Bitcoin as a Transaction Ledger: A Composable Treatment

Christian Badertscher[1], Ueli Maurer[1], Daniel Tschudi[1], and Vassilis Zikas[2]

[1] {christian.badertscher,maurer,tschudid}@inf.ethz.ch, ETH Zurich
[2] vzikas@inf.ethz.ch, RPI

Abstract. Bitcoin is perhaps the most prominent example 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 a ledger shared-functionality, aka global setup, in the (G)UC model of Canetti et al. [TCC'07]. Our ledger functionality is weaker than the one recently proposed by Kiayias, Zhou, and Zikas [EUROCRYPT'16], but unlike the latter suggestion which is arguably not implementable given the Bitcoin assumptions, we prove that the one proposed here is securely UC realized under standard assumptions by an appropriate abstraction of Bitcoin as a UC protocol. 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 (G)UC without restricting the environment or the adversary.

1 Introduction

Since Nakamoto first proposed Bitcoin as a decentralized cryptocurrency [28] several works have focused on analyzing and/or predicting its behavior under different attack scenarios [4, 14, 13, 34, 33, 18, 30]. However, a core question remained:

What security goal does Bitcoin achieve and under what assumptions?

An intuitive answer to this question was already given in Nakamoto's original white paper [28]: Bitcoin aims to achieve some form of consensus on a set of valid transactions. The core difference of this consensus mechanism with traditional consensus [26, 24, 25, 31] is that it does not rely on having a known (permissioned) set of participants, but everyone can join and leave at any point. (This is often referred to as the *permissionless* model.) This is achieved by shifting from the traditional assumptions on the number of cheaters compared to the total number of participants to assumptions on the total (i.e., collective) computing power of the cheaters in comparison 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.)

Despite however the fact that glimpses of this idea were implicit already in the original white paper, a formal description of this goal had not been proposed or known to be achieved (and under what assumptions) until the recent works of Garay, Kiayas, and Leonardos [15] and Pass, Seeman, and shelat [29]. In a nutshell these works set forth models of computation and, in these models, an abstraction of Bitcoin as a cryptographic distributed protocol, and proved that the output of this protocol satisfies an appropriate security properties, e.g., the *common prefix* [15] or consistency [29] property. This property confirms—under the assumption that not too much of the total computing power of the system is invested in breaking it—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 in the blockchain of every honest party and will never be reversed.[3] In addition

[3] 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.

to the common prefix property, other quality properties of the output of the abstracted blockchain protocol are also defined and proved. (A more detailed description of the security properties and a comparison of the assumptions in [15] and [29] is included in Section 4.4).

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 economic forces that guide its sustainability are yet to be understood, and formal rational models predict it is not a stable solution, the fact is that Bitcoin has not yielded to any of these pessimistic predictions for several years and it is not clear it will ever do. And even if it does, the research community has produced and is testing several alternative decentralized cryptocurrencies, e.g., [27, 6, 8] that are more functional and/or resilient to theoretic attacks than Bitcoin.

Thus, it is reasonable to assume that decentralized cryptocurrencies are here to stay. This leads to the natural questions of how can we use this new reality to improve the security and/or efficiency of cryptographic protocol? First answers to this question were given in [2, 3, 7, 23, 21, 22, 20, 1] 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 construction in a widely accepted cryptographic frameworks for multi-party protocols, one needs to define what it means for these protocols to leave in a world that gives them access to Bitcoin 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 [7] introduced a model of computation in which protocols can use a punishment mechanism to incentivize adversaries to adhere to their protocol instruction. [7] uses as basis the Universal composition framework of Canetti [10], but the proposed modifications do not support composition and it is not clear how standard UC cryptographic protocols can be cast as protocols in this model.

In a different direction, Kiayias, Zhou, and Zikas [19] connected the above question with the original question of what the security goal of Bitcoin is. More concretely, they proposed identifying the resource that Bitcoin (or more general of decentralized cryptocurrencies) offer to cryptographic protocols as its security goal, and capturing it in a standard way in which such resources/goals are captured in the cryptographic multi-party protocols literature, i.e., by means of an ideal transaction-ledger functionality in the universal composition framework. (To ensure a most accurate abstraction, the ledger of [19] is formally a global setup in the (extended) GUC framework of Canetti et al. [11].)

In a nutshell, the ledger proposed by [19] corresponds to a trusted third party which keeps a state of blocks of transactions which is 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 (which not makes them available to everyone.) As proved in [19], giving multi-party protocols access to such a transaction-ledger functionality allows for formally capturing, within the (G)UC framework, the mechanism of leveraging security loss with coins. The proposed ledger functionality guarantees all properties required (and proved) for Bitcoin in [15, 29]. Therefore, it is natural to postulate that it as candidate for defining, in an accurate manner, the security goal of Bitcoin (and potentially other cryptocurrencies). However, the ledger functionality proposed by [19] was not accompanied by a security proof that any of the cryptocurrencies implement it.

In this work, we show that despite being a step in the right direction, the ledger proposed in [19] cannot be realized under standard assumptions about the Bitcoin network. On the positive side, we specify a new transaction ledger functionality (as a global UC setup) which still guarantees all properties that were proved to hold in [15, 29] and prove that a reasonable abstraction of the Bitcoin protocol implements our transaction ledger. In our construction, we describe Bitcoin as a UC protocol which generalizes both the protocols proposed in [15, 29]. Along the way we identify the assumptions in each of [15, 29] by devising a compound way of capturing such assumptions in UC, and compare their strengths.

Related Literature The security of Bitcoin as a cryptographic protocol was previously studied by Garay, Kiayas, and Leonardos [15] and of Pass, Seeman, and shelat [29] who proposed and analyzed an abstraction of the core of Bitcoin protocol in a property-based manner. As such, the treatment of [15, 29] does not offer composable security guarantees. More recently, Kiayias, Zhou, and Zikas [19] proposed capturing the security goal and resource implemented by Bitcoin by means of a shared transaction-ledger functionality in the universal composition with global setup (GUC) framework of Canetti et al. [11]. Alas, as it turns out, the proposed ledger-functionality is too strong to be implementable by Bitcoin. We refer the interested reader to Section A of the appendix for the basic elements of these works, where we also discuss simulation-based security and its advantages. A formal comparison of our treatment with [15, 29] which indicates how both these protocols and definitions can be captures as special cases of our security definition is also given in Section 4.4.

Our Results We put forth the first fully (universally) composable (simulation-based) proof of security of Bitcoin in the (G)UC model of Canetti et at. [11]. We observe that the ledger functionality proposed by Kiayas et al. [19] is too strong to be implemented by the Bitcoin protocol (in fact, by any protocol in the permissionless setting which uses network assumptions similar to Bitcoin); intuitively, the reason is 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 length.

Therefore, we propose an alternative ledger functionality $\mathcal{G}_{\text{LEDGER}}$ which has properties similar to that of [19] but which can be provably implemented by the Bitcoin protocol. As in [19], our proposed functionality is in fact a global setup—i.e., it allows protocols with different sessions to make use of it, thereby enabling the ledger to be cast as shared among any protocol that wants to use it—and it is parametrized by a generic transaction validation predicate which allows it to capture many decentralized blockchain protocols. Our functionality allows for parties/miners to join and or leave the computation and allows for adaptive corruption.

Having defined our ledger functionality we next turn to proving that for an appropriate validation predicate it is implemented by Bitcoin assuming that miners which deviate from the Bitcoin protocol do not control a majority of the total hushing power in the network at any point. To this direction, we describe an abstraction of the Bitcoin protocol as a synchronous UC protocol. Our protocol construction follows a structure which generalized both [15, 29]; in fact, as we argue the protocols described in these works can be captured as instances of our protocols; the difference between these two instances is the network assumption that is used—more precisely, the assumption about knowledge or not of the network delay—and the assumption on number of queries per round. To capture these assumptions in UC, we devise a methodology that treats functionality wrappers as such assumptions and discuss the implications of such a method in preserving universal composability.

We design our protocol to work over a network of bounded-delivery asynchronous channels, where similar to the protocol in [29], the miners are not aware of the actual delay that the network induces. We prove that such a network is strictly weaker than a network with known bounded delay which, as we argue, is (implicit) in the synchrony assumptions of [15]. Notwithstanding, unlike previous works, instead of starting from a complete network of asynchronous multicast channels with eventual delivery, we show that such a network is implemented by running the message diffusion mechanism of Bitcoin over a lower level network. Intuitively, this network is build by every miner, upon joining the system, choosing some existing miners of its choice and using them as relayers.

Our security proof uses a nice modularization of the Bitcoin protocol. Concretely, first we identify a process of the protocol which can be argued to implement an appropriate UC functionality—that, intuitively, abstracts the lottery aspect of the protocol—and prove that a chunk of the Bitcoin code is in fact an implementation of this functionality. We then analyze the remainder of the protocol in the simpler world where the code-module implementing this process is replaced by it corresponding functionality. Using the UC composition theorem, we can then immediately combine the two parts into a proof of the full protocol.

As is the case with the so called ”backbone” protocol from [15] our above UC protocol description of Bitcoin relies only on proofs of work and not on digital signatures. As a results, we also get a somewhat weaker ledger, which does not guarantee that transactions submitted by honest parties will eventually make it in the blockchain. As a last result, we show that (similarly to [15]) by assuming public key crypto and 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—we can implement a stronger ledger which ensures honest transactions to eventually make it. The fact that our protocol is described in UC makes this a straight-forward application of our transaction ledger, since we do not need to consider the specific of the Bitcoin protocol in the proof, and it suffices to replace it with our ledger functionality. This also allows us to identify the maximum 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.

2 A Composable Model for Blockchain Protocols in the Permissionless Model

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

Bitcoin miners are represented as players—formally Interactive Turing Machine instances (ITIs)—in a multi-party computation that 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 (G)UC, the resources available to the parties are described as ideal functionalities. 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.

