Delta\gamma_S < 1$. Hence, this additional additive term can be compensated by choosing a sufficiently small constant δ in Eq. (2). $\square$

Mining limits. We state some helpful facts about bounds on the mining behavior.

Lemma 7.12. *The number of successful state-extensions that happen with $\mathcal{F}_{\text{STX}}^{\Delta, p_H, p_A}$ in any given interval of t rounds (in the real-world execution under the theorem conditions), where $p_A = p$ and $p_H = 1 - (1 - p)^q$ for some $q \geq 1$ and $p \in (0, 1)$ is bounded by $(1 + \delta) \cdot t \cdot \mathcal{T}_{mp}$ for any $\delta > 0$, except with probability negl$(\mathcal{T}_{mp} \cdot t)$. Consequently, for a number T of state-extensions to occur, the number of required rounds is less than $\frac{T}{(1+\delta)\mathcal{T}_{mp}}$ only with negligible probability in T. Finally, the number of adversarial state extensions in a sub-set S of t rounds is no more than $(1 + \delta)\overline{\beta}_S \cdot t$ except with probability $\exp(-\Omega(\overline{\beta}_S \cdot t))$ (for any $\delta > 0$.*

Proof. Since the state-exchange functionality evaluates each query independently, we can upper bound the number of successes of these independent Bernoulli-trials. We prove the bound for the environment $\mathcal{Z}$ (and $\mathcal{A}$) that makes as many queries as allowed per round (as limited by β_r and $\mathtt{T}_{mp}$). As in the previous lemma, a coupling argument shows that any other query-distribution cannot induce a larger probability exceeding the given bound than $\mathcal{Z}$, for which the query distribution is fixed. For a round, let $X^{(r)} = \sum_i X_i$ model the sum of the involved independent trials to the state-exchange functionality. Clearly, $\beta_r \leq \mathbb{E}[X^{(r)}] \leq \mathtt{T}_{mp}$. Let S be a set of t rounds. By linearity of expectation and invoking Theorem 2.3 we get the tail-estimate

$$\Pr\left[\sum_{i \in S} X^{(i)} \geq (1+\delta) \cdot t \cdot \mathtt{T}_{mp}\right] \leq \exp(-\Omega(\overline{\beta}_S \cdot t)) \\ \leq \exp(-\Omega(\mathtt{T}_{mp} \cdot t)),$$

where the last step invokes the theorem assumption that $\forall r : \beta_r \geq \rho_a \mathtt{T}_{mp}$ for the relative-power coefficient ρ_a.

Similarly, denote by $Y^{(r)} = \sum_i Y_i$ the number of adversarial state-extensions in round r. Again it is sufficient to consider a maximizing $\mathcal{Z}$ which has an expected value of $t \cdot \overline{\beta}_S$ over a sub-set of rounds of size t. Hence, we again can obtain an estimate of the form

$$\Pr\left[\sum_{i \in S} Y^{(i)} \geq (1+\delta) \cdot t \cdot \overline{\beta}_S\right] \leq \exp(-\Omega(\overline{\beta}_S \cdot t)).$$

As a final conclusion, we observe that for any number of state blocks T, the probability that for any $\delta > 0$ it takes less than $t = \frac{T}{(1+\delta)\mathtt{T}_{mp}}$ rounds to get T state extensions is negligible in T. Consequently, for this large time interval, all tail bounds hold except with probability $\exp(-\Omega(T))$, where the constant hidden in $\Omega(\cdot)$ depend on δ and on the relative-power coefficient ρ_a. $\square$

Block withholding. From chain growth and the theorem's condition, we derive that if an honest-and-synchronized miner adopts a new state that contains a block the adversary obtained by $\mathcal{F}_{\mathrm{STX}}$ then either this block has been published by the adversary before, or it was mined quite recently by a corrupted party.

Lemma 7.13. (Bound on Withholding strategies) *In the real-world execution (under the conditions of the theorem), assume that in round r, an honest-and-synchronized miner adopts a new longer state* `state`*. Assume there is a block* `st` *in this new state that was accepted upon an adversarial query to $\mathcal{F}_{\mathrm{STX}}$ and that is not part of any state adopted by any honest-and-synchronized party before round r. The probability that such a block* `st` *was first accepted by $\mathcal{F}_{\mathrm{STX}}$ before round $r - \omega t$ happens only with probability negl$(\overline{\beta}_S \cdot t)$, for any constant $0 < \omega < 1$, where S denotes the interval $r - \omega t, \ldots, r$.*

Proof. Let us define $\vec{\mathtt{st}}^{(r)} = \mathtt{st}_0 || \ldots || \mathtt{st}_k$ to be the state adopted by the honest-and-synchronized miner in round r as assumed in the lemma statement. Let $\vec{\mathtt{st}}^{(r')}$ be the longest prefix of $\vec{\mathtt{st}}^{(r)}$ such that $\vec{\mathtt{st}}^{(r')}$ is either the genesis block or a state newly accepted by $\mathcal{F}_{\mathrm{STX}}$ upon a query by an honest-and-synchronized party in round $r' \leq r$. Hence all the blocks in that prefix are known to at least one honest-and-synchronized party by round r'. In light of the lemma statement, we consider the case that $r - r' \geq \omega t$.

Let S denote the set of rounds from r' to r. The number of new states mined by the adversary does not exceed $(1+\delta') \cdot \overline{\beta}_S \omega t$ (except with probability $\mathrm{negl}(\overline{\beta}_S \cdot t)$) by the previous lemma.

At the same time, Eq. (1) implies that on any subset S' of size $t' = \omega t(1 - \alpha_{max}\Delta)$ the condition $\overline{\alpha}_{S'}(1 - \Delta\overline{\alpha}_{S'}) \geq (1+\delta)\overline{\beta}_S$ has to hold for some constant $\delta \in (0, 1)$. This is the case since for all $x, \Delta > 0$, $\frac{x}{1+x\Delta} > x(1 - x\Delta)$ (and $\mathtt{T}_{mp} \geq \overline{\alpha}_{S'}$) and this implies that $\gamma := \frac{\overline{\alpha}_{S'}}{1+\overline{\alpha}_{S'}\Delta} \geq (1+\delta)\overline{\beta}_S$. Lemma 7.11 gives us a chain growth of $|\vec{\mathtt{st}}^{(r)}| - |\vec{\mathtt{st}}^{(r')}| \geq (1-\delta') \cdot \gamma\omega t$ except with probability $\mathrm{negl}(\overline{\beta}_S \cdot t)$.

Since all $|\vec{\mathtt{st}}^{(r)}| - |\vec{\mathtt{st}}^{(r')}|$ blocks must have been mined by the adversary, we have $|\vec{\mathtt{st}}^{(r)}| - |\vec{\mathtt{st}}^{(r')}| \leq (1+\delta'') \cdot \overline{\beta}_S \omega t$. We get a contradiction, since now

$$(1-\delta') \cdot \gamma\omega t \leq (1+\delta'') \cdot \overline{\beta}_S \cdot \omega t,$$

which, for sufficiently small δ', δ'' would imply that $\gamma < (1+\delta)\overline{\beta}_S$. □

Chain-growth upper-bound. Our ledger also restricts the growth over time. This is based on the following observation.

Lemma 7.14. (Chain-Growth Upperbound) *Consider the real-world execution (under the conditions of the theorem) and let P_i be a miner, and let $r \geq 0$. Assume P_i is honest-and-synchronized in round r, and the longest state received or stored by P_i in round r has length ℓ. Then, in round $r+t$, it holds, except with probability $R \cdot \mathrm{negl}(T)$, that the length of the longest state (received or stored) of at least one honest-and-synchronized miner P_j in that round has length at most $\ell + T$ if $t \leq \frac{T}{(1+\delta)\cdot\mathtt{T}_{mp}}$ for any $\delta > 0$.*

Proof. We can combine the previous observations to upper bound the number of accepted blocks. By Lemma 7.12, the number of rounds to generate T new extensions of states is at least $t' \geq \frac{T}{(1+\delta')\mathtt{T}_{mp}}$ except with probability $\mathrm{negl}(T)$ (for any $\delta' > 0$) and thus with overwhelming probability, in $t' \leq \frac{T}{(1+\delta')\mathtt{T}_{mp}}$, no more than T new blocks are mined.

In addition, we can invoke Lemma 7.13 to conclude that a new state that contains a block that the adversary is withholding since a round prior to $r - \omega t$ is accepted by an honest-and-synchronized party only with probability $\mathrm{negl}(\beta_{min} t)$, for any $0 < \omega < 1$ (since β_{min} can be achieved in any round by an adversarial strategy and hence can serve as the lower bound in the exponent of the tail bound). Analogously to Lemma 7.12, by the definition $\rho_a \cdot \mathtt{T}_{mp} = \beta_{min}$ this error probability is thus negligible in T.

Both observations together imply that in $t' = t(1+\omega) \leq \frac{T}{(1+\delta')\mathtt{T}_{mp}}$ rounds, no honest-and-synchronized party experiences a state increase of more than T blocks for any δ' except with negligible probability in T. This is equivalent to the condition that $t \leq \frac{T}{(1+\omega)(1+\delta')\mathtt{T}_{mp}}$ and we can choose δ' sufficiently small to obtain the bound with respect to $t \leq \frac{T}{(1+\delta)\mathtt{T}_{mp}}$ and any given $\delta > 0$ as required by the statement. The claim follows by taking the union bound over all rounds as the arguments above hold for any round r. □

Worst-case chain quality. We give a very coarse bound on the overall chain quality in any sequence of T blocks as follows:

Lemma 7.15. (Fraction of honest blocks) *Let P_i be a miner, and let $r \geq 0$. Assume P_i is honest-and-synchronized in round r and that the length of the longest state received or stored is $\ell \geq T$. The fraction of adversarially mined blocks within a sequence of T blocks in the state is at most* $\min\{1, (1+\delta) \cdot \frac{\beta_{max}}{\gamma_{min}}\}$ *except with probability $R \cdot negl(T)$ for any $\delta > 0$.*

Proof. Let us assume that at round r, the state adopted by miner P_i is $\vec{\mathtt{st}}_{r'} = \mathtt{st}_0||\ldots||\mathtt{st}_k$. We show that in any sub-sequence of T state blocks $\mathtt{st}_{j+1}, \ldots, \mathtt{st}_{j+T}$ in $\vec{\mathtt{st}}_r$, the fraction of adversarially mined blocks is bounded. Without loss of generality, one can assume that the state $\vec{\mathtt{st}}^{<j} := \mathtt{st}_0||\ldots||\mathtt{st}_j$ as well as the state $\vec{\mathtt{st}}^{>j+T} := \mathtt{st}_0||\ldots||\mathtt{st}_{j+T+1}$ are mined by honest-and-synchronized miners (or $j+T$ equals the length of the state). Otherwise, one can enlarge T to meet this condition — this can only increase the fraction of adversarial blocks in the sequence of T blocks and since any state is finite and starts with the genesis block, the condition will be fulfilled for some T. We further assume that $\vec{\mathtt{st}}^{<j}$ is mined at round r', and that in round $r'+t$, the state $\vec{\mathtt{st}}^{>j+T}$ appears for the first time as the state, or the prefix of a state, of at least one honest-and-synchronized miner. We conclude that if an adversary successfully extended the state during some round by a new state block $\mathtt{st}_{j+s}$ of the above sequence $\mathtt{st}_{j+1}, \ldots, \mathtt{st}_{j+T}$, then this happens in a round between r' and $r'+t$.

We now relate the number t of rounds to the number T of blocks. Since t is assumed to be the minimal number of rounds until the first honest-and-synchronized miner adopted a state containing $\mathtt{st}_{j+1}$, we can make use of the minimal chain-growth Lemma 7.11 to conclude that the probability that the condition $t > \frac{T}{(1-\delta')\gamma_{min}}$ occurs in such an execution is at most $\mathrm{negl}(T)$. We hence have $t \leq \frac{T}{(1-\delta')\gamma_{min}}$ with overwhelming probability in T.

Similar to above, by Lemma 7.12 the time it takes to generate T blocks is at least $t \geq \frac{T}{(1+\delta)\mathtt{T}_{mp}}$ except with probability $\mathrm{negl}(T)$ and thus with overwhelming probability, in $t \leq \frac{T}{(1+\delta)\mathtt{T}_{mp}}$, no more than T blocks are mined.