Functionalities with dynamic party sets In many UC functionalities, the set of parties is defined upon initiation of the functionality and is not change throughout the lifecycle of the functionality. Nonetheless, UC does support for functionalities in which the set of parties that might interact with the functionality is dynamic. In fact, this dynamic nature is an inherent property of 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 following three instructions that allow honest parties to join or leave the set $\mathcal{P}$ of players that the functionality interacts with, and inform the adversary about the current set of registered parties:[4]

- Upon receiving $(\text{REGISTER}, \mathsf{sid})$ from some party p_i (or from $\mathcal{A}$ on behalf of some p_i), set $\mathcal{P} = \mathcal{P} \cup \{p_i\}$. Return $(\text{REGISTER}, \mathsf{sid}, p_i)$ to the caller.
- Upon receiving $(\text{DE-REGISTER}, \mathsf{sid})$ from some party $p_i \in \mathcal{P}$, set $\mathcal{P} := \mathcal{P} \setminus \{p_i\}$. Return $(\text{DE-REGISTER}, \mathsf{sid}, p_i)$ to p_i.
- Upon receiving $(\text{GET-REGISTERED}, \mathsf{sid})$ from the adversary $\mathcal{A}$, return $(\text{GET-REGISTERED}, \mathsf{sid}, \mathcal{P})$ to $\mathcal{A}$.

For simplicity in the description functionality, for a party $p_i \in \mathcal{P}$ we will use p_i to refer this party's ID. In addition to the above, global setups—these are GUC functionalities that are available both in the real and in the ideal world and allow parties connected to them to share state [11]—might also allow UC functionalities to registered with them.[5] Concretely, global setups include, in addition to the above party registration instruction, two functionality registration/de-registration instructions:

- Upon receiving $(\text{REGISTER}, \mathsf{sid_C})$ from a functionality $\mathcal{F}$, set $F := F \cup \{\mathcal{F}\}$.
- Upon receiving $(\text{DE-REGISTER}, \mathsf{sid_C})$ from a functionality $\mathcal{F}$, set $F := F \setminus \{\mathcal{F}\}$.

[4] Note that making the set of parties dynamic means that the adversary need to be informed about which parties are currently in the computation so that he can chose which/how-many parties to corrupt.

[5] Although we do not allow communication between functionalities, we will allow functionalities to communicate with global setups as additional honest parties, as these are anyway open to interaction with the environment.

The above two (or four in case of global setups) instructions will be part of the code of *all* ideal functionalities considered in this work. However, to keep the description simpler we will omit these instructions from the formal descriptions. We are now ready to formally describe each of the available functionalities.

The Communication Network In Bitcoin, parties/miners communicate over an incomplete network of asynchronous unauthenticated unidirectional channels. Concretely, every miner chooses a set of other miners as its immediate neighbors—typically by using some public information on IP addresses of common miner—and uses its neighbors to send messages to all the miners in the Bitcoin network—i.e., multicast the message[6]—via a standard diffusion mechanism: 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. We make the following two assumptions on the communication channels in the above diffusion mechanism/protocol:

– They are asynchronous but they guarantee (reliable) delivery of messages within a delay parameter Δ. I.e., the adversary might delay any message sent through such a channel but only by a bounded amount Δ (in particular, the adversary cannot block messages). This means that the adversary can arbitrary re-order messages sent to some party.
– The receiver gets no information other than the messages themselves. In particular, he 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 guarantee—i.e., the adversary is given read access to all messages sent to the network.

Our formal description of communication with eventual delivery within the UC framework builds on ideas from [17, 5, 12]. In particular, we capture such communication by assuming for each miner $p_j \in \mathcal{P}$ a multi-use *unicast* channel with receiver p_j, where any miner $p_i \in \mathcal{P}$ can connect-to and input a messages to be delivered to $p_j \in \mathcal{P}$. A miner connecting to the unicast channel with receiver p_j corresponds to the above process of looking up p_j and making him one of its access points. The unicast channel does not give any information its receiver about who else is using it. In particular, messages are buffered but the information of who is the sender is deleted; instead, the channel creates unique independent messages IDs that are used as handles for the messages. Furthermore, 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 is allows to deliver them in an arbitrary out-of-order manner.

To ensure that when p_r is honest, the adversary cannot arbitrarily delay the delivery of messages submitted by honest parties, we use the following idea: We first turn the UC secure channels functionality to work in a "fetch message" mode, where the channel delivers the message to its intended recipient p_j if and only if p_j asks to receive it by issuing a special "fetch" command. If the adversary wishes to delay the delivery of some message with message ID mid, he needs to submit to the channel functionality an integer value T_{mid}—the *delay* for message with ID mid. This will have the effect of the channel ignoring the first T_{mid} fetch attempts following the reception of the sender's message. Importantly, we require that the channel does not accept more than Δ accumulative delay on any message. To allow the adversary freedom in scheduling 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; in order, however, to ensure that the adversary is able to re-order such batches of messages arbitrarily, we allow $\mathcal{A}$ to send special $(\mathsf{swap}, \mathsf{mid}, \mathsf{mid}')$ commands that have as an effect to change the order of the corresponding messages. The detailed specification of the above asynchronous channels, denoted $\mathcal{F}_{\text{A-CH}}$ is given in Appendix B. Note that in the descriptions throughout the paper, for a vector $\vec{M}$ we denote by $||$ the operation which adds a new element to $\vec{M}$.

[6] In [15] this mechanism is referred to as "broadcast"; here, we use multicast to make explicit the fact that this primitive is different from a standard Byzantine-agreement-type broadcast, in that it does not guarantee any consistency for a malicious sender.

Asynchronous Multicast As already mentioned, the Bitcoin protocol uses the above asynchronous-with-bounded-delivery unicast network to implement a multicast mechanism. A asynchronous multicast functionality with bounded delivery can be defined similar to the above unicast channel. The main difference is that once a message is inserted it is recorded $\mathcal{P}$ times, once for each possible receiver. The adversary can add delays to any subset of messages (but again for any message the cumulative delay cannon exceed Δ) and is allows 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 channel). The detailed specification of the corresponding functionality $\mathcal{F}_{\text{A-MC}}$ is similar to that of $\mathcal{F}_{\text{A-CH}}$ and can be found in Appendix B, where we also show that the simple round-based (cf. the paragraph about synchrony below) diffusion mechanism implements it from $\mathcal{F}_{\text{A-CH}}$ as long as the corresponding network among honest parties is strongly connected (i.e., there is always a directed path between two honest parties where the edges correspond to unicast channels in the direction from sender to receiver).

The Random Oracle As usually in cryptographic proofs, the queries to the hash function are modeled by assuming access to a random oracle functionality. This functionality is specified as follows: upon receiving a query $(\textsc{eval}, \mathsf{sid}, x)$ from a registered party, if x has not been queried before, a value y is chosen uniformly at random from $\{0,1\}^\kappa$ and returned to the party (and the mapping (x, y) is internally stored). IF x has been queried before, the corresponding y is returned. For completeness we include the functionality in B.

Synchrony Katz et al. [17], proposed a methodology for casting synchronous protocols in UC by assuming they have access to an ideal functionality, *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 $(\textsc{clock-read}, \mathsf{sid}_C)$. This value is updated only once all honest parties send the clock a $(\textsc{clock-update}, \mathsf{sid}_C)$ command.

Given such a clock, [17] describe how synchronous protocols can maintain their necessary round structure in UC: For every round ρ each parties first executes all its round-ρ instructions and then sends the clock a CLOCK-UPDATE command. Subsequently, whenever activated, it send 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 rounds ρ. In [19], this idea was transfered to the (G)UC setting, by assuming that the clock is a global setup. This allows for different protocols to use the same clock and is the model we will also use here. For completeness we include the clock functionality in Appendix B.

As argued in [17] in order for an eventual-delivery, aka guaranteed termination, functionality to be UC implementable by a synchronous protocol it needs to keep track of the number of activations that an honest party gets, so that it knows when to generate output for honest parties. This requires that the protocol itself, when described as a UC interactive Turing-machine instance (ITI), has a predictable behavior when it comes to the pattern of activations that it needs before it send the clock an update command. We capture this property in a generic manner in the following definition.

In order to understand the definition, we recall the reader of how activations work in UC. In a UC protocol execution, an honest party (ITI) gets activated either by receiving an input from the environment, or by receiving a message from one of its hybrid-functionalities (or from the adversary). Any activation results in the activated ITI performing some computation on its view of the protocol and its local state and ends with either the party sending a message to some of its hybrid functionalities or sending an output to the environment, or not sending any message. In either of this case, the party looses the activation.[7]

For any given protocol execution, we define the *honest inputs sequence* $\vec{\mathcal{I}}_H$ to consist of all inputs that the environment gives to honest parties in the given execution (in the order that they were given) along with the identity of the party who received the input. I.e., for an execution in which the environment has given m inputs to the honest parties in total, $\vec{\mathcal{I}}_H$ is a vector of the form $((x_1, \mathtt{pid}_1), \ldots, (x_m, \mathtt{pid}_m))$, where x_i is the i-th input that was given in this execution, and $\mathtt{pid}_i$ is the corresponding party who received this input. We further define the *timed honest inputs sequence,* denoted as $\vec{\mathcal{I}}_H^T$, to be the honest inputs sequence augmented with clock times at the point of the input, i.e., i.e., if timed honest input of an execution

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

being $\vec{\mathcal{I}}_H^T = ((x_1, \mathtt{pid}_1, \tau_1), \ldots, (x_m, \mathtt{pid}_m, \tau_m))$ means that $((x_1, \mathtt{pid}_1), \ldots, (x_m, \mathtt{pid}_m))$ is the honest inputs sequence corresponding to this execution, and for each $i \in [n]$ τ_i is the current values of the global clock when input x_i was handed to $\mathtt{pid}_i$.

Definition 1. *A $\mathcal{G}_{\text{CLOCK}}$-hybrid protocol Π has a* predictable synchronization pattern *iff there exist an algorithm $\texttt{predict-time}_\Pi(\cdot)$ such that for any possible execution of Π (i.e., for any adversary and environment, and any choice of random coins) the following holds: If $\vec{\mathcal{I}}_H^T = ((x_1, \mathtt{pid}_1, \tau_1), \ldots, (x_m, \mathtt{pid}_m, \tau_m))$ is the corresponding timed honest inputs sequence for this execution, then for any $i \in [m-1]$: $\texttt{predict-time}_\Pi((x_1, \mathtt{pid}_1, \tau_1), \ldots, (x_i, \mathtt{pid}_i, \tau_i)) = \tau_{i+1}$*

As we argue, all synchronous protocol described in this work are designed to have a predictable synchronization pattern.

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 [15, 29] is that in each round the adversary cannot do more calls to the random oracle than what the honest parties (collectively). This can be captured by assuming a restricted environment and adversary which balances the amount of times that the adversary queries the random oracle. And in a property-based treatment such as [15, 29] this assumptions is typically acceptable.

However, in a simulation-based definition restricting the class of adversaries and environments in a security statement means that we can no longer generically apply the composition theorem which dismisses one of the major advantages of using simulation-based security in the first place. Therefore, instead of restricting the class of environments/adversaries, here we take a different approach to capture the fact that the adversary's access to the RO is restricted with respect to that of honest parties. In particular, we capture this assumption by means of a functionality wrapper, that wraps the RO functionality and forces the above restrictions on the adversary, i.e., gives to each corrupted party at most q activations per round for a parameter q. (To keep track of rounds the functionality registers with the global clock $\mathcal{G}_{\text{CLOCK}}$.) For completeness we include the wrapped random oracle functionality $\mathcal{W}^q(\mathcal{F}_{\text{RO}})$ in Appendix B.

Remark 1 (Functionally Black-box Use of the Network (Delay)). A key difference between the models in [15] and [29] is that in the latter the parties do not know any bound on the delay of the network. In particular, although both models are in the synchronous setting, i.e., in [29] a party in the protocol does not know when to expect a message which was sent to it in the previous round. Using terminology from [32], the protocol uses the channel in a *functionally black-box* manner. Restricting to such protocols—a restriction which we also adopt in this work—is in fact implying a weaker assumption on the protocol than standard (known) bounded delay channel. Intuitively the reason is that no such protocol can realize a bounded-delay network with a known upper bound (unless it sacrifices termination) since the protocol cannot decide whether or not the bound has been reached.

3 The Transaction-Ledger Functionality

In this section we describe our transaction ledger functionality, denoted as $\mathcal{G}_{\text{LEDGER}}$, which we will prove is implemented by (a UC version) of the Bitcoin protocol. As in [19], 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—this is consistent to 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 at list limiting the use of) the clock to allow for less synchrony.

At a high level, our ledger $\mathcal{G}_{\text{LEDGER}}$ has a similar structure as the ledger proposed in [19]. Concretely, anyone (whether an honest miner or the adversary) might submit a transaction transaction which is validated by means of a predicate $\mathsf{Validate}$ and if it is found valid it is added to a buffer buffer. (The adversary $\mathcal{A}$ is informed that the transaction was received and is given its contents.)[8] Informally, this buffer corresponds

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

to transactions that, although validated, they are not yet deep enough in the blockchain so that we can consider them out-of-reach for the adversary.[9] Periodically, $\mathcal{G}_{\text{LEDGER}}$ fetches some of the transactions in the buffer, and using an algorithm Blockify creates a block including these transactions and adds this block to its permanent state state. The state state is a data structure that includes the part of the blockchain which the adversary can no longer change. (This is referred to as the *common prefix* in [15, 29].) Any miner or the adversary is allowed to request a read of the contents of the state.

The above specification is simple, but, in order to have a ledger that can be implemented by existing blockchain protocol, e.g., by the Bitcoin protocol, we need to relax this functionality by giving the adversary more power to interfere with it and influence its behavior. (Recall that as argued in the introduction, the Ledger functionality in [19] is too strong to be implemented by known protocols.) In the following, we review these relaxations.

State-buffer validation. The first relaxation is with respect to the invariant that is enforced by the validation predicate Validate. Concretely, in [19] it is assumed that the validation predicate enforces that the buffer does not include conflicting transactions, i.e., upon reception of a transaction, Validate checks that it does not 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.[10] The only time when it becomes clear which of these conflicting transactions will make it into the state is once it has been added into a block which has made it deep enough in the block-chain (i.e., has become part of state). Hence, given that the buffer includes all transactions that might end up into the state, it might at some point include both conflicting 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 enforces the invariant no single transaction in the buffer contradicts the state state—but different transactions in buffer might contradict each other. (The stronger version that is implemented by adding digital signature, cf. Section 5, enforces also that no transaction submitted by an honest party contradicts other transactions.) As in [19], whenever a new transaction x 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 x 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 revalidation to be generic, transaction that are added to the buffer are accompanied by certain metadata (i.e., the identity of the submitter, a unique transaction ID `txid`, [11] the time τ when x was received.)

State update policies and security guarantees. The second relaxation is with respect to the rate and the form and/or origin of transactions that make it into the block. Concretely, instead of assuming that the state is extended in fixed time intervals, we allow the adversary to define when this update will occur. 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 vector containing the contents of the next block that $\mathcal{A}$ wants to be inserted in 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 (and if anything ever does) yield a very weak ledger. (Intuitively, this would be a ledger that only guarantees the common prefix property [15] 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 yields a default policy-complying block using the current contents of the buffer. We point out

[9] E.g., in [19] the adversary is allowed to permute the contents of the buffer.

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

[11] In Bitcoin, `txid` would be the hash-pointer corresponding to this transaction.

that a good simulator for a ledger-protocol 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 simulators actions always comply with the policies of $\mathtt{ExtendPolicy}$.

In a nutshell, $\mathtt{ExtendPolicy}$ takes the current contents of the buffer buffer, along with the adversary's recommendation $\mathtt{NxtBC}$, and the *block-insertion times vector* $\vec{\tau}_{\mathtt{state}}$ (i.e., the vector listing the times when each block was inserted into state,)[12] and outputs the vector including the actual contents of the block to be inserted during the next state-extend time-slot (where again, $\mathtt{ExtendPolicy}$ outputting an empty vector is a signal to not extend). To ensure that $\mathtt{ExtendPolicy}$ 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 [15]—we also pass to it the the timed honest inputs sequence $\vec{\mathcal{I}}_H^T$ (cf. Section 2).

Some examples of how $\mathtt{ExtendPolicy}$ allows us to define ways that the protocol might restrict the adversary's interference in the state-update include the following properties from [15]:

- *Liveness* corresponds to $\mathtt{ExtendPolicy}$ which enforces 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 $\mathtt{NxtBC}$, $\mathtt{ExtendPolicy}$ can choose some of the transactions in the buffer (e.g., those that have been in the buffer longer) and add them to the next block. Note that a good simulator 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 $\mathtt{ExtendPolicy}$ which enforces the following policy: $\mathtt{ExtendPolicy}$ looks into the blocks of state for a special type of transaction (corresponding to coinbase transaction) and parses the state (using the sequence of of honest inputs $\vec{\mathcal{I}}_H^T$ and the block-insertion times vector $\vec{\tau}_{\mathtt{state}}$) to see how long ago (in time or block-number) the last block that gave a block-mining reward to some honest party was inserted into the state. If this happened "too long" ago (this will be a parameter of this $\mathtt{ExtendPolicy}$), then $\mathtt{ExtendPolicy}$ forces the coinbase transaction of the next block to have miner ID the ID submitted by some (e.g., the last to speak) honest miner.

In addition to the above standard properties, $\mathtt{ExtendPolicy}$ allows us to also capture additional security properties of various blockchain protocols, e.g., the fact that honest transactions eventually make it into a block and the fact that transactions with higher rewards make it into a block faster than others.

In Section 4 where we prove the security of Bitcoin, we will provide the concrete specification of $\mathsf{Validate}$ and $\mathtt{ExtendPolicy}$ for which the Bitcoin protocol realizes our ledger.

Output Slackness. The common prefix property guarantees that blocks which are sufficiently deep into 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 [15]. Thus it is unclear how state can be defined so that at any point all parties have the same view on it.

Therefore, to have an ledger implementable by standard assumptions we make the following relaxation: We interpret 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 state_i of state of length $|\mathsf{state}_i| = \mathtt{pt}_i$ 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 $\mathtt{pt}_i$). For convenience, we denote by $\mathsf{state}|_{\mathtt{pt}_i}$ the subchain of state that finishes in the $\mathtt{pt}_i$-th block. Once again, to avoid over-relaxing the functionality to an unuseful setup our ledger forces the adversary to leave at least one honest miner p_j with $\pi_j := \mathsf{state}$—i.e., p_j will get the whole state as a result of read already in the current time unit—and it forbids the adversary to define pointers for honest miners that are too far apart (i.e., more than $\mathtt{slack}$ time units, where $\mathtt{slack} \in \mathbb{N}$ is a parameter of the ledger.)

[12] We will assume that the transactions are accompanied by such identifiers; to ensure the same level of unlinkability as the Bitcoin network, these IDs are chosen by the parties in an arbitrary way in each transaction (in the actual protocol these would be hashes of a public key).

Synchrony In order to keep the ideal execution indistinguishable from the real execution, the adversary should be unable to use the clock for distinguishing. Since in the ideal world when a dummy party receives a CLOCK-UPDATE-message for the $\mathcal{G}_{\text{CLOCK}}$ it will forward it, the ledger needs to be responsible that the clock counter does not advance before all honest parties have received sufficiently many activations. This is achieved by the use of the function $\texttt{predict-time}(\vec{\mathcal{I}}_H^T)$ (see Definition 1) , which, as we show, is defined for out ledger protocol. This function allows $\mathcal{G}_{\text{LEDGER}}$ to predict when the protocol would update the round and ensure that it only allows the clock to advance if and only if the protocol would (Observe that the ledger sees all protocol relevant inputs/activations to honest parties and can therefore easily keep track of the honest inputs sequence $\vec{\mathcal{I}}_H^T$.)

A final observation is with respect to guarantees that the protocol (and therefore also the ledger) can give to recently registered honest parties. 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—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, the parties might take up to Δ rounds to receive the NEW-MINER notification, and their response might also take up to Δ rounds before it arrives to the new miner. However, because we do not make any assumption on honest parties knowing Δ (see Remark 1) they need to start mining as soon as a message arrives (otherwise they might wait indefinitely). 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 want as the current state while they wait for the network's response: simply delay everything sent to these parties by honest miners by the maximum delay Δ, and instead, immediately deliver what you want them to work on. Thus, for the first $\texttt{Delay} := 2\Delta$ rounds[13] (where $\texttt{Delay}$ is a parameter of our ledger) these parties are practically in the control of the adversary and their computing power is contributed to his. We will call such miners *de-synchronized* and denote the set of such miners by $\mathcal{P}_{DS}$. The formal specification of our ledger functionality $\mathcal{G}_{\text{LEDGER}}$ is given in the following. Using standard notation, we write $[n]$ to denote the set $\{1, \ldots, n\}$.