Furthermore, also by Lemma 7.12, we get a worst-case upper bound. Let N_A^t denote the expected value in t rounds, invoking Lemma 7.12 gives us that $N_A^t \leq (1+\delta)\beta_{max} t$ except with probability $\mathrm{negl}(\beta_{min} t)$ (where we again use the minimum to bound the average of any interval). Hence, since $\rho_a \cdot \mathtt{T}_{mp} = \beta_{min}$ by definition it follows as in previous lemmata that the bound holds except with probability $\mathrm{negl}(T)$.

Putting things together, we conclude that except with negligible probability in T, the number of times the adversary was successful in extending the state by one block is upper bounded by the quantity

$$N_A^{\frac{T}{(1-\delta')\gamma}} \leq \frac{1+\delta}{1-\delta'} \cdot T \cdot \frac{\beta_{max}}{\gamma_{min}}.$$

Hence, the fraction of adversarial blocks within T consecutive blocks cannot be more than $f = \min\{1, (1+\delta'')\frac{\beta_{max}}{\gamma_{min}}\}$ for any δ'' and sufficiently small constants $\delta, \delta' > 0$, except with negligible probability in the length T of the sequence.

Since our arguments hold for any interval, the proof is concluded by taking the union bound over the number of such sequences (which is in the order of number of rounds). □

Consistency (common prefix). We now state the lemma on the common-prefix property in our setting.

Lemma 7.16. (Consistent states) *Consider the real-world execution under the condition of the main theorem. Let P_i and P_j be miners (potentially the same), and let $r' \geq r \geq 0$. Assume P_i is honest-and-synchronized in round r, and P_j is honest-and-synchronized in round r'. Assume that the length of the longest state received or stored by P_i in round r is $\ell \geq T$. Then, the $\ell - T$-prefix of that longest state of P_i in round r is identical to the $\ell - T$-prefix of the state of P_j stored or received in round r' except with probability $R \cdot negl(T)$.*

Proof. We again follow the basic line of reasoning in [44] and adapt the appropriate arguments to our setting. First, since an inconsistency at round r implies an inconsistency at round $r' > r$, if the claim is proven for the case $r \leq r' \leq r+1$, then by an inductive argument, the claim holds for any $r' \geq r$.

The protocol mandates that the honest-and-synchronized miners truncates the T newest blocks from the current respective state. Thus, we need to argue that the block which is $T+1$ far away from the head will be part of any state output by any honest-and-synchronized miner. Suppose we are at round r' in the protocol, then the time it takes to generate the last T blocks is at least $t \geq \frac{T}{(1+\delta)\mathrm{T}_{mp}}$ except with negligible probability in T as established in Lemma 7.12 and any $0 < \delta < 1$.

Looking ahead, we will eventually conclude that with overwhelming probability within the interval of rounds $s = r - t, \ldots, r' \in \{r, r+1\}$ (where $r \geq t$), the honest-and-synchronized miners have an opportunity to agree on a common state and hence at round r', they will still agree on a large common prefix of the current state at round r'.

In the interval of rounds, let this set be denoted as usual by S, between round s and round $r' = r$, the expected number of rounds, where at lest one honest-and-synchronized miner is successful, is at least $\overline{\alpha}_S t$. Thus, again by a standard Chernoff bound, the probability that the number of these successful rounds is smaller than $\bar{q}_{min} := (1-\delta') \cdot \overline{\alpha}_S t$ is no more than $\exp(-\Omega(t\overline{\alpha}_S))$ in the real-world UC random experiment. Again, a coupling argument as in Lemma 7.11 yields that this tail-bound (where the environment activates the least number of parties possible and hence the random variables that describe the success are independent) applies to any environment. Finally, the conditions of the theorem in particular assure that $\overline{\alpha}_S > \beta_{min}$ and hence this probability can be upper bounded by $\mathrm{negl}(\beta_{min} t)$.

Unfortunately, the “race” between the good guys and the bad guys is not yet conclusively analyzed, since the mere superiority of honestly mined blocks does not imply that the honest parties will reach agreement. In particular, not all of the expected honestly mined blocks are equally useful to obtain a so-called convergence opportunity. In particular, we need to know how many of the honestly mined blocks happen in isolated, sufficiently silent intervals.

Formally, let us introduce the random variable R_i that measures the number of elapsed round between successful round $i-1$ and successful round i in the real-world UC execution, where R_1 measures the number of elapsed rounds to the first successful round. Based on R_i, the random variable X_i is defined as follows: $X_i = 1$ if and only if $R_i > \Delta$ and exactly one honest-and-synchronized miner mines a new state (i.e., successfully appends a new block to the state) in the ith successful round.

Let E_1^i be the event that there is at least one successful round in the interval of Δ rounds starting after successful round $i-1$ (or at the onset of the experiment). Let E_2^i be the event that strictly more than one miner is successful in the following successful round i.

Overall, our goal is to suitably bound the number of blocks that prevent those events of "success & silence" (i.e., bound the probability of the event $X_i = 0$) in an interval of t rounds. We call these the undesirable blocks. They have to be infrequent enough such that in combination with adversarially mined blocks, they do not prevent too many convergence opportunities. We hence need to suitably bound the occurrence of the above two bad events E_j^i in our experiment.

By a union bound, and invoking that $\alpha_r \le \mathtt{T}_{mp}$, we directly have that $\Pr[X_i = 0] = \Pr[E_1^i \cup E_2^i] \le \Delta \mathtt{T}_{mp} + \mathtt{T}_{mp}$, hence, on the positive side, $\Pr[X_i = 1] \ge 1 - \mathtt{T}_{mp}(\Delta + 1)$.

Let $X := \sum_{i=1}^{\bar{q}_{min}} X_i$, and let us define $\bar{q}'_{min} := (1-\delta'') \cdot (1 - \mathtt{T}_{mp}(\Delta+1)) \cdot \bar{q}_{min}$. Since by Eq. (1) the term $1 - 2(\Delta+1)\mathtt{T}_{mp}$ must be positive, we have that $\mathtt{T}_{mp}(\Delta+1) \le \frac{1}{2}$ and, because $\mathcal{F}_{\mathrm{STX}}$ treats each new state-submission independently of previous submission, we conclude that $\Pr[X_i = 1 \mid X_1, \ldots, X_{i-1}] \ge \frac{1}{2}$. Since we do not argue here about any particular optimal strategy by an environment-adversary pair $(\mathcal{Z}, \mathcal{A})$, we need to invoke Lemma 7.17 from which we get

$$\Pr[X \le \bar{q}'_{min}] \le \exp\left(-(\delta'')^2 \bar{q}_{min}/2\right). \tag{3}$$

To express this w.r.t. β_{min}, observe that not only $\alpha_r > \beta_r$ (and thus $\alpha_{min} > \beta_{min}$) by Eq. (1) but also there is an actual constant $0 < \widehat{\delta} < 1$ such that $\mathtt{T}_{mp}(\Delta+1) < 1 - \widehat{\delta}$. This is true since by the theorem condition we deduce that

$$\begin{aligned} &(1 - 2(\Delta+1)\mathtt{T}_{mp}) \ge \lambda(\beta_{min}/\alpha_{min}) \\ &\implies 1 - \lambda(\beta_{min}/\alpha_{min}) \ge 2(\Delta+1)\mathtt{T}_{mp} > (\Delta+1)\mathtt{T}_{mp}. \end{aligned}$$

And since $\lambda > 1$, i.e., we get can bound the constant by $0 < \widehat{\delta} < \lambda(\beta_{min}/\alpha_{min})$ and obtain

$$(1 - \mathtt{T}_{mp}(\Delta+1)) \cdot \bar{q}_{min} > \widehat{\delta}(1-\delta') \cdot \overline{\alpha}_S t > \widehat{\delta}(1-\delta') \cdot \overline{\beta_S} t.$$

And hence conclude by Eq. (3) that $\Pr[X \le \bar{q}'_{min}] \le \exp(-\Omega(\beta_{min} t))$. We thus have a (high-probability) lower bound on the number of silent patterns.

We are actually interested in the number of times that $X_i = X_{i+1} = 1$. This situation, as already introduced above, means that we have a situation, in which for Δ rounds, no miner is successful, then exactly one honest-and-synchronized miner is successful, and afterward, we again have Δ rounds of silence. This is denoted in [44] as a *convergence opportunity*. For example, a convergence opportunity has the desirable property, that *at the end* of such an opportunity, if the adversary is unable to provide a longer state to the honest-and-synchronized miners during this period, all honest-and-synchronized miners will reach an agreement on the current longest state. Thus, in order to prevent this, an adversary needs to be successful in mining roughly at the rate of the number of convergence opportunities within t rounds.

We have already seen that with overwhelming probability, there are at least $\bar{q}_{min}$ successful rounds, and among which $(\bar{q}_{min} - \bar{q}'_{min})$ can disturb convergence opportunities. Since a single disturbing round can at most prevent two convergence opportunities (it violates the condition for a convergence opportunity with its neighbors in the sequence $X_1, \ldots, X_k$), the number of effective convergence opportunities C is lower bounded (except with negligible probability) by

$$\begin{aligned} C &\geq \bar{q}_{min} - 2(\bar{q}_{min} - \bar{q}'_{min}) = 2\bar{q}'_{min} - \bar{q}_{min} \\ &\geq (1-\delta')\overline{\alpha}_S t[1 - 2\mathrm{T}_{mp}(\Delta+1) - 2\delta'']. \end{aligned}$$

For any constant ϵ, by picking δ' and δ'' sufficiently small, this yields a bound (except with negligible probability as derived above) of

$$C > (1-\epsilon)(1 - 2\mathrm{T}_{mp}(\Delta+1))\overline{\alpha}_S t.$$

The final argument is a counting argument. Let us denote by $\mathcal{S}_{r'}$ the set of maximal states known to $\mathcal{F}_{\text{STX}}$ at round r' (i.e., any path from the root to some a leaf) of length at least $\ell + C$, where ℓ is the length of the longest state known to at least one honest-and-synchronized miner at round s. Note that $\mathcal{S}_{r'}^{\ell+C}$ is non-empty: since each convergence opportunity increases the length by at least one, and before each successful round, there is a period of Δ rounds where no honest miner mines a new state, there has to exist at least one state with length at least $\ell + C$ at round r'.

Assume that the number of successful state extensions made by the adversary (to $\mathcal{F}_{\text{STX}}$) between round s and r' is $T_{\mathcal{A}} < C$. Then, by the pigeonhole principle, for all $\vec{\texttt{st}} \in \mathcal{S}_{r'}$, it holds that there is at least one block $\texttt{st}_k$, such that functionality $\mathcal{F}_{\text{STX}}$ is successfully queried by exactly one honest-and-synchronized miner P in round i to extend the state to length $k+1$, but no query by the adversary to extend a state of length k to a state of length $k+1$ has been successful up to and including round r'. Even more, $T_{\mathcal{A}} < C$ implies that such an i has to exist that also constitutes a convergence opportunity.

After this convergence opportunity at round i, all honest-and-synchronized miners have a state whose first $k+1$ blocks are $\vec{\texttt{st}}_i = \texttt{st}_0 \ldots, \texttt{st}_k$. Unless the adversary provides an alternative state with a prefix $\vec{\texttt{st}}'_i$ of length $k+1$, such that $\texttt{st}'_l \neq \texttt{st}_l$ for at least one index $0 < l \leq k$, no honest-and-synchronized miner will ever mine on a state whose first $k+1$ blocks do not agree with $\vec{\texttt{st}}_i$.

The existence of an alternative prefix $\vec{\mathtt{st}}_i'$ of length $k+1$ for any such i and for all states $\vec{\mathtt{st}} \in \mathcal{S}_{r'}^{\ell+C}$ implies $T_{\mathcal{A}} \geq C$ and therefore contradicts the assumption that $T_{\mathcal{A}} < C$.

What is left to prove is that for any such interval of size t (from round s to round r'), the probability that $T_{\mathcal{A}} < C$ holds in any real-world execution except with negligible probability in $\beta_{min}t$. Analogously to Lemma 7.12, by the definition $\rho_a \cdot \mathtt{T}_{mp} = \beta_{min}$ (and recalling that we established a lower bound on t in the beginning) we get that this error probability is negligible in T.

First, by Lemma 7.13, for any $\omega > 0$, the probability that any new state accepted by an honest-and-synchronized miner during the period of at most $t+1$ rounds (from s to r') is actually a state extension that the adversary withheld since round $s - \omega(t+1)$ (or even before) is at most $\mathrm{negl}(\beta_{min}t)$. By Lemma 7.12, the number of adversarial blocks (i.e., successful state extensions by $\mathcal{A}$) generated within this slightly larger interval S' of size $|S'| = (1+\omega)(t+1)$ rounds is (except with probability $\mathrm{negl}(\beta_{min}t)$) upper bounded by $T_{\mathcal{A}} \leq (1+\delta)(1+\omega)(t+1)\overline{\beta}_{S'}$. Also by picking constant ω sufficiently small, we have that $|S| \geq (1 - \alpha_{max}\Delta)|S'|$ and thus $\overline{\alpha}_S$ dominates $\overline{\beta}_{S'}$ by the theorem assumptions. We hence get $T_{\mathcal{A}} \leq \frac{(1+\delta)(1+\omega)}{\lambda}(t+1)\overline{\alpha}_S \cdot (1 - 2\mathtt{T}_{mp} \cdot (\Delta+1))$ by Eq. (1). By picking the constants δ and ω, and ϵ sufficiently small relative to λ, we hence get $T_{\mathcal{A}} < C$ except with probability $\mathrm{negl}(\beta_{min}t)$. Since our arguments hold for any particular intervals S, we again apply the union bound over the number of rounds and get the desired claim. □

We used the following useful lemma in the previous proof to bound the deviation with respect to an arbitrary environment (inducing a certain sequence of random variables):

Lemma 7.17. *Let* $\tau \geq \frac{1}{2}$ *and consider boolean random variables* $X_1, \ldots, X_n$ *for which it holds that* $\Pr[X_i = 1 \mid X_1, \ldots, X_{i-1}] \geq \tau$. *Then, for any* $\delta > 0$,

$$\Pr\left[\sum_{i=1}^{n} X_i \leq (1-\delta)\tau n\right] \leq \exp\left(-\delta^2 n/2\right).$$

Proof. We define the random variables $Y_k := \sum_{i=1}^{k}(X_i - \tau) = (\sum_{i=1}^{k} X_i) - k\tau$. First, they satisfy the sub-martingale condition, i.e., for all k, $\mathbb{E}[Y_k \mid Y_1, \ldots Y_{k-1}] \geq Y_{k-1}$: let $\Pr[Y_k = y_{k-1} + (1-\tau) \mid Y_{k-1} = y_{k-1}] = \Pr[X_k = 1 \mid X_1, \ldots, X_{k-1}] =: p_1 \geq \tau$ and $\Pr[Y_k = y_{k-1} + (-\tau) \mid Y_{k-1} = y_{k-1}] = \Pr[X_k = 0 \mid X_1, \ldots, X_{k-1}] := p_0 \leq 1 - \tau$. The (conditional) expected value is $p_1(y_{k-1} + (1-\tau)) + p_0(y_{k-1} - \tau) \geq y_{k-1} + p_1(1-\tau) - p_0\tau \geq y_{k-1} + [\tau(1-\tau) - (1-\tau)\tau] = y_{k-1}$.

Second, we have a bounded difference of $|Y_k - Y_{k-1}| \leq \max(\tau, 1-\tau) = \tau$ by the condition $\tau \geq 1/2$. Applying the Azuma-Hoeffding bound given by Theorem 2.4 to the variables Y_k gives

$$\Pr[Y_n \leq -\delta\tau n] \leq \exp(-\delta^2 n/2).$$

And by definition $Y_n \leq -\delta\tau n \leftrightarrow X_n \leq n\tau - n\delta\tau$, the statement follows. □

Concluding observations. Finally, we conclude the proof by noting that after a delay of Δ rounds, all honestly multicast transactions are known to all honest-and-synchronized miners and would be included into the next honestly minded block if valid. In the simulation, the simulator also does it in the ideal world and hence will never propose blocks of honest parties that do not comply with the conditions of the defined $\mathsf{ExtendPolicy}$ of $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$. Further, the synchronization of a party takes at most $\texttt{Delay} = 4\Delta$ clock ticks: if P_j joins the network, his knowledge of the longest chain and the set of valid transactions relative to that state, which is known to at least one honest and synchronized miner is only reliable after 2Δ rounds (4Δ clock ticks) since it takes at most Δ rounds to multicast the initial message that the miner has joined the network, and additional Δ rounds until the replies are received. During this 2Δ round the new miner will also have received all messages sent at or after he joined the network, and in particular all transactions that are more than Δ rounds ($2\Delta = \frac{\texttt{Delay}}{2}$) old and potentially valid.

The pointers of honest-and-synchronized parties can also not be too distant, i.e., the slackness is upper bounded by $\texttt{windowSize} \geq T$ as otherwise we would have a common-prefix violation in that execution (assume the prefix of the chain known to a honest-and-synchronized party was further away than T blocks from the prefix of the actual longest chain, this would yield a fork with substantial probability). The theorem follows. □

7.5. *Improving the Chain-Quality Parameter*

As long as $\alpha_{min} > \beta_{max}$, we see that among $\texttt{windowSize}$ state blocks, there is at least an honestly generated block, because then, by Eq. (1), we also have $\gamma_{min} > \beta_{max}$ and thus $\frac{\beta_{max}}{\gamma_{min}} < 1$. Such an assumption is usually taken in existing analyses. However, we can derive more general bounds for chain-quality (where the above case is one special case) to obtain bounds for more general scenarios. In light of the chain-growth statement in Lemma 7.11, we introduce the following useful quantity:

Definition 7.18. Let the mining pattern be $(\vec{\alpha}, \vec{\beta})$ for R rounds, let the network delay be Δ, and let S be an interval. Define

$$cg_{\Delta}(S) := \max\{\tau \in (0,1) \mid \forall S' \subseteq S \text{ with } |S'| \geq \max\{1, |S|(1-\Delta\cdot\gamma(\tau,\Delta))\} : \overline{\alpha}_{S'} \geq \tau\};$$

and define the fraction

$$f_{cq} := \max_{S\subseteq\{0,\ldots,R-1\}} \frac{\overline{\beta}_S}{\gamma(cg_{\Delta}(S),\Delta)}.$$

Both quantities are well-defined as functions since we assume that $\forall r : \alpha_r > 0$. We derive a more general worst-case guarantee for the fraction of adversarial blocks which in particular shows that this fraction is less than one under the theorem condition.

Lemma 7.19. (Generalization of Lemma 7.15) *Consider a real-world execution as in Theorem 7.9. Let P_i be a miner, and let $r \geq 0$. Assume P_i is honest-and-synchronized* *in round r and that the length of the longest state received or stored is $\ell \geq T$. The fraction of adversarially mined blocks within a sequence of T blocks in the state is at most* $\min\{1, (1+\delta)\cdot f_{cq}\}$ *except with probability $R \cdot negl(T)$ for any $\delta > 0$ and where f_{cq} is defined as in Definition 7.18. Under the condition of Theorem 7.9, this means that for the ledger $\mathcal{G}^{B}_{\text{LEDGER}}$, we can guarantee*

$$\eta \geq \min\{(1+\delta)\cdot f_{cq},\ 1\},$$

with $f_{cq} < 1$ (and for any $\delta > 0$).

Proof. The proof proceeds as the one of Lemma 7.15: consider any sub-sequence of T state blocks $\mathtt{st}_{j+1}, \ldots, \mathtt{st}_{j+T}$ in $\vec{\mathtt{st}}_r$. We again assume that $\vec{\mathtt{st}}^{<j}$ is mined at round r' (by an honest-and-synchronized party), and that in round $r'+t$, the state $\vec{\mathtt{st}}^{>j+T}$ appears for the first time as the state, or the prefix of a state, of at least one honest-and-synchronized miner. Recall that if an adversary successfully extended the state during some round by a new state block $\mathtt{st}_{j+s}$ of the above sequence $\mathtt{st}_{j+1}, \ldots, \mathtt{st}_{j+T}$, then this happens in a round between r' and $r'+t$. Let us denote this interval by the set S of rounds.

Since t is assumed to be the minimal number of rounds until the first honest miner adopted a state containing $\mathtt{st}_{j+1}$, we can actually make use of the general part of Lemma 7.11 to conclude that the probability that the condition $t \geq \frac{T}{(1-\delta')\gamma(cg_\Delta(S),\Delta)}$ occurs in such an execution is at most $\text{negl}(T)$ and obtain $t \leq \frac{T}{(1-\delta')\gamma(cg_\Delta(S),\Delta)}$ with overwhelming probability in T. On the other hand, the lower bound on t is as in the proof of Lemma 7.15.

Let again N_A^t denote the expected value of adversarial blocks in t rounds, invoking Lemma 7.12 gives us that $N_A^t \leq (1+\delta)\overline{\beta}_S t$ except with probability $\text{negl}(\overline{\beta}_S t)$.

The number of times the adversary was successful in extending the state by one block can therefore be upper bounded by the quantity

$$N_A^{\frac{T}{(1-\delta')\gamma}} \leq \frac{1+\delta}{1-\delta'}\cdot T \cdot \frac{\overline{\beta}_S}{\gamma(cg_\Delta(S),\Delta)}.$$

Since our arguments hold for any interval, the proof is concluded by taking the worst case over all rounds and the maximal fraction equals f_{cq} as claimed.

To establish the last part of the statement, we observe that Eq. (1) in particular implies that for any interval S (of sufficient size), we have that any subset S' of rounds of size $(1-\alpha_{max}\Delta)|S|$ fulfills $\overline{\alpha}_{S'}(1-\mathtt{T}_{mp}\Delta) > (1+\epsilon)\overline{\beta}_S$ for some $\epsilon > 0$. Since a lower bound x for $\overline{\alpha}_{S'}$ over all subsets of size $(1-\alpha_{max}\Delta)|S|$ implies that x is also a lower bound for any larger subset S'' and hence for $cg_\Delta(S)$. Observing that for $x, \Delta > 0$, $\frac{x}{1+x\Delta} > x(1-x\Delta)$ and $\mathtt{T}_{mp} \geq cg_\Delta(S)$, we get $\gamma(cg_\Delta(S),\Delta) > \overline{\beta}_S$ as required to conclude that $f_{cq} < 1$. □

8. Special Cases of our Model and Functionality Wrappers

In this section, we first explain how our main theorem relates to the influential initial provable security analyses of Bitcoin. Afterward, we show how to use functionality wrappers to enforce the main theorem's conditions in order to obtain composable statements (i.e., with respect to all environments).

8.1. *Special Cases and Existing Works*

We demonstrate how the protocols, assumptions, and results from the two existing works analyzing security of Bitcoin (in a property based manner) can be cast as special cases of our construction. We focus on the early analyses of Pass et al. [44] (PSs for short) and the original analysis of Garay et al. [24] (GKL for short). Both initial models assume a fixed upper bound n on the number of active participants in the protocol execution. All honest parties are assumed to be synchronized (e.g., by special initialization messages by the environment).

GKL analysis (fixed difficulty and delay). We start with the result in [24], in particular with the so-called flat (every party has the same hashing power) and synchronous model with next-round delivery. The relevant variables are defined as follows:

- Each party is allowed to perform $q \geq 1$ hash queries. This translates to a success probability of $p_H = 1 - (1 - p)^q$ and $p_A = p$, and to a total mining power $\mathrm{T}_{mp}^{\mathrm{GKL}} := p \cdot q \cdot n$.
- The adversary gets (at most) q queries per corrupted party with probability $p_A = p$ (there are no desynchronized parties). If t_r denotes the number of corrupted parties in round r, the expected value would be $t_r \cdot q \cdot p$, and thus, we can define the upper bound on the adversarial mining power $\beta_{\max}^{\mathrm{GKL}} = p \cdot q \cdot (\rho \cdot n)$, where ρn is the (assumed) upper bound on the number of miners contributing to the adversarial mining power (independent of r). Since the adversary is free to go to the limit in the model, the mining pattern is also flat: $\vec{\beta} = (\beta_{\max}^{\mathrm{GKL}}, \ldots, \beta_{\max}^{\mathrm{GKL}})$.
- Each honest and synchronized miner gets exactly one activation per round and has success probability $p_H = 1 - (1 - p)^q \in (0, 1)$, for some integer $q > 0$ and hence we get a minimal honest mining power of $\alpha_{\min}^{\mathrm{GKL}} = 1 - (1 - p)^{q(1-\rho)\cdot n}$ (independent of r). Note that since n is assumed to be fixed in their model, $q(1 - \rho) \cdot n$ is in fact a lower bound on the honest and synchronized hashing power. Since the model assumes that this lower bound could potentially always be allowed, we again define the flat mining pattern $\vec{\alpha} = (\alpha_{\min}^{\mathrm{GKL}}, \ldots, \alpha_{\min}^{\mathrm{GKL}})$.
- If instant delivery is assumed, this translates to defining $\Delta^{\mathrm{GKL}} := 1$, i.e., guaranteed delivery in the next round.