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

$\mathcal{G}_{\text{LEDGER}}$ is parametrized with four algorithms, $\mathsf{Validate}$, $\texttt{ExtendPolicy}$, $\mathsf{Blockify}$, and $\texttt{predict-time}_{BC}$, along with two parameters: $\texttt{slack}, \texttt{Delay} \in \mathbb{N}$.
The functionality manages variables state, $\texttt{NxtBC}$, buffer, τ_L, and $\vec{\tau}_{\mathsf{state}}$, as described above, initialized as follows: $\mathsf{state} := \vec{\tau}_{\mathsf{state}} := \texttt{NxtBC} := \varepsilon$, $\mathsf{buffer} := \emptyset$, $\tau_L = 1$. It also maintains for each party/miner $p_i \in \mathcal{P}$ a pointer $\texttt{pt}_i$ (initially set to 1) and a current-state view $\mathsf{state}_i := \varepsilon$ (initially set to empty), and also keeps track of the timed honest inputs sequence in a vector $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T := \varepsilon$). Finally, $\mathcal{G}_{\text{LEDGER}}$ maintains the set of registered parties/miners $\mathcal{P}$, the set of honest registered miners $\mathcal{H}$, and a subset $\mathcal{P}_{DS} \subset \mathcal{H}$ of de-synchronized honest registered parties (all are initially set to $\emptyset$). When a new honest party is registered, it is added to all $\mathcal{P}_{DS}$ (hence also to $\mathcal{H}$ and $\mathcal{P}$) and the current time of registration is also recorded; similarly, when a party is deregistered, it is removed from both $\mathcal{P}$ and $\mathcal{P}_{DS}$.

To be executed upon any activation:

- **Upon receiving any input** I from any party or from the adversary, send (CLOCK-READ, sid_C) to $\mathcal{G}_{\text{CLOCK}}$ and upon receiving response (CLOCK-READ, $\mathsf{sid}_\mathsf{C}, \tau$) set $\tau_L := \tau$ and do the following:
  1. If I was received by an honest party $p_i \in \mathcal{P}$:
     (a) Set $\vec{\mathcal{I}}_H^T := \vec{\mathcal{I}}_H^T || (I, p_i, \tau_L)$;
     (b) Compute $\vec{\mathbf{B}} = (\vec{B}_1, \ldots, \vec{B}_\ell) := \texttt{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \texttt{NxtBC}, \mathsf{buffer}, \vec{\tau}_{\mathsf{state}})$ and if $\vec{B} \neq \varepsilon$ set $\mathsf{state} := \mathsf{state} || \mathsf{Blockify}(\vec{B}_1) || \ldots || \mathsf{Blockify}(\vec{B}_\ell)$ and $\vec{\tau}_{\mathsf{state}} := \vec{\tau}_{\mathsf{state}} || \tau_L^\ell$, where $\tau_L^\ell = \tau_L || \ldots, || \tau_L$.
     (c) For each $\texttt{BTX} \in \mathsf{buffer}$: if $\mathsf{Validate}(\texttt{BTX}, \mathsf{state}, \mathsf{buffer}) = 0$ then delete $\texttt{BTX}$ from buffer.

[13] For technical reasons which will be described in Section 4.1, Δ rounds in the protocol correspond to 2Δ clock-ticks.

(d) If there exists $j \in [\ell]$ with $p_{i_j} \in \mathcal{H} \setminus \mathcal{P}_{DS}$: $|\mathsf{state}| - \hat{\mathtt{pt}}_i \leq \mathtt{slack}$ or $\hat{\mathtt{pt}}_{i_j} \geq |\mathsf{state}_{i_j}|$, then for every $j \in [\ell]$ set $\mathtt{pt}_i := |\mathsf{state}| - |\vec{\mathbf{B}}|$ for every $p_i \in \mathcal{P}$.

2. Let $\hat{\mathcal{P}} \subseteq \mathcal{P}_{DS}$ denote the set of desynchronized honest parties that were registered at time $\tau' \leq \tau_L - \mathtt{Delay}$. Set $\mathcal{P}_{DS} := \mathcal{P}_{DS} \setminus \hat{\mathcal{P}}$.
3. Depending on the above input I and its sender's ID, $\mathcal{G}_{\text{LEDGER}}$ executes the corresponding code from the following list:

- *Submiting a transaction:*
If $I = (\text{SUBMIT}, \mathsf{sid}, \mathtt{x})$ and is received from a party $p_i \in \mathcal{P}$ or from $\mathcal{A}$ (on behalf of a corrupted party p_i) do the following:
  1. Choose a unique transaction ID $\mathtt{txid}$ and set $\mathtt{BTX} := (x, \mathtt{txid}, \tau_L, p_i)$
  2. If $\mathsf{Validate}(\mathtt{BTX}, \mathsf{state}, \mathsf{buffer}) = 1$, then $\mathsf{buffer} := \mathsf{buffer} \cup \{\mathtt{BTX}\}$
  3. send $(\text{SUBMIT}, \mathtt{BTX})$ to $\mathcal{A}$

- *Reading the state:*
If $I = (\text{READ}, \mathsf{sid})$ is received from a party $p_i \in \mathcal{P}$ then set $\mathsf{state}_i := \mathsf{state}|_{\min\{\mathtt{pt}_i, |\mathsf{state}|\}}$ and return $(\text{READ}, \mathsf{sid}, \mathsf{state}_i)$ to the requestor. If the requestor is $\mathcal{A}$ then send $(\mathsf{state}, \mathsf{buffer}, \vec{\mathcal{I}}_H^T)$ to $\mathcal{A}$.

- *Updating the state:*
If $I = (\text{MAINTAIN-LEDGER}, \mathsf{sid}, \mathsf{minerID})$ is received by an honest party $p_i \in \mathcal{P}$ and (after updating $\vec{\mathcal{I}}_H^T$ as above) $\mathtt{predict\text{-}time}(\vec{\mathcal{I}}_H^T) = \hat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \mathsf{sid}_\mathsf{C})$ to $\mathcal{G}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.

- *The adversary proposing the next block:*
If $I = (\text{NEXT-BLOCK}, (\mathtt{txid}_1, \ldots, \mathtt{txid}_\ell))$ is sent from the adversary, update $\mathtt{NxtBC}$ as follows:
  1. Set $\mathtt{NxtBC} := \varepsilon$
  2. For $i = 1, \ldots, \ell$ do: if there exists $\mathtt{BTX} := (x, \mathtt{txid}, \mathsf{minerID}, \tau_L, p_i) \in \mathsf{buffer}$ with ID $\mathtt{txid} = \mathtt{txid}_i$ then set $\mathtt{NxtBC} := \mathtt{NxtBC}||\mathtt{txid}_i$.
  3. Output $(\text{NEXT-BLOCK}, ok)$ to $\mathcal{A}$.

- *The adversary setting state-slackness:*
If $I = (\text{SET-SLACK}, (p_{i_1}, \hat{\mathtt{pt}}_{i_1}), \ldots, (p_{i_\ell}, \hat{\mathtt{pt}}_{i_\ell}))$, with $\{p_{i_1}, \ldots, p_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, and for all $j \in [\ell]$: $|\mathsf{state}| - \hat{\mathtt{pt}}_{i_j} \leq \mathtt{slack}$ and $\hat{\mathtt{pt}}_{i_j} \geq |\mathsf{state}_{i_j}|$, then for every $j \in [\ell]$ set $\mathtt{pt}_{i_1} := \hat{\mathtt{pt}}_{i_1}$, and return $(\text{SET-SLACK}, ok)$ to $\mathcal{A}$. Else set $\mathtt{pt}_{i_j} := |\mathsf{state}|$ for all $j \in [\ell]$.

- *The adversary setting the state for desychronized parties:*
If $I = (\text{DESYNC-STATE}, (p_{i_1}, \mathsf{state}'_{i_1}), \ldots, (p_{i_\ell}, \mathsf{state}'_{i_\ell}))$, with $\{p_{i_1}, \ldots, p_{i_\ell}\} \subseteq \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$, set $\mathsf{state}_{i_j} := \mathsf{state}'_{i_j}$ for each $j \in [\ell]$ and return $(\text{DESYNC-STATE}, ok)$ to $\mathcal{A}$.

4 Bitcoin as a Transaction Ledger Protocol

In this section we prove our main theorem, namely that, under appropriate assumptions, Bitcoin UC realized an instantiation of the ledger functionality from he previous section. More concretely, we cast the Bitcoin protocol as a UC protocol, where consistent to the existing methodology we assume that the protocol is synchronous, i.e., parties can keep track of the current round by using an appropriate global clock functionality. We start in Section 4.1 by giving a UC protocol, denoted as Ledger-Protocol, that abstracts the components of Bitcoin that are relevant for the construction of such a ledger—similar to how the backbone protocol [15] captures core Bitcoin properties in their respective model of computation.

Following that, in Section 4.2 we specify the ledger functionality $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$ that is implemented by the UC ledger protocol as an instance of our general ledger functionality, i.e., by providing appropriate instantiations of algorithms Validate, Blockify, and ExtendPolicy. In fact, for sake of generality, we specify generic classes of Validate and Blockify and parameterize our Ledger-Protocol with these classes, so that the security statement holds for any choice in these classes.