PSs analysis (fixed difficulty). Similarly, we can instantiate the above values with the assumptions of [44]:

- For each corrupted party, the adversary gets at most one query per round. Each honest miner makes exactly one query per round. In total, there are n parties among which ρn can be corrupted (in any round).

- In the PSs model, $p_H = p_A = p$ and hence $\mathrm{T}_{mp}^{\mathrm{PSs}} = p \cdot n$. With these values we get $\beta_{\max}^{\mathrm{PSs}} = p \cdot (\rho \cdot n)$. Putting things together, we also have $\alpha_{\min}^{\mathrm{PSs}} = 1 - (1-p)^{(1-\rho)\cdot n}$, where $(1-\rho)\cdot n$ is the lower bound on the honest (and hence also synchronized) parties. As before, the mining pattern is flat.
- The delay of the network is upper bounded by a constant Δ^{PSs} (as usual, unknown to the participants).

The security is established by the following lemma:

Lemma 8.1. *For the special settings above, if we impose the assumption that*

$$\alpha_{\min}^{\{\mathrm{GKL,\,PSs}\}} \cdot (1 - 2 \cdot (\Delta^{\{\mathrm{GKL,\,PSs}\}} + 1) \cdot \alpha_{\min}^{\{\mathrm{GKL,\,PSs}\}}) \geq \lambda \cdot \beta_{\max}^{\{\mathrm{GKL,\,PSs}\}} \tag{4}$$

then this implies the secure realization of the Bitcoin ledger with the parameters assured by Theorem 7.9 for the above choices of values, respectively.

Proof Sketch. The statement of course follows from the arguments given in the respective works [24] and [44] since our execution model in particular allows us to formulate the above assumptions. However, it is instructive to see how the security follows in view of Theorem 7.9. In particular, why security follows when replacing the condition in Eq. (1) by Eq. (4). At first sight, the condition is stronger as it implies that the best strategy of the adversary is dominated by the worst strategy of the honest players. However, the discount factor $(1 - 2 \cdot (\Delta^{\{\mathrm{GKL,\,PSs}\}} + 1) \cdot \alpha_{\min}^{\{\mathrm{GKL,\,PSs}\}})$ is better than $(1 - 2 \cdot (\Delta^{\{\mathrm{GKL,\,PSs}\}} + 1) \cdot \mathrm{T}_{mp}^{\mathrm{GKL,PSs}})$. The key observation why Eq. (4) subsumes Eq. (1) in the special cases described above are the following:

- Since the number n of parties is fixed and exactly divided into honest and adversarial, and because the worst-case honest strategy still dominates the adversary's best strategy, we can use to following argument to justify why Eq. (4) is actually sufficient. Still, the best strategy of the adversary is to activate as many corrupted parties, say t, as allowed by the upper bound β_{max}. Since the number of parties is fixed, this implies that at most $n - t$ activations of honest parties remain and by definition $\alpha_{min} = 1 - (1-p)^(n-t)$ is the matching lower bound. Hence, and in contrast to the more general setting, here the best strategy for corrupted parties induces a concrete strategy for honest parties.[13] A bit more formally, let x denote the number of queries such that $\alpha_{min} = 1 - (1-p)^x$ holds. Assume in some round r, more honest parties are activated, say q_H^r. By definition, $\beta_{max} \geq p \cdot (n - x)$ and we can formally assign the difference $(q_H^r - x)$ to the adversary's budget (and the condition $\alpha_{min} > \beta_{max}$ is preserved as stated below). First, observe that for integers $x, y > 1$,

[13]Note that in a more general setting, this not need to be the case: even if the bound on the adversary is small, by activating a huge fraction of honest parties the consensus of honest parties could still be disturbed and hence our analysis has to consider such "malicious" strategies as well.

$$\begin{aligned}\alpha_r - \alpha &= (1-(1-p)^{x+y}) - (1-(1-p)^x) = (1-p)^x - (1-p)^{x+y}\\ &= (1-p)^x \cdot (1-(1-p)^y) \le (1-(1-p)^y) \le (1-(1-y\cdot p))\\ &= y \cdot p,\end{aligned}$$

where the last inequality is a consequence of Bernoulli's inequality. The adversary's mining power is thus increased, however not beyond β_{max} since the identity $n - x = (n - q_H^r) + (q_H^r - x)$ is guaranteed because n and x are fixed for the analysis.

- Looking at the proof of Theorem 7.9, we see that the quantities $\overline{\alpha}_S$ and $\overline{\beta}_S$ can be identified by α_{min} and β_{max}, respectively, and in addition the relationship $\alpha_{min} > \beta_{max}$ is implied by Eq. (4) (and thus $\overline{\alpha}_S > \overline{\beta}_S$ for any subset S of rounds of any size. With this, all Lemmata in the proof of Theorem 7.9 simplify and no further condition in addition to Eq. (4) is needed.

With this in mind, replacing the condition in Eq. (1) by Eq. (4) the proof of Theorem 7.9, under the conditions imposed by the above models, yields the statement of the lemma. □

8.2. *Restrictions and Composition*

Note that the theorem statement a-priori holds for any environment (but simply yields a void statement if the conditions are violated). In order to turn this into a composable statement without restrictions, we follow the approach proposed in Sect. 3 and model restrictions in the setup of the protocol via wrapper functionalities. The general conceptual principle behind this is the following: For the hybrid world, that consists of a network $\mathcal{F}_{\text{N-MC}}$, a clock $\mathcal{G}_{\text{CLOCK}}$ and a random oracle $\mathcal{F}_{\text{RO}}^{\kappa}$ with output length κ (or alternatively the state-exchange functionality $\mathcal{F}_{\text{STX}}$ instead of the random oracle), define a wrapper functionality $\mathcal{W}$ which enforces a given mining pattern $(\vec{\alpha}, \vec{\beta})$ (and the upper bounds on the mining power). If the conditions of Theorem 7.9 are met, then we get a UC-realization statement with respect to all (efficient) environments.

A general wrapper. We define such a general wrapper for our setting and denote it by $\mathcal{W}_{\vec{\alpha},\vec{\beta},\mathrm{D}}^{\Delta,\mathrm{T}_{mp}}(\mathcal{F}_{\text{N-MC}}^{\Delta}, \mathcal{F}_{\text{RO}}^{\kappa})$ in Fig. 10. Note that this wrapper slightly changes the synchrony pattern of the real-world execution: since a lower bound on honest mining power is enforced (otherwise, the clock does not go on), we realize the ledger with a slightly different predicate $\mathsf{predict\text{-}time}_{BC}$ to reflect this assumption. It is easy to see that this is a straightforward extension to the derivation in Lemma 5.2. We note that this change to the synchronization pattern just stems from the fact how we implement such restricting assumptions but does not affect other modeling decisions and the we still realize the ledger (this is actually a major motivation to abstract the time-dependency of the ledger using such an abstract predicate, such that minor details have only local effects). For this wrapper we have the following desired corollary to Theorem 7.9 and Lemma,7.2. This statement is guaranteed to compose according to the UC composition theorem.

Corollary 8.2. *The protocol* $\mathsf{Ledger\text{-}Protocol}_{q,D,T}$, *defined in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{W}_{\vec{\alpha},\vec{\beta},D}^{\Delta,T_{mp}}(\mathcal{F}_{\text{N-MC}}^{\Delta}, \mathcal{F}_{\text{RO}}^{\kappa}))$*-hybrid world, UC-realizes functionality* $\mathcal{G}_{\text{LEDGER}}^{B}$ *(for the parameters established by Theorem 7.9 and the extended predicate* $\mathsf{predict\text{-}time}_{BC}$

Wrapper from Corollary 8.2

The wrapper is assumed to be registered with the global clock $\mathcal{G}_{\text{CLOCK}}$ and wraps the network and the random oracle. The functionality manages the variable `counter` and is aware of set of registered parties, and the set of corrupted parties.

Initially, $\mathcal{P}' = \emptyset$ and $\texttt{counter} = 0$, $q_A = 0$ and $q_H = 0$. Define $p := \frac{\mathtt{D}}{2^\kappa}$ (where κ is the output length of the underlying random oracle).

General:

- The wrapper stops the interaction with the adversary as soon as the adversary tries to exceed its allowed budget of hashing power.

Relaying inputs to the random oracle:

- Upon receiving $(\text{EVAL}, \text{sid}, x)$ from $\mathcal{A}$ on behalf of a party P which is corrupted or registered but de-synchronized, then first execute *Round Reset*. Then do the following:
  $q_A \leftarrow q_A + 1$; $\beta^{(\texttt{counter})} \leftarrow q_A \cdot p$
  if $(q_A + q_H) \cdot p \leq \mathtt{T}_{mp}$ **then**
    if $\beta^{(\texttt{counter})} \leq \vec{\beta}[\texttt{counter}]$ **then**
      Forward the request to $\mathcal{F}^{\kappa}_{\text{RO}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}^{\kappa}_{\text{RO}}$ returns.

- Upon receiving $(\text{EVAL}, \text{sid}, x)$ from an uncorrupted, registered and synchronized party P, then first execute *Round Reset*. Then do the following:
  $q_H \leftarrow q_H + 1$; $\alpha^{(\texttt{counter})} \leftarrow 1 - (1-p)^{q_H}$
  if $(q_A + q_H) \cdot p \leq \mathtt{T}_{mp}$ **then**
    Forward the request to $\mathcal{F}^{\kappa}_{\text{RO}}$ and return to P whatever $\mathcal{F}^{\kappa}_{\text{RO}}$ returns.
  if $\alpha_{\texttt{counter}} \geq \vec{\alpha}[\texttt{counter}]$ **then**
    Send $(\text{CLOCK-UPDATE}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ ▷ Release the clock if lower bound is reached.

Relaying inputs to the network:

- Upon receiving $(\text{MULTICAST}, \text{sid}, (m_{i_1}, P_{i_1}), \ldots, (m_{i_\ell}, P_{i_\ell}))$ from $\mathcal{A}$ for the network then first execute *Round Reset*. Then do the following:
  for each m_{i_j} **do**
    if m_{i_j} can be parsed as sequence of blocks $(\mathbf{B}_1, \ldots, \mathbf{B}_n)$ **then**
      Parse m_{i_j} as $(\mathbf{B}_1, \ldots, \mathbf{B}_n)$
      for each $\mathbf{B}_i$ **do**
        if $\mathbf{B}_i$ has not yet been queried to $\mathcal{F}^{\kappa}_{\text{RO}}$ **then**
          Execute the above instructions on input $(\text{EVAL}, \text{sid}, \mathbf{B}_i)$ from a corrupted party.

- Any other request is relayed to the underlying functionality (and recorded by the wrapper) and the corresponding output is given to the destination specified by the underlying functionality.

Standard UC Corruption Handling:

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

Procedure Round-Reset:
Send $(\text{CLOCK-READ}, \text{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and receive $(\text{CLOCK-READ}, \text{sid}_C, \tau)$ from $\mathcal{G}_{\text{CLOCK}}$. If $|\tau - \texttt{counter}| > 0$ and the new time τ is even (i.e., a new round started), then set $\texttt{counter} \leftarrow \tau$ and set $q_A \leftarrow 0$ and $q_H \leftarrow 0$.

Fig. 10. The wrapper that restricts the adversarial access to the real-world resources.

as described above) if the parameters of the wrapper (and thus formally enforced by the setup-functionality of the protocol), satisfy equation (1).

Remark. It is straightforward to design different wrappers capturing a range of assumptions that one might want to make (and which imply the conditions of Theorem 7.9), such as an explicit restriction on number of active participants etc. An additional interesting observation is that if Bitcoin did require that block extensions are accompanied by their hash explicitly, i.e., report $y := H[\mathbf{B}_i]$ along with a new block $\mathbf{B}_i$ (and honest parties consequently ignored a received block that is not paired with the right hash value), then the wrapper could be simplified. As the adversary would not be able to abuse honest verifications as actual (adversarial) work, the wrapper would not have to consider the special case of adversarial multicast-messages in Fig. 10 when computing the budget.

As a final remark, we point out that the wrapper enforces that honest and adversarial attempts at solving PoW-puzzles (modeled via evaluations of the RO) are made in tandem, that is, the adversary cannot consume the budget of future rounds ahead of time. This also means that, effectively, the execution starts at the same time for everyone. This is one of two ways how to render pre-computation attacks ineffective, the other one being to define the start of the execution to be the point when a random genesis block is published—by an additional hybrid functionality as done for example in [6] in the PoS case.

9. Modular Constructions Based on the Ledger

The ledger functionality can be enhanced in a modular way in various directions. In this section, we show a simple extension to strengthen liveness thanks signatures. Informally, the stronger guarantee ensures that every transaction submitted by an honest participant will eventually make it into the state. In this section we present this stronger ledger and show how such an implementation can be captured as a UC protocol which makes black-box use of the Ledger-Protocol to implement this ledger. The UC composition theorem makes such a proof immediate, as we do not need to think about the specifics of the invoked ledger protocol, and we can instead argue security in a hybrid world with access to $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$.

Protection of transactions using addresses. In Bitcoin, a participant creates a unique address denoted by AddrID by generating a signature key pair and hashing the public key. Any transaction of this party includes this address, i.e., $\mathtt{tx} = (\mathsf{AddrID}, \mathtt{tx}')$. An important property is that a transaction of a certain address cannot be invalidated by a transaction with a different address ID. Hence, to protect the validity of a transaction, upon submitting $\mathtt{tx}$, party P_i has to sign it, append the signature and verification key to get a transaction $((\mathsf{AddrID}, \mathtt{tx}'), vk, \sigma)$. The validation predicate now additionally has to check that the address is the hash of the public key and that the signature σ is valid with respect to the verification key vk. Roughly, an adversary can invalidate $\mathtt{tx}$, only by either forging a signature relative to vk, or by possessing key pair whose hash of the public key collides with the address of the honest party.

The realized ledger abstraction, denoted by $\mathcal{G}^{\text{B}+}_{\text{LEDGER}}$, is a ledger functionality as the one from the previous section, but which additionally allows parties to create unique addresses. Upon receiving a transaction from party P_i, $\mathcal{G}^{\text{B}+}_{\text{LEDGER}}$ only accepts a transaction containing the AddrID that was previously associated to P_i and ensures that parties are restricted to issue transactions using their own addresses. As we explain, this amplifies transaction liveness.

9.1. *A Stronger Ledger with Address Management*

9.1.1. *Overview and Definitions*

To achieve stronger guarantees than our original Bitcoin ledger, a party issues transactions relative to an address. More abstractly speaking, a transaction contains an identifier, AddrID, which can be seen as the abstract identity that claims ownership of the transaction. More specifically, we can represent this situation by having transactions $\mathtt{tx}$ be pairs $(\mathsf{AddrID}, \mathtt{tx}')$ with the above meaning. Signatures enter the picture at this level: an honest participant will issue only signed transactions. In order to link verification key to the address, AddrID is the hash of the verification keys, where we require collision resistance. More concretely, whenever a miner is supposed to submit a transaction $\mathtt{tx}$, it signs it and appends the signature and its verification key. The validation consists of three parts. First, it is verified that the public key matches the address, second, the signature is verified, and third, its validated whether the actual transaction $(\mathsf{AddrID}, \mathtt{tx}')$ is valid, with respect to a separate validation predicate $\mathsf{ValidTx}_{\text{₿}}$ on states and transactions $\mathtt{tx}$ of the above format. Only if all three tests succeed, the transactions is valid. We make use of an existentially unforgeable digital signature scheme and recall its definition here:

Definition 9.1. A digital signature scheme DSS $:= (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ for a message space $\mathcal{M}$, signature space $\mathcal{S}$, and key space $\mathcal{K} = \mathcal{SK} \times \mathcal{PK}$ consists of a (probabilistic) key generation algorithm Gen that returns a key pair $(sk, vk) \in \mathcal{K}$, a (possibly probabilistic) signing algorithm Sign, that given a message $m \in \mathcal{M}$ and the signing key $sk \in \mathcal{SK}$ returns a signature $s \leftarrow \mathsf{Sign}(sk, m)$, and a (possibly probabilistic, but usually deterministic) verification algorithm Ver, that given a message $m \in \mathcal{M}$, a candidate signature $s' \in \mathcal{S}$, and the verification key $vk \in \mathcal{PK}$ returns a bit $\mathsf{Ver}(vk, m, s')$. The bit 1 is interpreted as a successful verification and 0 as a failed verification. We require correctness, that is, we demand that $\mathsf{Ver}(vk, m, \mathsf{Sign}(sk, m)) = 1$ for all $m \in \mathcal{M}$ and all pairs (vk, sk) in the support of Gen.

Definition 9.2. A digital signatures scheme is existentially unforgeable under chosen message attacks if no efficient adversary A can win the following security experiment better than with negligible probability: the challenger first chooses a key pair $(sk, vk) \leftarrow \mathsf{Gen}$. Then it acts as a signing oracle, receiving messages $m \in \mathcal{M}$ from the adversary and responding with $\mathsf{Sign}(sk, m)$. At any point, A can undertake a forging attempt by providing a message m' and a candidate signature s' to the challenger. The adversary wins if and only if $\mathsf{Ver}(vk, m', s') = 1$ and m' was never queried before by A.

9.1.2. *The Protocol for Address Management*

Hybrid ledger functionality. Let $\mathsf{ValidTx}_{\text{₿}}$ and $\mathsf{blockify}_{\text{₿}}$ be as in the previous section but with the following additional property: each transaction is a pair $\mathtt{tx} = (\mathsf{AddrID}, \mathtt{tx}')$ where the first part is bitstring of fixed length and the second part is an arbitrary transaction payload. In addition we require the following property: for any state $\mathtt{state}$ and any transaction $\mathtt{tx}$ it holds that $\mathsf{ValidTx}_{\text{₿}}(\mathtt{tx}, \mathtt{state}) = 1$ implies, for any state extension $\mathtt{state}||\mathtt{st}'$, that $\mathsf{ValidTx}_{\text{₿}}(\mathtt{tx}, \mathtt{state}||\mathtt{st}') = 1$, if $\mathtt{st}'$ does not contain a transaction with the same identifier AddrID (this is clearly satisfied for Bitcion for example). Recall that we also assume that Definition 6.1 is satisfied.

Our protocol is defined w.r.t. a Bitcoin ledger functionality with the following validation predicate, which is defined relative to a hash function H, and a signature scheme DSS.

Algorithm to describe the assumed validation predicate

function $\mathsf{Val}_{\mathrm{DSS}}(\mathtt{BTX}, \mathtt{state}, \mathtt{buffer})$
  Let $\mathtt{BTX} = (\mathtt{tx}, \text{txid}, \tau_L, p_i)$
  Parse $\mathtt{tx}$ as $((\mathsf{AddrID}, \mathtt{tx}'), vk, \sigma)$ (Return 0 in case of a wrong format)
  if $\mathsf{AddrID} = H(vk)$ **and** $\mathsf{Ver}(vk, \mathtt{tx}, \sigma) = 1$ **then**
    return $\mathsf{ValidTx}_{\text{\Bitcoin}}(\mathtt{tx}, \mathtt{state})$
  else
    return 0

Protocol. The protocol is straightforward: whenever the protocol is given an input of the form $(\mathsf{AddrID}, \mathtt{tx})$ it first checks that it is the party associated with this address ID. Then, it receives the newest state from the ledger and checks, whether this input is valid with respect to the current state. If this is the case, the party signs the input and submits it to the ledger.

Protocol $\mathsf{addrMgmt}(P)$

General Behavior:

This protocol exports the same interface as $\mathcal{G}_{\text{LEDGER}}$, and only changes the behavior of read or submit-queries to the ledger. Any other command is simply relayed to $\mathcal{G}_{\text{LEDGER}}$ and the corresponding output is given to the environment.

Address Management:

- The protocol maintains a counter i (initially 0).
- Upon receiving (CREATEADDRESS, sid), execute $(sk, vk) \leftarrow \mathsf{Gen}$, update $i \leftarrow i + 1$ and set $\mathsf{AddrID}_i \leftarrow H(vk)$. Return (CREATEADDRESS, sid, AddrID_i)

Ledger Read and Write:

- Upon receiving (READ, sid) send (READ, sid) to $\mathcal{G}_{\text{LEDGER}}$ and receive as answer the current $\mathtt{state} = \mathtt{st}_1 || \ldots || \mathtt{st}_n$. Then do the following:

  $\mathtt{state}' \leftarrow \mathtt{st}_1$ ▷ Genesis state
  for $i = 2$ to n **do**
    From state block $\mathtt{st}_i$, extract the contents $(\mathtt{tx}_1, vk_1, \sigma_1) || \ldots || (\mathtt{tx}_n, vk_n, \sigma_n)$
    Define new block-content $\vec{x}' \leftarrow \mathtt{tx}_1 || \ldots || \mathtt{tx}_n$
    $\mathtt{state}' \leftarrow \mathtt{state} || \mathsf{blockify}_{\text{\Bitcoin}}(\vec{x}')$

  Return (READ, sid, $\mathtt{state}'$)

- Upon receiving (SUBMIT, sid, $\mathtt{tx}$), check that $\mathtt{tx} = (\mathsf{AddrID}, \mathtt{tx}')$ for $\mathsf{AddrID} \in \{\mathsf{AddrID}_1, \ldots \mathsf{AddrID}_i\}$. If the check fails, ignore the input. Otherwise, do the following:

  1. Read the state $\mathtt{state}$ from $\mathcal{G}_{\text{LEDGER}}$ as above.
  2. If $\mathsf{ValidTx}_{\text{\Bitcoin}}(\mathtt{tx}, \mathtt{state}) = 1$, then sign the input by $\sigma \leftarrow \mathsf{Sign}(sk, \mathtt{tx})$ and send (SUBMIT, sid, $(\mathtt{tx}, vk, \sigma)$)

9.1.3. *The Enhanced Ledger Functionality*

We present an enhanced ledger functionality with a validation predicate that enforces that an adversarial transaction cannot prevent a transaction by an honest party to eventually make it into the stable state of the ledger.

Functionality $\mathcal{G}_{\text{LEDGER}}^{₿+}$

$\mathcal{G}_{\text{LEDGER}}^{₿+}$ is identical to $\mathcal{G}_{\text{LEDGER}}^{₿}$ except with the following changes:

Difference to standard Ledger:

- Upon receiving (CREATEADDRESS, sid) from party P_i (or the adversary on behalf of a party P_i), send (ACCOUNTREQ, sid, P_i) to $\mathcal{A}$ and upon receiving a reply (ACCOUNTREQ, sid, P_i, AddrID) do the following:
  1. If AddrID is not yet associated to any party, store the pair (AddrID, P_i) internally and return (CREATEADDRESS, sid, AddrID) to P_i.
  2. If AddrID is already associated to a party, then output (CREATEADDRESS, sid, Fail) to P_i.

Standard Bitcoin Ledger:

- Identical to $\mathcal{G}_{\text{LEDGER}}^{₿}$ with validation predicate $\mathsf{Val}_{\text{strong}}$ and with the fixed transaction format described above. We omit the formal specification here.

The following validation predicate is used within $\mathcal{G}_{\text{LEDGER}}^{₿+}$.