We then shift to proving our main realization theorem (Theorem 1) with, informally can be described as follows:

Theorem (Informal). Let $\mathsf{Validate}$ be the class of predicates that only take into account the current state and a transaction (i.e., no transaction IDs, time, or party IDs), and let $\texttt{slack} = \omega(\log \kappa)$ where κ is the size of the output of the random oracle. Then for an appropriate $\texttt{ExtendPolicy}$ and for any $\mathsf{Blockify}$ the protocol $\mathsf{Ledger\text{-}Protocol}$ instantiated with this $\mathsf{Validate}$ and $\mathsf{Blockify}$ algorithms statistically UC securely realizes the functionality $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$ with the above parameters under the following 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 formally be defined later):

- 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 in that round. 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. Informally, this modularization separates the protocol into a reactive *state-extend* subprocess, which, intuitively, captures the lottery that decides which miner gets to advance the blockchain next—and the process of propagating this state in the network—from the rest of the protocol. We prove in Lemma 3 that the state-extend module/subprocess implements an appropriate reactive UC functionality $\mathcal{F}_{\text{StX-ED}}$ and then use the UC composition theorem which allows us to argue security of $\mathsf{Ledger\text{-}Protocol}$ in a simpler hybrid world where instead using this subprocess parties make calls to the functionality $\mathcal{F}_{\text{StX-ED}}$. In this end of this section (Subsection 4.4) we show how both the GKL and PSs protocols can be cast as special cases of our protocol (for appropriate instantiations of its hybrids) and use this to compare our results with these works and formally confirm the differences in their respective assumptions.

4.1 The Bitcoin ledger as a UC Protocol

In the following we provide the formal description of protocol $\mathsf{Ledger\text{-}Protocol}$. $\mathsf{Ledger\text{-}Protocol}$ assumes as hybrids an asynchronous multi-cast network with eventual delivery $\mathcal{F}_{\text{A-MC}}$—in fact, we will assume that this network does have an upper bound Δ on the delay the adversary can incur but this is not known to the protocol or used in any way —and a random oracle functionality $\mathcal{F}_{\text{RO}}$. Before providing the detailed specification of our ledger protocol, we establish some useful notation and terminology that we will use throughout this section. For backwards compatibility, wherever it does not overload notation we use some of the terminology and notation from [15].

Blockchain A *blockchain* $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ is a (finite) sequence of blocks where each *block* $\mathbf{B}_i = \langle \mathsf{s}_i, \texttt{st}_i, \mathsf{n}_i \rangle$ is a triple consisting of the *pointer* s_i, the *block state* $\texttt{st}_i$, and the *nonce* n_i. A special block is the *genesis block* $\mathbf{G} = \langle \bot, \texttt{gen}, \bot \rangle$ which contains the genesis state $\texttt{gen}$. The *head* of chain $\mathcal{C}$ is the block $\mathsf{head}(\mathcal{C}) := \mathbf{B}_n$ and the *length* $\mathsf{length}(\mathcal{C})$ of the chain is the number of blocks, i.e., $\mathsf{length}(\mathcal{C}) = n$. The chain $\mathcal{C}^{\lceil k}$ is the (potentially empty) sequence of the first $\mathsf{length}(\mathcal{C}) - k$ blocks of $\mathcal{C}$. The *state* $\vec{\texttt{st}}$ corresponding to $\mathcal{C}$ is defined as a sequence of the corresponding block states, i.e., $\vec{\texttt{st}} := \texttt{st}_1 || \ldots || \texttt{st}_n$. In other words, one should think of the blockchain $\mathcal{C}$ as an encoding of its underlying state $\vec{\texttt{st}}$; 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 our protocol blockchains are used to store a sequence of messages, often referred to as transactions. Furthermore, as in [19] in order to capture blockchains with syntactically different states we use an algorithm $\mathsf{blockify}_{BC}$ to map a vector of transaction into a state block. Thus, each state block $\texttt{st} \in \vec{\texttt{st}}$ (except the genesis state) in the state corresponding in our blockchain has the form $\texttt{st} = \mathsf{Blockify}(\vec{N})$ where $\vec{N}$, is a vector of transactions.

For a blockchain $\mathcal{C}$ to be considered a valid blockchain, it needs to satisfy certain conditions. Concretely, the validity of a blockchain $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ where $\mathbf{B}_i = \langle \mathsf{s}_i, \texttt{st}_i, \mathsf{n}_i \rangle$ 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 is defined with respect to a difficulty parameter $\mathsf{D} \in [k]$, where k is the security parameter, and a given hash function $H(\cdot) : \{0,1\}^* \to \{0,1\}^\kappa$; it requires that for each $i > 1$ it must hold that for the value s_i contained in $\mathbf{B}_i$: $\mathsf{s}_i = H[\mathbf{B}_{i-1}]$ and $H[\mathbf{B}_i] < \mathsf{D}$.

The semantic validity on the other hand is defined on state-blocks in the $\vec{\mathtt{st}}$ corresponding to $\mathcal{C}$ and specifies whether the contents of the state are valid. This is similar to the validation predicate Validate defined in the ledger functionality (Section 3); in fact we will use this predicate to define the syntactic validity in a parametrizable format, i.e., so that for any choice of Validate, the blockchain protocol which uses this Validate for semantic validation implements the ledger parametrized with this Validate. We remark, however, that for the protocol to be able to use Validate, the predicate should ignore all information other than the state and transaction that is being validated.[14] To avoid confusion, throughout this section we use $\mathsf{ValidTx}_{BC}$ to refer to the validate predicate with the above restriction. The complete algorithm that checks state validity in our proposed UC abstraction of the Bitcoin ledger can be found in Appendix C.1. The algorithm ensures (among other things) that the chains start with the genesis block and that state blocks contain a special *coin-base* transaction $\mathtt{x}_{\mathsf{minerID}}^{\mathrm{coin-base}}$ which assigns them to a miner. Finally, denote by $\mathsf{isvalidchain}_D(\mathcal{C})$ the predicate which decides if chain $\mathcal{C}$ is valid.

The Ledger Protocol We are now ready to formally define our blockchain protocol Ledger-Protocol. The protocol allows an arbitrary number of parties/miners which communicate by means of multicasts over network $\mathcal{F}_{\text{A-MC}}$. (Note that this means that the adversary can send different messages to different parties.) New might dynamically joint/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.

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 $\mathsf{ValidTx}_{BC}$) and creates a new block state $\mathtt{st}$ using $\mathsf{blockify}_{BC}$. Next, the party attempts to mine a new block which can be validly added to their local blockchain. The mining is done using the $\mathsf{extendchain}_D$ algorithm which takes a chain $\mathcal{C}$, a state block $\mathtt{st}$ and the number of attempts q as inputs. The core idea of the algorithm is to find a proof-of-work which allows to extend the $\mathcal{C}$ by a block which encodes $\mathtt{st}$. The algorithm is given in Appendix C.2. After each mining attempt parties will multicast their current chain. A party will replace its local chain if it receives a longer chain. This ensures that all honest parties maintain the same ledger state.

As already mentioned our Bitcoin-Ledger protocol proceeds in rounds which are implemented by using a global synchronization clock $\mathcal{G}_{\text{CLOCK}}$. For formal reasons that have to do with how activations are handled in UC we have each round corresponds to the two rounds (to avoid confusion we refer to clock rounds as *clock-ticks*). I.e., we say that a protocol is in round r, if the current time of the clock is $\tau \in \{2r-1, 2r\}$. In fact, having such two clock-ticks is the way to ensure in synchronous UC that messages (e.g., a block) sent within a round is delivered at the beginning of the next round. The idea is that we will separate each round in two mini-rounds—where each mini-round corresponds to a clock tick—and treat the first mini-round as a *working mini-round* where parties might mine new blocks and submit them to the multicast network for delivery, and in the second *reading mini-round* they simply send a fetch to their network to see if some message was sent in the previews round. The detailed description of the UC blockchain protocol, denoted as Ledger-Protocol, can be found in Appendix C.3 where we also argue that the protocol satisfies Definition 1.

4.2 The Bitcoin Ledger

We next show how to instantiate the ledger functionality from Section 3 with appropriate parameters so that it is implemented by protocol Ledger-Protocol. To define this Bitcoin ledger $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$, we need to give specific instantiations of the three functions Validate, Blockify, and ExtendPolicy.

As mentioned above, for Validate, we use the same predicate as the one used by the protocol to validate states: For a given transaction $\mathtt{x}$, and a given state state decides whether this transaction is valid with respect

[14] Recall that in the general ledger description, Validate might depend on some associated metadata; although this might be useful to capture alternative blockchains, it is not the case in Bitcoin.

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 $\mathtt{x}$ and the state state, i.e., for any values of $\mathtt{txid}, \tau_L, p_i$, and buffer:

$$\mathsf{Validate}((\mathtt{x}, \mathtt{txid}, \tau_L, p_i), \mathsf{state}, \mathsf{buffer}) := \mathsf{ValidTx}_{BC}(\mathtt{x}, \mathsf{state}).$$

Blockify can be an arbitrary algorithm—and if the same algorithm is used in Ledger-Protocol the security proof will go through—although as discussed bellow (see Definition 2) a meaningful Blockify should be in certain relation with the ledger's Validate predicate. (This relation is satisfied by the Bitcoin protocol.)

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.) Moreover, the ExtendPolicy ensures the following property:

1. The speed of the Ledger is not too fast. This is implemented by defining a time window $\mathsf{window}^{\text{slack}}_{\text{maxGrow}}$ within which no more than a certain number of state blocks can be added.
2. The speed of the ledger is not too slow. This is implemented by defining a time window $\mathsf{window}^{\text{slack}}_{\text{minGrow}}$ within which at least a certain number of state blocks have to be added. This is known as minimal chain-growth.
3. The adversary cannot claim too many block for parties it is corrupting. This is formally enforced by requiring defining an upper bound η on the number of these so-called adversarial blocks within a sequence of state blocks. This is known as chain quality. Formally, this is enforced by requiring that a certain fraction of blocks need to start with a coinbase transaction that is associated with an actual honest and synchronized party.
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 conflicted. As we show Section 4.4, this guarantee can be amplified by using digital signatures.

In order to enforce these policies, ExtendPolicy first defines an alternative block, which satisfies all of the above criteria in 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 block. The formal description of Extend Policy for $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$ is given in Appendix C.4.

On the relation between Blockify *and* Validate*.* As discussed above, ExtendPolicy guarantees that the adversary cannot block the extension of the state indefinitely, and that occasionally, an honest miner will receive the block reward (via the coin-base) transaction. These correspond to the chain-growth and chain-quality properties from [15]. However, our generic ExtendPolicy makes explicit that a priori, we cannot exclude that the chain always extends with blocks that include a coin-base transaction only, i.e., any submitted transaction is ignored and never inserted into a new blocks. This issue is an orthogonal one to ensuring that honest transactions are not invalidated by adversarial interaction—which, as argued in [15] is obtained by adding digital signatures.