Algorithm to define the strong validation

```
function Val_strong(BTX, state, buffer)
    Let BTX = (tx, txid, τ_L, p_i)
    if tx = (AddrID, tx') and AddrID is associated to P_i then
        return ValidTx_₿(tx, state)
    else
        return 0
```

On the better guarantees. The stronger guarantee for honestly submitted transactions stems from two facts. First, by Definition 6.1, the state blocks contain transactions beyond coin-base transactions. Second, since a transaction of a party is associated with its address, and cannot be invalidated by another transaction with a different address, this implies that the transaction remains valid relative to `state` (unless the honest party itself issues a transaction that contradicts a previous transaction for one of its addresses, but we neglect this here). As an example, assume an honest party submits a single transaction for one of its addresses, and assume this transaction is valid relative to the state `state`. Then, by the defined enforcing mechanism of ExtendPolicy, this transaction is guaranteed to enter the state after staying in the buffer for long enough, because the ledger continuously enforces that a certain fraction of blocks contain all those unconfirmed (and still valid) transactions that are older than a certain threshold.

We have the following lemma:

Lemma 9.3. *Let* DSS *be a secure digital signature scheme and let H be a collision resistant hash function. Then the protocol* addrMgmt *in the $\mathcal{G}_{\text{LEDGER}}^{₿}$-hybrid world UC-realizes ledger $\mathcal{G}_{\text{LEDGER}}^{₿+}$, where the functionalities are instantiated as described above.*

Proof Sketch. It is straightforward to write a simulator in the ideal-world execution that perfectly mimics the protocol as long as no hash-collision or signature forgery occurs. This is because the only non-trivial property that the ledger enforces (in addition to what the assumed ledger guarantees) is that only the address holder can submit a transaction but no one else. If no hash-function collision is found, the only possible way is to forge a signature. If both events do not happen, the real world indeed implements the stronger validation predicate. Assuming a collision-resistant hash function and a signature scheme that is unforgeable under chosen-message attacks, this implies the statement. $\square$

Funding Open access funding provided by Swiss Federal Institute of Technology Zurich.

A. Further Details on the Model

This section includes complementary material for Sect. 3.

A.1. *Unicast Channels*

A unicast channel can be defined as follows:

Functionality $\mathcal{F}_{\mathbf{U\text{-}CH}}^{\Delta, P_R}$

The functionality is parametrized with a receiver P_R, and and upper bound Δ on the delay of any channel. It keeps track of the set of possible senders $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$. The list of messages is stored in $\vec{M}$, initially empty.

- Upon receiving (SEND, m) from some $P_s \in \mathcal{P}$ or from the adversary $\mathcal{A}$, choose a new unique message-ID mid for m, initialize variables $D_{\mathsf{mid}} := 1$ and $D_{\mathsf{mid}}^{MAX} = 1$, set $\vec{M} := \vec{M}||(m, \mathsf{mid}, D_{\mathsf{mid}})$, and send $(m, \mathsf{mid}, D_{\mathsf{mid}})$ to the adversary.
- Upon receiving (FETCH) from P_R:
  1. For all registered mids, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.
  2. Let $\vec{M}_0$ denote the subvector $\vec{M}$ including all triples $(m, \mathsf{mid}, D_{\mathsf{mid}})$ with $D_{\mathsf{mid}} = 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}_0$ from $\vec{M}$ and send $\vec{M}_0$ to P_R.
- Upon receiving $(\text{DELAY}, T_{\mathsf{mid}}, \mathsf{mid})$ from the adversary, if $D_{\mathsf{mid}}^{MAX} + T_{\mathsf{mid}} \le \Delta$ and mid is a message-ID registered in the current $\vec{M}$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} + T_{\mathsf{mid}}$ and $D_{\mathsf{mid}}^{MAX} := D_{\mathsf{mid}}^{MAX} + T_{\mathsf{mid}}$; otherwise, ignore the message.
- Upon receiving $(\text{SWAP}, \mathsf{mid}, \mathsf{mid}')$ from the adversary, if mid and mid' are message-IDs registered in the current $\vec{M}$, then swap the triples $(m, \mathsf{mid}, D_{\mathsf{mid}})$ and $(m, \mathsf{mid}', D_{\mathsf{mid}'})$ in $\vec{M}$. Return (SWAP-OK) to the adversary.

A.2. *On realizing Multicast from Unicast*

We briefly sketch how to realize such a multicast network, in particular its synchronized version along the lines of [34], by means of a synchronized message-diffusion protocol over a network of unicast channels (and implicitly assuming a local clock to obtain the round structure). The core of this diffusion protocol are the assumed and known (e.g., by a common list of IP addresses) relay-nodes to which parties thus can connect and which forward in each round all new messages they received (either from registered parties or other relay nodes) in the previous round to all the unicast channels they are connected to as senders.[14] Let $G = (V, E)$ denote the (dynamically updatable) directed graph whose vertices V are the parties and the relay-nodes which are currently participating in the execution and an edge (p_i, p_j) is in E iff p_i is one of the senders of the multicast channel with receiver p_j. It is straightforward to verify that provided that G restricted to the honest parties (i.e., when corrupted parties and the edges that use them are deleted from G) remains *strongly connected* (i.e., there is a directed path between any two honest parties, in either direction), then the diffusion mechanism executed over unicast channels with delay at most Δ security realizes a multicast network with delay Δd where d is an upper bound of the diameter of G. Indeed, the simulator, which is given any message submitted to any unicast channel and enough activations when the dummy parties themselves get activated (note that it is essentially a synchronous computation among the relay-nodes) needs to simply simulate when the respective parties would see a message and schedule the corresponding deliveries by using the delays submitted by the adversary. The fact that each channel has at most Δ delay means that it will take delay at most ΔL rounds for it to travel through an honest path of length L. Last but not least, in order to receive messages from the network established this way, when a party joins the network, it has to multicast a special message to the relay-nodes that has to contain its identifier such that the relay-nodes can start sending messages to that party. This induces at most a delay of Δ rounds until the party is guaranteed to receive the messages sent over the network. For simplicity, we ignore this additional delay incurred by the registration to the network, and omit it in our specification of the multicast functionality in Sect. 3.2. If one implements the network using the above sketched method, one would formally obtain the a multicast functionality as given in Fig. 1, but where the party set $\mathcal{P}$ contains all parties that have joined (and not yet left) the network at least Δ rounds ago, since the sketched solution does not support instant registration. All remaining guarantees remain unchanged with respect to this new party set.

B. Further Details on the Bitcoin Ledger

This section includes complementary material for Section Sect. 6. We here give the formal description of the Extend Policy for $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$. It is easy to observe that the computation performed by this algorithm is well-defined for any definition of $\mathsf{Validate}$ and $\mathsf{Blockify}$.

Compared to previous versions of this work, the presentation is now logically divided into the step of deriving a default extension and the actual tests whether the adversarial proposal is admissible. The default extension is taken as the ledger-state extension if and only the proposal by the adversary does not pass the test specified and implemented by $\mathsf{ExtendPolicy}$. The derivation of the default extension is given as pseudo-code below. Note also that the policy makes the initial bootstrapping time of the chain now explicit, where by bootstrapping time we mean the time it takes for the first state block to be inserted into the ledger state.

[14] In order to ensure that parties can send some messages twice, a nonce is attached to each input message that is to be multicasted. The relayers do not add another nonce to the message they relay.

Algorithm for Default State Extension

function DefaultExtension($\vec{\mathcal{I}}_H^T$, state, NxtBC, buffer, s_{ep})

We assume call-by-value and hence the function has no side effects.
The function returns a policy-compliant extension of the ledger state.

Let τ_L be current ledger time (computed from $\vec{\mathcal{I}}_H^T$)
Read $\vec{\tau}_{\texttt{state}}$ and $\vec{\text{hf}}$ from the passed state s_{ep}
$\vec{N}_{\texttt{df}} \leftarrow \varepsilon$
Set $\vec{N}_0 \leftarrow \texttt{tx}^{\text{coin-base}}_{\text{minerID}}$ of an honest miner
Sort buffer according to time stamps and let $\vec{\texttt{tx}} = (\texttt{tx}_1, \dots, \texttt{tx}_n)$ be the transactions in buffer
Set st $\leftarrow \mathsf{blockify}_{₿}(\vec{N}_0)$
repeat
    Let $\vec{\texttt{tx}} = (\texttt{tx}_1, \dots, \texttt{tx}_n)$ be the current list of (remaining) transactions
    for $i = 1$ to n **do**
        if $\mathsf{ValidTx}_{₿}(\texttt{tx}_i, \texttt{state}||\texttt{st}) = 1$ **then**
            $\vec{N}_0 \leftarrow \vec{N}_0 || \texttt{tx}_i$
            Remove $\texttt{tx}_i$ from $\vec{\texttt{tx}}$
            Set st $\leftarrow \mathsf{blockify}_{₿}(\vec{N}_0)$
until $\vec{N}_0$ does not increase anymore
$c \leftarrow 0$
if $|\texttt{state}| < \texttt{windowSize} - 1$ **then** ▷ First extend to windowSize − 1 state blocks.
    while $|\texttt{state}| + c < \texttt{windowSize} - 1$ **do**
        if $c > 0$ **then**
            Set $\vec{N}_c \leftarrow \texttt{tx}^{\text{coin-base}}_{\text{minerID}}$ of an honest miner
        $\vec{N}_{\texttt{df}} \leftarrow \vec{N}_{\texttt{df}} || \vec{N}_c$
        $\vec{\tau}_{\texttt{state}} \leftarrow \vec{\tau}_{\texttt{state}} || \tau_L$
        $c \leftarrow c + 1$
$\vec{\tau}_{\texttt{state}} \leftarrow \vec{\tau}_{\texttt{state}} || \tau_L$ ▷ Check whether more extensions possible.
$hr \leftarrow \max_{s=1,\dots,|\vec{\tau}_{\texttt{state}}|-\texttt{windowSize}+1; t=s+\texttt{windowSize}-1,\dots,|\vec{\tau}_{\texttt{state}}|} : \frac{t-s+1}{\vec{\tau}_{\texttt{state}}[t]-\vec{\tau}_{\texttt{state}}[s]+1}$
while $hr \leq \frac{\texttt{windowSize}}{\texttt{minTime}_{\texttt{window}}}$ **do**
    if $c > 0$ **then**
        Set $\vec{N}_c \leftarrow \texttt{tx}^{\text{coin-base}}_{\text{minerID}}$ of an honest miner
    $\vec{N}_{\texttt{df}} \leftarrow \vec{N}_{\texttt{df}} || \vec{N}_c$
    $\vec{\tau}_{\texttt{state}} \leftarrow \vec{\tau}_{\texttt{state}} || \tau_L$
    $c \leftarrow c + 1$
    $hr \leftarrow \max_{s=1,\dots,|\vec{\tau}_{\texttt{state}}|-\texttt{windowSize}+1; t=s+\texttt{windowSize}-1,\dots,|\vec{\tau}_{\texttt{state}}|} : \frac{t-s+1}{\vec{\tau}_{\texttt{state}}[t]-\vec{\tau}_{\texttt{state}}[s]+1}$
return $\vec{N}_{\texttt{df}}$

Algorithm ExtendPolicy for $\mathcal{G}^{\text{฿}}_{\text{LEDGER}}$ - Part 1

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

We assume call-by-value and hence the function has no side effects.
This Function implements the Extend Policy of the Bitcoin Ledger.