To see where this could problematic in general, consider a blockify that, at some point, creates a block that renders all possible future transactions invalid. Observe that this does not mean that our protocol is insecure and, and that this is as well possible for the protocols of [15, 29]; indeed our proof shows that the protocol will give exactly the same guarantees as an $\mathcal{G}_{\text{LEDGER}}$ parametrized with such an algorithm Blockify.

Nonetheless, a look in reality indicates that this situation never occurs with Bitcoin. To capture that this is the case Validate and Blockify need to be in certain relation with each other. Informally, this relation should ensure that the above standstill situation never occurs. A way to ensure this, which is already implemented by the Bitcoin protocol, is by restricting Blockify to only make an invertible manipulation of the blocks when they are inserted into the state—e.g., be an encoding function of a code—and define Validate to depend on the inverse of Blockify. This is captured in the following definition.

Definition 2. *A co-design of* Blockify *and* Validate *is* non-self-disqualifying *is there exists an efficient function* $\mathtt{Dec}$ *mapping outputs of* Blockify *to vectors such that and there exists a validate predicate* Validate′ *such that the following properties hold for any possible state* $\mathsf{state} = \mathtt{st}_1||\ldots||\mathtt{st}_\ell$*, buffer* buffer*, vectors* $\vec{N} := (\mathtt{x}_1, \ldots, \mathtt{x}_m)$ *and transaction* $\mathtt{x}$*:*

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

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 Blockify. Hence, in the following, we assume that $\mathsf{ValidTx}_{BC}$ and $\mathsf{blockify}_{BC}$ satisfy the relation in Definition 2.

4.3 Security Analysis

We next turn to the security analysis of our protocol. As already mentioned, we argue security in two step. In a first step, we extract from the protocol Ledger-Protocol a state-extend module/subprocess, denoted as StateExchange-Protocol, and devise an alternative, modular description of the Ledger-Protocol protocol in which every party makes invocations of this subprocess. We denote this modularized protocol by Modular-Ledger-Protocol. By a game-hopping argument, we prove that the original protocol Ledger-Protocol and the modularized protocol Modular-Ledger-Protocol are in fact functionally equivalent. The advantage of having such a modular description is that we are now able to define an appropriate ideal functionality $\mathcal{F}_{\text{StX-ED}}$ that is realized by Modular-Ledger-Protocol. Using the universal composition theorem we can deduce that Ledger-Protocol UC emulates Modular-Ledger-Protocol where invocations of StateExchange-Protocol are replaced by invocations of $\mathcal{F}_{\text{StX-ED}}$. The second step of the proof consists of proving that, under appropriate assumptions, Modular-Ledger-Protocol where invocations of StateExchange-Protocol are replaced by invocations of $\mathcal{F}_{\text{StX-ED}}$ implements the Bitcoin ledger described in Section 4.2. We next elaborate on the above two steps.

Step 1. The state-exchange functionality $\mathcal{F}_{\text{StX-ED}}$ 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 seeing as as lottery on (valid) states to be multicasted. Parties can use $\mathcal{F}_{\text{StX-ED}}$ to multicast a valid state, but instead of accepting any submitted state and sending it to all (registered) parties, $\mathcal{F}_{\text{StX-ED}}$ keeps track of all states that it ever saw, and implements the following mechanism upon submission of a new ledger state $\vec{\mathtt{st}}$ and a state block $\mathtt{st}$ from any party: If $\vec{\mathtt{st}}$ was previously submitted to $\mathcal{F}_{\text{StX-ED}}$ and $\vec{\mathtt{st}}||\mathtt{st}$ is a valid state, then $\mathcal{F}_{\text{StX-ED}}$ accepts $\vec{\mathtt{st}}||\mathtt{st}$ with probability p_H (resp. p_A for dishonest parties); accepted states are then sent to all registered parties. The formal specification follows:

Functionality $\mathcal{F}^{p_H,p_A}_{\text{StX-ED}}$

The functionality is parametrized with a set of parties $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$. For each party $p \in \mathcal{P}$ the functionality manages a tree $\mathcal{T}_p$ where each rooted path corresponds to a valid state the party has received. Initially each tree contains the genesis state. Finally, it manages a buffer $\vec{M}$ which contains successfully submitted states which have not yet been delivered to (some) parties in $\mathcal{P}$.

Submit/receive new states:

– Upon receiving ($\text{SUBMIT-NEW}, \mathsf{sid}, \vec{\mathtt{st}}, \mathtt{st}$) from some participant $p_s \in \mathcal{P}$, if $\mathsf{isvalidstate}(\vec{\mathtt{st}}||\mathtt{st}) = 1$ and $\vec{\mathtt{st}} \in \mathcal{T}_p$ do the following:
  1. Sample B according to a Bernoulli-Distribution with parameter p_H (or p_A if p_s is dishonest).
  2. If $B = 1$, set $\vec{\mathtt{st}}_{new} \leftarrow \vec{\mathtt{st}}||\mathtt{st}$ and add $\vec{\mathtt{st}}_{new}$ to $\mathcal{T}_{p_s}$. Else set $\vec{\mathtt{st}}_{new} \leftarrow \vec{\mathtt{st}}$.
  3. Output ($\text{SUCCESS}, \mathsf{sid}, B$) to p_s.

4. On response (CONTINUE, 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 $D_{\mathsf{mid}_1} := \ldots := D_{\mathsf{mid}_n} := 1$ set $\vec{M} := \vec{M}||(\vec{\mathtt{st}}_{new}, \mathsf{mid}_1, D_{\mathsf{mid}_1}, p_1)|| \ldots ||(\vec{\mathtt{st}}_{new}, \mathsf{mid}_n, D_{\mathsf{mid}_n}, p_n)$, and send (SUBMIT-NEW, $\mathsf{sid}, \vec{\mathtt{st}}_{new}, p_s, (p_1, \mathsf{mid}_1), \ldots, (p_n, \mathsf{mid}_n)$) to the adversary.

– Upon receiving (FETCH-NEW, sid) from a party $p \in \mathcal{P}$ or $\mathcal{A}$ (on behalf of p), do the following:
 1. For all tuples $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, p) \in \vec{M}$ set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.
 2. Let $\vec{M}_0^p$ denote the subvector of $\vec{M}$ including all tuples of the form $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, p)$ where $D_{\mathsf{mid}} = 0$ (in the same order as they appear in $\vec{M}$). For each tuple $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, p) \in \vec{M}_0^p$ add $\vec{\mathtt{st}}$ to $\mathcal{T}_p$. Delete all entries in $\vec{M}_0^p$ from $\vec{M}$ and send $\vec{M}_0^p$ to p.

– Upon receiving (SEND, $\mathsf{sid}, \vec{\mathtt{st}}, p'$) from $\mathcal{A}$ on behalf some *corrupted* $p \in \mathcal{P}$, if $p' \in \mathcal{P}$ and $\vec{\mathtt{st}} \in \mathcal{T}_p$, choose a new unique message-ID mid, initialize $D := 1$, add $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, p')$ to $\vec{M}$, and return (SEND, $\mathsf{sid}, \vec{\mathtt{st}}, p', \mathsf{mid}$) to $\mathcal{A}$.

Further adversarial influence on the network:

– Upon receiving (SWAP, $\mathsf{sid}, \mathsf{mid}, \mathsf{mid}'$) from $\mathcal{A}$, if mid and mid' are message-IDs registered in the current $\vec{M}$, swap the corresponding tuples in $\vec{M}$. Return (SWAP, sid) to $\mathcal{A}$.

– Upon receiving (DELAY, $\mathsf{sid}, T, \mathsf{mid}$) from $\mathcal{A}$, if T is a valid delay and mid is a message-ID for a tuple $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, p)$ in the current $\vec{M}$ set $D \leftarrow D + T$. Return (DELAY) to $\mathcal{A}$.

The Modular-Ledger-Protocol uses the same hybrids as our original protocol Ledger-Protocol but abstracts the lottery implemented by the mining process by making calls to the above state exchange functionality $\mathcal{F}_{\text{StX-ED}}^{p_H,p_A}$. The protocol is parametrized by T which is the number of blocks chopped off to get the ledger state. The detailed specification of the Modular-Ledger-Protocol protocol can be found in Appendix D.1. The following Lemma states that our Bitcoin protocol is a good implementation of the above modular ledger protocol and will be use in the final theorem. The proof can be found in Appendix D.2.

Lemma 1. *The protocol* $\mathsf{Ledger\text{-}Protocol}_{q,\mathsf{D},T}$ *UC emulates the protocol* $\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T$ *with* $\mathcal{F}_{\text{StX-ED}}^{p_H,p_A}$ *where* $p_A := \frac{\mathsf{D}}{2^\kappa}$ *and* $p_H = 1 - (1 - p_A)^q$.

Step 2. We are now ready to complete the proof of our main theorem. Before providing the formal statement it is useful to discuss some of the key properties use both in the statement and the proof. The security of the Bitcoin-Protocol depends on various key properties of an execution. This means that its security depends on the number of random oracle queries (or, in the $\mathcal{F}_{\text{StX-ED}}$ hybrid world, the number of submit-queries) by the pool of corrupted miners. Therefore it is important to capture the relevant properties of such a UC execution. In the following we denote by upper-case R the number of rounds of a given protocol execution.

Capturing query power in an execution. In an execution, we measure the query power per logical round r, which can be conveniently captured as a function $\mathsf{T_{qp}}(r)$. We observe that in an execution of t_{rc} rounds, the total number of queries is

$$Q_{t_{rc}}^{r'} = \sum_{r=r'}^{r'+t_{rc}-1} \mathsf{T_{qp}}(r).$$

In each round $r \in [R]$, each honest miner gets a certain number $q_i^{(r)}$ of activations from the environment to maintain the ledger (i.e., to query a state extend). Let

$$q_H^{(r)} := \sum_{p_i \text{ honest in round r}} q_i^{(r)}.$$

Also, the adversary makes a certain number $q_A^{(r)}$ of queries to $\mathcal{F}_{\text{StX-ED}}$. We get

$$\mathsf{T_{qp}}(r) = q_A^{(r)} + q_H^{(r)}.$$

Quantifying total mining power in an execution. Mining power is the expected number of successful state extensions, i.e., the number of times a new state block is successfully mined. The mining power of round r is therefore

$$\mathsf{T_{mp}}(r) := q_A^{(r)} \cdot p_A + q_H^{(r)} \cdot p_H.$$

Recall that p_H is the success probability per query of an honest miner and p_A is the success probability per query of a corrupted miner. If $p_A = p$ and $p_H = 1-(1-p)^q$, it is convenient to consider $(q_A^{(r)} + q \cdot q_H^{(r)}) \cdot p$ as the total mining power (by applying Bernoulli's inequality). Within an interval of t_{rc} rounds, we can quantify the expectation by

$$\mathsf{T}_{\mathsf{mp}}^{\text{total}}(t_{rc}) := \sum_{r=1}^{t_{rc}} \mathsf{T}_{\mathsf{mp}}(r).$$

Ideally, the re-calibration of Bitcoin keeps the value $\mathsf{T}_{\mathsf{mp}}^{\text{total}}(t_{rc})$ constant, for t_{rc} corresponding to the desired time bound for 2016 new state blocks.

Quantifying adversarial mining power in an execution. The *adversarial mining power* $\mathsf{mp}_A(r)$ per round is made up of two parts: first, queries by corrupted parties, and second, queries by honest, but de-synchronized miners (recall that a party is considered de-synchronized for 2Δ rounds after its registration).

$$\mathsf{mp}_A(r) := p_A \cdot q_A^{(r)} + p_H \cdot \sum_{p_i \text{ is de-sync}} q_i^{(r)}.$$

It is convenient to measure the adversary's contribution to the mining power as the fraction of the overall mining power. In particular, we assume there is a parameter $\rho \in (0,1)$ such that in any round r, the relation $\mathsf{mp}_A(r) \leq \rho \cdot \mathsf{T}_{\mathsf{mp}}(r)$ holds. We then define $\beta_r := \rho \cdot \mathsf{T}_{\mathsf{mp}}(r)$. Looking ahead, if a model is flat, then the fraction $(1-\rho)$ corresponds to the fraction of users that are honest and synchronized.

Quantifying honest and synchronized mining power in an execution. In each round $r \in [R]$, each honest miner gets a certain number $q_{i,r}$ of activations from the environment, where it can submit one new state to $\mathcal{F}_{\text{STX-ED}}$. This state is accepted with probability p_H. We define the vector $\vec{q}_r$ such that for any honest miner p_i in round r, $\vec{q}_r[i] = q_{i,r}$. The probability that a miner p_i is successful to extend the state by at least one block is $\alpha_{i,r} := 1-(1-p_H)^{q_{i,r}}$. and the probability that at least one *registered* and uncorrupted miner successfully queries $\mathcal{F}_{\text{STX-ED}}$ to extend its local longest state is

$$\alpha_r := 1 - \prod_{\text{honest sync } p_i} (1-\alpha_{i,r}) = 1 - \prod_{\text{honest sync } p_i} (1-p_H)^{q_{i,r}}.$$

Looking ahead, in existing models, parties are expected to be synchronized and are otherwise counted as dishonest, and due to the flatness, $(1-\rho)$ is the fraction of honest and synchronized miners.

Worst-Case Analysis. We analyze Bitcoin in a worst-case fashion. Let us assume that the protocol runs for $[R]$ rounds (e.g., $R = t_{rc}$ if we do not take re-calibration into account), then

$$\alpha := \min\{\alpha_r\}_{r\in[R]}, \text{ and } \beta := \max\{\beta_r\}_{r\in[R]}.$$

Remark 2. This view on Bitcoin gives already a glimpse for the relevance of the re-calibration sub-protocol which is considered as part of future work. Ideally, we would like the variation among the values α_r and among the values β_r to be small, which needs an additional assumption on the increase of computing power per round. Thanks to the re-calibration phase, such a bound can exist at all. If no re-calibration phase would happen, any strictly positive gradient of the computing power development would eventually provoke Bitcoin failing, since, mathematically, the expected value β could not be reasonably bounded.

We are now ready to state the main theorem. The proof of the theorem can be found in Appendix D.3.

Theorem 1. *Let the functions* $\mathsf{ValidTx}_{BC}$*,* $\mathsf{blockify}_{BC}$*, and* ***ExtendPolicy*** *be as defined above. Let* $p \in (0,1)$*, integer* $q \geq 1$ *and let* $p_H = 1-(1-p)^q$ *and* $p_A = p$*. Let* $\Delta \geq 1$ *be the upper bound on the network delay. Consider* $\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T$ *in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{F}_{\text{STX-ED}}^{p_H,p_A}, \mathcal{F}_{\text{A-MC}})$*-hybrid world. If, for some* $\lambda > 1$*, the relation*

$$\alpha \cdot (1 - 2 \cdot (\Delta+1) \cdot \alpha) \geq \lambda \cdot \beta \quad (1)$$

is satisfied in any real-world execution, where α and β are defined as above, then the protocol $\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T$ *UC-realizes* $\mathcal{G}^{B}_{\text{LEDGER}}$ *for any parameters in the range*

$$\mathtt{slack} = T \quad \text{and} \quad \mathtt{Delay} = 4\Delta,$$

$$\mathsf{window}^{\text{slack}}_{\text{minGrow}} \geq \frac{\mathtt{slack}}{(1-\delta)\cdot\gamma} \quad \text{and} \quad \mathsf{window}^{\text{slack}}_{\text{maxGrow}} \leq \frac{\mathtt{slack}}{(1+\delta)\cdot \max_r \mathsf{T}_{\mathsf{mp}}(r)},$$

$$\eta > (1+\delta)\cdot \mathtt{slack} \cdot \frac{\beta}{\gamma},$$

where $\gamma := \frac{\alpha}{1+\Delta\alpha}$ and $\delta > 0$ is an arbitrary constant. In particular, the realization is perfect except with probability $R \cdot \mathsf{negl}(T)$, where R denotes the upper bound on the number of rounds, and $\mathsf{negl}(T)$ denotes a negligible function in T.

Remark 3. It is worth noting the implications of Equation 1. In practice, typically p is small such that α (and thus γ) can be approximated using Bernoulli's inequality to be $(1-\rho)mp$, where m is the estimated number of hash queries in the Bitcoin network per round. Hence, by cancelling out the term mp and letting p be sufficiently small (compared to $\frac{1}{\Delta m}$), Equation 1 collapses roughly to the condition that $(1-\rho)(1-\epsilon) \geq (1+\delta)\rho$, which basically relates the fractions of adversarial vs. honest mining power. Also, as pointed out by [29], for too large values of p in the order of $p > \frac{1}{mp}$, Equation 1 is violated for any constant fraction ρ of corrupted miners and they present an attack in this case.

Proof (Overview). To show the theorem we specify a simulator for the ideal world that internally runs the round-based mining procedure of every honest party. Whenever the real world parties complete a working round, then the simulator has to assemble the views of its simulated honest (and synchronized) miners and determine their common prefix of states, i.e., the longest state stored or received by each simulated such 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. To reflect that not all parties have the same view on this common prefix, the simulator can adjust the state pointers accordingly. The simulation inside the simulator is perfect and corresponds to an 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 do not forbid a proper simulation, we have to justify why the choice of the parameters in the theorem statement is acceptable. To this end, we analyze the real-world execution to bound the corresponding bad events that prevent a perfect simulation. This can be done following the detailed analysis provided by Pass, Seeman, and shelat [29] which includes the necessary claims for lower and upper on chain growth, chain quality, and prefix consistency. From these Claims, it follows that our simulator can simulate the real-world, since the restrictions imposed by the ledger prohibit a prefect simulation only with probability $R \cdot \mathsf{negl}(T)$. This is an upper bound on the distinguishing advantage of the real and ideal world. The detailed proof is found in Appendix D.3 □

Note that Theorem statement a-priori holds for any environment (but simply yields a void statement if the conditions are not met). In order to get useful composable statements, we follow the approach proposed in Section 2 and model restrictions as wrapper functionalities to ensure the condition of the theorem. We review two particular choices in 4.4. The general conceptual principle behind this is the following: define a wrapper functionality $\mathcal{W}$ for the hybrid world, consisting of a network $\mathcal{F}_{\text{A-MC}}$, a clock $\mathcal{G}_{\text{CLOCK}}$ and a random oracle with output length κ (the security parameter), which ensures the condition in Equation 1. This can be done by enforcing appropriate restrictions along the lines of the basic example in Section 2 (e.g., imposing an upper bound on parties, or RO queries per round etc.). For such a wrapper $\mathcal{W}$, we have the following desired corollary to Theorem 1 and Lemma 1. This statement is guaranteed to compose according to the UC composition theorem.

Corollary 1. *Protocol* $\mathsf{Ledger\text{-}Protocol}$*, in the $(\mathcal{G}_{\text{CLOCK}}, \mathcal{F}_{\text{A-MC}}, \mathcal{W}(\mathcal{F}_{\text{RO}}))$-hybrid world, UC realizes functionality $\mathcal{G}^{B}_{\text{LEDGER}}$.*

4.4 Comparison with Existing Work

We demonstrate how the protocols, assumptions, and results from the two existing works analyzing security of Bitcion (in a property based manner) can be cast as special cases of our construction. We start with the result in [15], which is the so-called flat and synchronous model[15] with instant delivery and a constant number of parties n (i.e., Bitcoin is seen as an n-party MPC protocol). [16] Consider the concrete values for α and β as follows:

- Let n denote the number of parties. Each corrupted party gets at most q activations to query the $\mathcal{F}_{\text{StX-ED}}$ per round. Each honest party is activated exactly once per round.
- In the model of GKL, we have $q \geq 1$. Thus, we get $p_H = 1-(1-p)^q$ and $p_A = p$. Thus, $\mathsf{T}_{\mathsf{mp}}^{\text{GKL}}(r) \leq p \cdot q \cdot n$.
- The adversary gets (at most) q queries per corrupted party with probability $p_A = p$ and one query per honest but de-synchronzied party with success probability $p_H = 1-(1-p)^q$. If t_r denotes the number of corrupted or de-synchronized parties in round r, we get $\mathsf{mp}_A^{\text{GKL}}(r) \leq t \cdot q \cdot p$ and thus $\beta_r^{\text{GKL}} = p \cdot q \cdot (\rho \cdot n)$, where ρn is the number of miners contributing to the adversarial mining power (which is independent of r.)
- Each honest and synchronized miner gets exactly one activation per round, i.e., $q_{i,r} := 1$, with $p_H = 1-(1-p)^q \in (0,1)$, for some integer $q > 0$. Inserting it into the general equation yields $\alpha_r^{\text{GKL}} = 1-(1-p)^{q(1-\rho)\cdot n}$ (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.