$\vec{N}_{\mathtt{df}} \leftarrow$ DefaultExtension($\vec{\mathcal{I}}_H^T$, $\mathtt{state}$, $\mathtt{NxtBC}$, $\mathtt{buffer}$; s_{ep}) ▷ Extension if adversary violates policy.
Let τ_L be current ledger time (computed from $\vec{\mathcal{I}}_H^T$)
Read $\vec{\tau}_{\mathtt{state}}$ and $\vec{\mathrm{hf}}$ from state s_{ep}. If the state is empty, initialize two empty vectors.
Parse $\mathtt{NxtBC}$ as a vector $((\mathrm{hFlag}_1, \mathtt{NxtBC}_1), \cdots, (\mathrm{hFlag}_n, \mathtt{NxtBC}_n))$
$\vec{N} \leftarrow \varepsilon$ ▷ Initialize Result
if $|\mathtt{state}| \geq \mathtt{windowSize}$ **then** ▷ Determine time of the block which is $\mathtt{windowSize}$ blocks behind the state head
  Set $\tau_{\mathtt{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 1]$
else
  Set $\tau_{\mathtt{low}} \leftarrow 0$
for each list $\mathtt{NxtBC}_i$ of transaction IDs **do** ▷ Compute the next state block and verify validity
  $\vec{N}_i \leftarrow \varepsilon$
  Use the txid contained in $\mathtt{NxtBC}_i$ to determine the list of transactions
  Let $\vec{\mathtt{tx}} = (\mathtt{tx}_1, \ldots, \mathtt{tx}_{|\mathtt{NxtBC}_i|})$ denote the transactions of $\mathtt{NxtBC}_i$
  if $\mathtt{tx}_1$ is not a coin-base transaction **then**
    $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}}||\tau_L|| \ldots ||\tau_L$, $\vec{\mathrm{hf}} \leftarrow \vec{\mathrm{hf}}||1|| \ldots ||1$ (extended by $|\vec{N}_{\mathtt{df}}|$ elements) and store the vectors in s_{ep}.
    return $\vec{N}_{\mathtt{df}}$ and new state s_{ep}
  else
    $\vec{N}_i \leftarrow \mathtt{tx}_1$
    for $j = 2$ to $|\mathtt{NxtBC}_i|$ **do**
      Set $\mathtt{st}_i \leftarrow$ blockify$_{\text{฿}}(\vec{N}_i)$
      if ValidTx$_{\text{฿}}(\mathtt{tx}_j, \mathtt{state}||\mathtt{st}_i) = 0$ **then**
        $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}}||\tau_L|| \ldots ||\tau_L$, $\vec{\mathrm{hf}} \leftarrow \vec{\mathrm{hf}}||1|| \ldots ||1$ (extended by $|\vec{N}_{\mathtt{df}}|$ elements) and store the vectors in s_{ep}.
        return $\vec{N}_{\mathtt{df}}$ and new state s_{ep} ▷ Ignore the adversarial proposal if invalid.
      $\vec{N}_i \leftarrow \vec{N}_i||\mathtt{tx}_j$
    Set $\mathtt{st}_i \leftarrow$ blockify$_{\text{฿}}(\vec{N}_i)$
  $\mathrm{hFlag}_i \leftarrow 1$
  for each $\mathtt{BTX} = (\mathtt{tx}, \mathrm{txid}, \tau', P_i) \in \mathtt{buffer}$ of an honest party P_i with time $\tau' < \tau_{\mathtt{low}} - \frac{\mathtt{Delay}}{2}$ **do**
    if ValidTx$_{\text{฿}}(\mathtt{tx}, \mathtt{state}||\mathtt{st}_i) = 1$ but $\mathtt{tx} \notin \vec{N}_i$ **then**
      $\mathrm{hFlag}_i \leftarrow 0$ ▷ Block is not honestly filled with transactions.
  $\vec{N} \leftarrow \vec{N}||\vec{N}_i$
  $\mathtt{state} \leftarrow \mathtt{state}||\mathtt{st}_i$
  $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}}||\tau_L$, $\vec{\mathrm{hf}} \leftarrow \vec{\mathrm{hf}}||\mathrm{hFlag}_i$ and store those vectors in s_{ep}.
  if $|\mathtt{state}| \geq \mathtt{windowSize}$ **then**
    Set $\tau_{\mathtt{low}} \leftarrow \vec{\tau}_{\mathtt{state}}[|\mathtt{state}| - \mathtt{windowSize} + 1]$
  else
    Set $\tau_{\mathtt{low}} \leftarrow 0$
if $\tau_L < \mathtt{maxTime}_{\mathtt{window}} \wedge \mathtt{state} = \varepsilon$ **then**
  return ε
⋮
See Part 2

Algorithm ExtendPolicy for $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$ - Part 2

⋮

▷ Lowest growth rate, highest growth rate (if applicable), and adversarial rate are within their bounds.
inv ← 0 ▷ invalid flag
if $|\mathtt{state}| < \mathtt{windowSize}$ **then** ▷ Ensure a timely startup with enough honest blocks.
  $lr \leftarrow \min_{j=0,\ldots,\tau_L-\mathtt{maxTime}_{\mathtt{window}};i=j+\mathtt{maxTime}_{\mathtt{window}}\ldots\tau_L} : \frac{\mathtt{windowSize}+|\{n\in[|\vec{\tau}_{\mathtt{state}}|] : j\leq\vec{\tau}_{\mathtt{state}}[n]\leq i\}|}{i-j+1}$
  $ar \leftarrow \frac{|\{n\in[|\mathtt{state}|] : \vec{\mathrm{hf}}[n]=0\}|}{\mathtt{windowSize}}$
  if $lr < \frac{\mathtt{windowSize}}{\mathtt{maxTime}_{\mathtt{window}}} \vee ar > \eta$ **then**
    inv ← 1
if $|\mathtt{state}| \geq \mathtt{windowSize}$ **then** ▷ Ensure ledger growth limits and enough honest blocks.
  $lr \leftarrow \min_{s=1,\ldots,|\vec{\tau}_{\mathtt{state}}|-\mathtt{windowSize}+1;t=s+\mathtt{windowSize}-1,\ldots,|\vec{\tau}_{\mathtt{state}}|} : \frac{t-s+1}{\vec{\tau}_{\mathtt{state}}[t]-\vec{\tau}_{\mathtt{state}}[s]+1}$
  $hr \leftarrow \max_{s=1,\ldots,|\vec{\tau}_{\mathtt{state}}|-\mathtt{windowSize}+1;t=s+\mathtt{windowSize}-1,\ldots,|\vec{\tau}_{\mathtt{state}}|} : \frac{t-s+1}{\vec{\tau}_{\mathtt{state}}[t]-\vec{\tau}_{\mathtt{state}}[s]+1}$
  $ar \leftarrow \max_{s=1,\ldots,|\mathtt{state}|-\mathtt{windowSize}+1;t=s+\mathtt{windowSize}-1,\ldots,|\mathtt{state}|} : \frac{|\{n\in[s\ldots t] : \vec{\mathrm{hf}}[n]=0\}|}{t-s+1}$
  if $lr < \frac{\mathtt{windowSize}}{\mathtt{maxTime}_{\mathtt{window}}} \vee hr > \frac{\mathtt{windowSize}}{\mathtt{minTime}_{\mathtt{window}}} \vee ar > \eta$ **then**
    inv ← 1
if inv = 0 **then**
  return $\vec{N}$ and new state s_{ep}
else
  $\vec{\tau}_{\mathtt{state}} \leftarrow \vec{\tau}_{\mathtt{state}}||\tau_L||\ldots||\tau_L$, $\vec{\mathrm{hf}} \leftarrow \vec{\mathrm{hf}}||1||\ldots||1$ (extended by $|\vec{N}_{\mathtt{df}}|$ elements) and store the vectors in s_{ep}.
  return $\vec{N}_{\mathtt{df}}$ and new state s_{ep}

C. The Simulator of the Main Theorem

The simulator interacts with the backdoor tapes of the ideal protocol $\mathsf{M}[\mathcal{G}_{\text{LEDGER}}^{\text{B}}, \mathcal{G}_{\text{CLOCK}}]$ (to give instructions and receive replies), and since these are two ideal processes, only the backdoor tape of the functionalities $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$ and $\mathcal{G}_{\text{CLOCK}}$ are relevant. Note that technically, communication to these backdoor tapes is accomplished via the backdoor tape of a special shell $\mathsf{sh}[\mathcal{G}_{\text{LEDGER}}^{\text{B}}]$, where by definition this allows direct interaction between the simulator and the backdoor tape of the ITI running inside the shell (this holds analogously for the real-world protocol and the real-world adversary).

Simulator $\mathcal{S}_{\text{ledg}}$

Initialization:
The simulator manages internally a simulated state-exchange functionality $\mathcal{F}_{\text{STX}}$, a simulated network $\mathcal{F}_{\text{N-MC}}$. An honest miner P registered to $\mathcal{G}^{₿}_{\text{LEDGER}}$ is simulated as registered in all simulated functionalities. Moreover, the simulator maintains the local state $\vec{\texttt{st}}_P$ and the buffer of transactions $\texttt{buffer}_P$ 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, sid$_C$, P) from $\mathcal{G}_{\text{CLOCK}}$, 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}, τ) for this party. And in case τ is a working mini-round, execute EXTENDLEDGERSTATE before sending the final (CLOCK-UPDATE, sid$_C$, 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, sid$_C$, 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 interruptible computation of) SIMULATEMINING(P_{minerID}, τ) for this party.
  2. If P is not the identity of the ledger functionality, then just output (CLOCK-UPDATE, sid$_C$, P) to $\mathcal{A}$.

Messages 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 $I =$ (READ, sid) in this round, the fetch-information step of procedure SIMULATEMINING before proceedings with the specific actions below.
- Upon receiving (SUBMIT, BTX) from $\mathcal{G}^{₿}_{\text{LEDGER}}$ where BTX := ($\texttt{tx}$, txid, τ, P) forward (MULTICAST, sid, $\texttt{tx}$) to the simulated network $\mathcal{F}_{\text{N-MC}}$ in the name of P. Output the answer of $\mathcal{F}_{\text{N-MC}}$ to the adversary.
- Upon receiving (MAINTAIN-LEDGER, sid, minerID) from $\mathcal{G}^{₿}_{\text{LEDGER}}$, 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}, τ) for this party.

Simulation of the State Exchange Functionality:

- Upon receiving (SUBMIT-NEW, sid, $\vec{\texttt{st}}$, $\texttt{st}$) from $\mathcal{A}$ on behalf of a corrupted $P \in \mathcal{P}_{stx}$, then relay it to the simulated $\mathcal{F}_{\text{STX}}$ and do the following:
  1. If $\mathcal{F}_{\text{STX}}$ returns (SUCCESS, B) give this reply to $\mathcal{A}$
  2. If $\mathcal{A}$ replies with (CONTINUE, sid), input (CONTINUE, sid) to the simulated $\mathcal{F}_{\text{STX}}$
  3. 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 $\mathcal{F}_{\text{STX}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (SEND, sid, s, P') from $\mathcal{A}$ on behalf some *corrupted* party P, do the following:
  1. Forward the request to the simulated $\mathcal{F}_{\text{STX}}$.
  2. If the current mini-round is an update mini-round, then execute EXTENDLEDGERSTATE
  3. Return to $\mathcal{A}$ the return value from $\mathcal{F}_{\text{STX}}$.
- Upon receiving (SWAP, sid, mid, mid') from $\mathcal{A}$, forward the request to the simulated $\mathcal{F}_{\text{STX}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (DELAY, sid, T, mid) from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\text{STX}}$ and do the following:
  1. Query the ledger state $\texttt{state}$
  2. Execute ADJUSTVIEW($\texttt{state}$)

3. Return to $\mathcal{A}$ the output of $\mathcal{F}_{\text{STX}}$

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 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 for each transaction the transaction id txid
  2. Forward the request to the internally simulated $\mathcal{F}_{\text{N-MC}}$, which replies for each message with a message-ID mid
  3. Remember the association between each mid and the corresponding 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, sid) from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$ forward the request to the simulated $\mathcal{F}_{\text{N-MC}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (DELAYS, 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 $\mathcal{F}_{\text{N-MC}}$ 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 $\mathcal{F}_{\text{N-MC}}$ 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 same steps as above upon a (CLOCK-UPDATE, sid_C, P) from $\mathcal{G}_{\text{CLOCK}}$.

```
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 st using the transaction set buffer_P
      -Send (SUBMIT-NEW, sid, st⃗_P, st) to simulated functionality F_STX
      -If successful, store st⃗_P||st as the new st⃗_P
      -If successful, distribute the new state via F_STX
      -If done with all actions, the last action is outputting (CLOCK-UPDATE, sid_C, P) to 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 (tx_1, ..., tx_u) (on behalf of P) from
       simulated F_N-MC and add them to buffer_P
      -Fetch states st⃗_1, ..., st⃗_s (on behalf of P) from the simulated
       F_STX and update st⃗_P to the largest state among st⃗_P and st⃗_i
      -If done with all actions, the last action is outputting (CLOCK-UPDATE, sid_C, P) to A
```

procedure EXTENDLEDGERSTATE
  Consider all honest and synchronized 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 $\vec{\mathtt{st}}^{\lceil T}$ with the current state $\mathtt{state}$ of the ledger
  if $|\mathtt{state}| > |\vec{\mathtt{st}}^{\lceil T}|$ **then**
    Execute ADJUSTVIIEW($\mathtt{state}$)
  if $\mathtt{state}$ is not a prefix of $\vec{\mathtt{st}}^{\lceil T}$ **then**
    Abort the simulation (due to inconsistency)
  Define the difference diff to be the block sequence s.t. $\mathtt{state}||\mathsf{diff} = \vec{\mathtt{st}}^{\lceil T}$.
  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 an txid, then submit it to the ledger
      and receive the corresponding txid from $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$
    Let $\mathsf{list}_j = (\text{txid}_{j,1}, \ldots, \text{txid}_{j,\ell_j})$ be the corresponding list for this block.
    Output (NEXT-BLOCK, list_j) to $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$ (receiving (NEXT-BLOCK, *ok*) as an immediate answer)
  Execute ADJUSTVIEW($\mathtt{state}||\mathsf{diff}$)
procedure ADJUSTVIEW($\mathtt{state}$)
  $\mathsf{pointers} \leftarrow \varepsilon$
  for each honest and synchronized party P_i **do**
    Using the simulated functionality $\mathcal{F}_{\text{STX}}$ 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. $\vec{\mathtt{st}}^{\lceil T} = \mathsf{state}|_{\mathtt{pt}_i}$
    if such a pointer value does not exist **then**
      Abort simulation (due to inconsistency)
    if Party P_i has not executed step 3 of the mining protocol in this current mini-round **then** ▷ As otherwise, the new state is only fetched in the next round
      $\mathsf{pointers} \leftarrow \mathsf{pointers}||(P_i, \mathtt{pt}_i)$
  Output (SET-SLACK, $\mathsf{pointers}$) to $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$
  $\mathsf{pointers} \leftarrow \varepsilon$
  $\mathsf{desyncStates} \leftarrow \varepsilon$
  for each honest but de-synchronized party P_i **do**
    Using the simulated functionality $\mathcal{F}_{\text{STX}}$ 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** ▷ As otherwise, the new state is only fetched in the next round
      Set the pointer $\mathtt{pt}_i$ to be $|\vec{\mathtt{st}}^{\lceil T}|$
      $\mathsf{pointers} \leftarrow \mathsf{pointers}||(P_i, \mathtt{pt}_i)$
      $\mathsf{desyncStates} \leftarrow \mathsf{desyncState}||(P_i, \vec{\mathtt{st}}^{\lceil T})$
    Output (SET-SLACK, $\mathsf{pointers}$) to $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$
    Output (DESYNC-STATE, $\mathsf{desyncStates}$) to $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$

References

[1] M. Andrychowicz, S. Dziembowski, Pow-based distributed cryptography with no trusted setup, in R. Gennaro, M. Robshaw, editors, *Advances in Cryptology—CRYPTO 2015* (Springer, Berlin, 2015), pp. 379–399

[2] M. Andrychowicz, S. Dziembowski, D. Malinowski, L. Mazurek, Fair two-party computations via bitcoin deposits, in R. Böhme, M. Brenner, T. Moore, M. Smith, editors, *Financial Cryptography and Data Security* (Springer, Berlin, 2014), pp. 105–121

[3] M. Andrychowicz, S. Dziembowski, D. Malinowski, L. Mazurek, Secure multiparty computations on bitcoin. *Commun. ACM* **59**(4), 76–84 (2016)

[4] C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, V. Zikas, Universal composition with global subroutines: Capturing global setup within plain uc, in R. Pass, K. Pietrzak, editors, *Theory of Cryptography* (Springer, Cham, 2020), pp. 1–30

[5] M. Babaioff, S. Dobzinski, S. Oren, A. Zohar, On bitcoin and red balloons. *SIGecom Exch.* **10**(3), 5–9 (2011)

[6] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, V. Zikas, Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability, in D. Lie, M. Mannan, M. Backes, X. Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (ACM, 2018), pp. 913–930

[7] C. Badertscher, P. Gaži, A. Kiayias, A. Russell, V. Zikas, Consensus redux: Distributed ledgers in the face of adversarial supremacy. *Cryptology* ePrint Archive, Paper 2020/1021 (2020). https://eprint.iacr.org/2020/1021

[8] M. Backes, D. Hofheinz, J. Müller-Quade, D. Unruh, On fairness in simulatability-based cryptographic systems, in *Proceedings of the 2005 ACM Workshop on Formal Methods in Security Engineering, FMSE'05* (ACM, New York, NY, USA, 2005), pp. 13–22

[9] C. Badertscher, J. Hesse, V. Zikas, On the (ir)replaceability of global setups, or how (not) to use a global ledger, in K. Nissim, B. Waters, editors, *Theory of Cryptography* (Springer, Cham, 2021), pp. 626–657

[10] I. Bentov, R. Kumaresan, How to use bitcoin to design fair protocols, in J.A. Garay, R. Gennaro, editors, *Advances in Cryptology—CRYPTO 2014* (Springer, Berlin, 2014), pp. 421–439

[11] C. Badertscher, U. Maurer, D. Tschudi, V. Zikas, Bitcoin as a transaction ledger: A composable treatment, in J. Katz, H. Shacham, editors, *Advances in Cryptology—CRYPTO 2017* (Springer, Cham, 2017), pp. 324–356

[12] V. Buterin, *A next-generation smart contract and decentralized application platform*. White Paper on GitHub (2013). https://github.com/ethereum/wiki/wiki/White-Paper

[13] R. Canetti, Universally composable security: A new paradigm for cryptographic protocols, in *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science, FOCS'01* (IEEE Computer Society, Washington, DC, USA, 2001), p. 136

[14] R. Canetti, Universally composable security. *J. ACM* **67**, 5 (2020)

[15] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, G. Neven, The wonderful world of global random oracles, in J.B. Nielsen, V. Rijmen, editors, *Advances in Cryptology—EUROCRYPT 2018* (Springer, Cham, 2018), pp. 280–312

[16] R. Canetti, Y. Dodis, R. Pass, S. Walfish, Universally composable security with global setup, in S.P. Vadhan, editor, *Theory of Cryptography* (Springer, Berlin, 2007), pp. 61–85

[17] T.-H.H. Chan, N. Ephraim, A. Marcedone, A. Morgan, R. Pass, E. Shi, Blockchain with varying number of players. *Cryptology* ePrint Archive, Paper 2020/677 (2020)

[18] S. Coretti, J. Garay, M. Hirt, V. Zikas, Constant-round asynchronous multi-party computation based on one-way functions, in J.H. Cheon, T. Takagi, editors, *Advances in Cryptology—ASIACRYPT 2016* (Springer, Berlin, 2016), pp. 998–1021

[19] J. Chen, S. Micali, Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.* **777**, 155–183 (2019). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I.

[20] A. Dembo, S. Kannan, E.N. Tas, D. Tse, P. Viswanath, X. Wang, O. Zeitouni, Everything is a race and nakamoto always wins, in J. Ligatti, X. Ou, J. Katz, G. Vigna, editors, *CCS'20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9–13, 2020* (ACM, 2020), pp. 859–878

[21] D. Dubhashi, A. Panconesi, *Concentration of Measure for the Analysis of Randomized Algorithms*, 1st edn. (Cambridge University Press, New York, 2009)

[22] I. Eyal, E.G. Sirer, Majority is not enough: Bitcoin mining is vulnerable. *Commun. ACM* **61**(7), 95–102 (2018)

[23] I. Eyal, The miner's dilemma, in *2015 IEEE Symposium on Security and Privacy* (2015), pp. 89–103

[24] J. Garay, A. Kiayias, N. Leonardos, The bitcoin backbone protocol: Analysis and applications, in E. Oswald, M. Fischlin, editors, *Advances in Cryptology—EUROCRYPT 2015* (Springer, Berlin, 2015), pp. 281–310

[25] J.A. Garay, A. Kiayias, N. Leonardos, The bitcoin backbone protocol with chains of variable difficulty, in J. Katz, 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* (Springer, 2017), pp. 291–323

[26] J.A. Garay, A. Kiayias, N. Leonardos, How does nakamoto set his clock? Full analysis of nakamoto consensus in bounded-delay networks. *Cryptology* ePrint Archive, Paper 2020/277 (2020). https://eprint.iacr.org/2020/277

[27] P. Gazi, A. Kiayias, A. Russell, Tight consistency bounds for bitcoin, in J. Ligatti, X. Ou, J. Katz, G. Vigna, editors, *CCS'20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9–13, 2020* (ACM, 2020), pp. 819–838

[28] R. Kumaresan, I. Bentov, How to use bitcoin to incentivize correct computations, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14* (ACM, New York, NY, USA, 2014), pp. 30–41

[29] R. Kumaresan, I. Bentov, Amortizing secure computation with penalties, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS'16* (ACM, New York, NY, USA, 2016), p418–429p.

[30] T. Kerber, A. Kiayias, M. Kohlweiss, Composition with knowledge assumptions, in T. Malkin, C. Peikert, editors, *Advances in Cryptology—CRYPTO 2021—41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV, volume 12828 of Lecture Notes in Computer Science* (Springer, 2021), pp. 364–393

[31] T. Kerber, A. Kiayias, M. Kohlweiss, KACHINA—foundations of private smart contracts, in *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21–25, 2021* (IEEE, 2021), pp. 1–16

[32] A. Kiayias, E. Koutsoupias, M. Kyropoulou, Y. Tselekounis, Blockchain mining games, in *Proceedings of the 2016 ACM Conference on Economics and Computation, EC'16* (ACM, New York, NY, USA, 2016), pp. 365–382

[33] R. Kumaresan, T. Moran, I. Bentov, How to use bitcoin to play decentralized poker, in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS'15* (ACM, New York, NY, USA, 2015), pp. 195–206

[34] J. Katz, U. Maurer, B. Tackmann, V. Zikas, Universally composable synchronous computation, in A. Sahai, editor, *Theory of Cryptography* (Springer, Berlin, 2013), pp. 477–498

[35] R. Kumaresan, V. Vaikuntanathan, P.N. Vasudevan, Improvements to secure computation with penalties, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS'16* (ACM, New York, NY, USA, 2016), pp. 406–417

[36] A. Kiayias, H.-S. Zhou, V. Zikas, Fair and robust multi-party computation using a global transaction ledger, in M. Fischlin, J.-S. Coron, editors, *Advances in Cryptology—EUROCRYPT 2016* (Springer Berlin, 2016), pp. 705–734

[37] L. Lamport, The part-time parliament. *ACM Trans. Comput. Syst.* **16**(2), 133–169 (1998)

[38] L. Lamport, Paxos made simple, fast, and byzantine, in *Proceedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11–13, 2002* (Suger, Saint-Denis, rue Catulienne, 2002), pp. 7–9

[39] L. Lamport, R. Shostak, M. Pease, The byzantine generals problem. *ACM Trans. Program. Lang. Syst.* **4**(3), 382–401 (1982)

[40] I. Miers, C. Garman, M. Green, A.D. Rubin, Zerocoin: Anonymous distributed e-cash from bitcoin, in *2013 IEEE Symposium on Security and Privacy* (2013), pp. 397–411

[41] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*. White Paper (2008). http://bitcoin.org/bitcoin.pdf

[42] S. Park, A. Kwon, G. Fuchsbauer, P. Gazi, J. Alwen, K. Pietrzak, Spacemint: A cryptocurrency based on proofs of space, in S. Meiklejohn, K. Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers, volume 10957 of Lecture Notes in Computer Science* (Springer, 2018), pp. 480–499

[43] R. Pass, E. Shi, Fruitchains: A fair blockchain, in *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC'17* (ACM, New York, NY, USA, 2017), pp. 315–324

[44] R. Pass, L. Seeman, A. Shelat, Analysis of the blockchain protocol in asynchronous networks, in J.-S. Coron, J.B. Nielsen, editors, *Advances in Cryptology—EUROCRYPT 2017* (Springer, Cham, 2017), pp. 643–673

[45] M.O. Rabin, Randomized byzantine generals, in *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)* (1983), pp. 403–409

[46] E.B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, M. Virza, Zerocash: Decentralized anonymous payments from bitcoin, in *2014 IEEE Symposium on Security and Privacy* (2014), pp. 459–474

[47] Y. Sompolinsky, A. Zohar, Secure high-rate transaction processing in bitcoin, in R. Böhme, T. Okamoto, editors, *Financial Cryptography and Data Security* (Springer, Berlin, 2015), pp. 507–527

[48] A. Zohar, Bitcoin: Under the hood. *Commun. ACM* **58**(9), 104–113 (2015)