We can define the wrapper $\mathcal{W}$ that enforces this behavior, namely by the wrapper that simply ignores parties and queries that exceed the allowed budget. In the hybrid world $(\mathcal{G}_{\text{CLOCK}}, \mathcal{W}(\mathcal{F}_{\text{StX-ED}}^{p_H,p_A}), \mathcal{F}_{\text{A-MC}})$ (for a sufficiently small probability $p(\kappa)$ in the security parameter κ), this ensures the condition of Theorem 1 and we arrive at the following composable statement for this model:

Corollary 2. *The protocol* Modular-Ledger-Protocol *UC-realizes* $\mathcal{G}_{\text{LEDGER}}$ *in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{W}(\mathcal{F}_{\text{StX-ED}}), \mathcal{F}_{\text{A-MC}})$-*hybrid model (with network delay $\Delta = 1$) for the parameters assured by Theorem 1 for the specific values of*

$$\alpha^{\text{GKL}} = 1-(1-p)^{(1-\rho)\cdot q\cdot n} \quad \textit{and} \quad \beta^{\text{GKL}} = p \cdot q \cdot (\rho \cdot n).$$

Similarly, we can instantiate the above values with the assumptions of [29]:

- For each corrupted (and de-synchronized) party, the adversary gets at most one query per round. Each honest miner makes exactly one query per round. This means that $q_A^{(r)} + q_H^{(r)} = n_r$.
- In the PSs model, $p_H = p_A = p$ and hence $\mathsf{T}_{\mathsf{mp}}^{\text{PSs}}(r) \leq p \cdot n_r = p \cdot n$, where n is as above. With these values we get $\mathsf{mp}_A^{\text{PSs}}(r) = p \cdot n_r^{\text{corr}}$ and consequently $\beta_r^{\text{PSs}} = p \cdot (\rho \cdot n)$, where ρn denotes the upper bound on corrupted parties in any round. Putting things together, we also have $\alpha_r^{\text{PSs}} = 1-(1-p)^{(1-\rho)\cdot n}$. Note that since n is assumed to be fixed in their model, $(1-\rho)\cdot n$ is in fact a lower bound on the honest and synchronized hashing power.

For the same wrapper functionality $\mathcal{W}$ as above (where the restriction is 1 query per corrupted instead of q), we again see that $(\mathcal{G}_{\text{CLOCK}}, \mathcal{W}(\mathcal{F}_{\text{StX-ED}}^{p,p}), \mathcal{F}_{\text{A-MC}})$ (for a sufficiently small probability $p(\kappa)$ in the security parameter κ), will ensure the condition of the theorem and directly yields the following composable statement.

Corollary 3. *The protocol* Modular-Ledger-Protocol *UC-realizes* $\mathcal{G}_{\text{LEDGER}}$ *in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{W}(\mathcal{F}_{\text{StX-ED}}^{p,p}), \mathcal{F}_{\text{A-MC}})$-*hybrid model (with network delay $\Delta \geq 1$) for the parameters assured by Theorem 1 for the specific values of*

$$\alpha^{\text{PSs}} = 1-(1-p)^{(1-\rho)\cdot n} \quad \textit{and} \quad \beta^{\text{PSs}} = p \cdot (\rho \cdot n).$$

[15] The flat model means that every party gets the same number of hash queries in every round.

[16] In a recent paper, the authors of [15] propose an analysis of Bitcoin for a variable number of parties. Capturing the appropriate assumptions for this case, as a wrapper in our composable setting, is part of future work.

5 Implementing a Stronger Ledger

As already observed in [15], the Bitcoin protocol makes use of digital signatures to protect transactions which allows it to achieve a stronger guarantee that, informally mandates that every transaction submitted by an honest miner will eventually make it in the state. Using our terminology, this means that using digital signature Bitcoin implement a stronger ledger. 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 world where this protocol is replaced by $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$.

Protection of transactions using accounts. In Bitcoin, a miner creates an account ID AccountID by generating a signature key pair and hashing the public key. Any transaction of this party includes this account ID, i.e., $\mathtt{x} = (\mathsf{AccountID}, \mathtt{x}')$. An important property is that a transaction of a certain account cannot be invalidated by a transaction with a different account ID. Hence, to protect the validity of a transaction, upon submitting $\mathtt{x}$, party p_i has to sign it, append the signature and verification key to get a transaction $((\mathsf{AccountID}, \mathtt{x}'), vk, \sigma)$. The validation predicate now additionally has to check that the account ID 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 $\mathtt{x}$, only by either forging a signature relative to vk, or by possessing key pair whose hash of the public key collides with the account ID of the honest party. The details of the protocol and the validate predicate ledger are in Appendix E.

Realized ledger. The ledger abstraction realized is formally specified in Appendix E and denoted $\mathcal{G}^{\text{B}+}_{\text{LEDGER}}$. Roughly, it is a ledger functionality as the one from the previous section, but which additionally allows parties to create unique accounts. Upon receiving a transaction from party p_i, $\mathcal{G}^{\text{B}+}_{\text{LEDGER}}$ only accepts a transaction containing the AccountID that was previously associated to p_i and ensures that parties are restricted to issue transactions using their own account.

Amplification of transaction liveness. In Bitcoin a given transaction can only be invalidated due to another one with the same account. By definition of the enhanced ledger, this means that no other party can make a transaction of p_i not enter the state. The liveness guarantee for transactions specified by `ExtendPolicy` in the previous chapter implies captures that if a valid transaction is in the buffer for long enough then it eventually enters the state. For $\mathcal{G}^{\text{B}+}_{\text{LEDGER}}$, this implies that if p_i submits a single transaction which is valid according to the current state, then this transaction will eventually be contained in the state. More precisely, we can conclude that this happens within the next $2 \cdot \mathtt{slack}$ new state blocks in the worst case. Relative to the current view of p_i this is no more than within the next $3 \cdot \mathtt{slack}$ blocks as argued in Appendix E.

References

[1] Marcin Andrychowicz and Stefan Dziembowski. PoW-based distributed cryptography with no trusted setup. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 379–399. Springer, Heidelberg, August 2015.

[2] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Fair two-party computations via bitcoin deposits. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *FC 2014 Workshops*, volume 8438 of *LNCS*, pages 105–121. Springer, Heidelberg, March 2014.

[3] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 443–458. IEEE Computer Society Press, May 2014.

[4] Moshe Babaioff, Shahar Dobzinski, Sigal Oren, and Aviv Zohar. On bitcoin and red balloons. *SIGecom Exchanges*, 10(3):5–9, 2011.

[5] Michael Backes, Dennis Hofheinz, Jörn Müller-Quade, and Dominique Unruh. On fairness in simulatability-based cryptographic systems. FMSE ’05, 2005.

[6] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[7] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 421–439. Springer, Heidelberg, August 2014.

[8] Vitalik Buterin. A next-generation smart contract and decentralized application platform, 2013. `https://github.com/ethereum/wiki/wiki/White-Paper`.

[9] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[10] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[11] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

[12] Sandro Coretti, Juan A. Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In *ASIACRYPT*, 2016.

[13] Ittay Eyal. The miner's dilemma. In *2015 IEEE Symposium on Security and Privacy*, pages 89–103. IEEE Computer Society Press, May 2015.

[14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 436–454. Springer, Heidelberg, March 2014.

[15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 281–310. Springer, Heidelberg, April 2015.

[16] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol with chains of variable difficulty. Cryptology ePrint Archive, Report 2016/1048, 2016.

[17] Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 477–498. Springer, Heidelberg, March 2013.

[18] Aggelos Kiayias, Elias Koutsoupias, Maria Kyropoulou, and Yiannis Tselekounis. Blockchain mining games. In *EC*, 2016.

[19] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. Fair and robust multi-party computation using a global transaction ledger. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 705–734. Springer, Heidelberg, May 2016.

[20] Ranjit Kumaresan and Iddo Bentov. How to use bitcoin to incentivize correct computations. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 30–41. ACM Press, November 2014.

[21] Ranjit Kumaresan and Iddo Bentov. Amortizing secure computation with penalties. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 418–429. ACM Press, October 2016.

[22] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use bitcoin to play decentralized poker. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 195–206. ACM Press, October 2015.

[23] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. Improvements to secure computation with penalties. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 406–417. ACM Press, October 2016.

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

[25] Leslie Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, 2002.

[26] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[27] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. `http://bitcoin.org/bitcoin.pdf`.

[29] Rafael Pass, Lior Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. Cryptology ePrint Archive, Report 2016/454, 2016. `http://eprint.iacr.org/2016/454`.

[30] Rafael Pass and Elaine Shi. FruitChains: A fair blockchain. Cryptology ePrint Archive, Report 2016/916, 2016. `http://eprint.iacr.org/2016/916`.

[31] Michael O. Rabin. Randomized byzantine generals. In *FOCS*, 1983.

[32] Mike Rosulek. Must you know the code of f to securely compute f? In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 87–104. Springer, Heidelberg, August 2012.

[33] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 507–527. Springer, Heidelberg, January 2015.

[34] Aviv Zohar. Bitcoin: under the hood. *Commun. ACM*, 58(9):104–113, 2015.

A Related Literature

At a high level, the Bitcoin 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 them 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. If we assume that the hash function in this experiment is fully unpredictable (i.e., before computing it one has probability $2^{-\kappa}$ of predicting its output, where κ is the length of the hash), then the best strategy for finding such metadata is 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.

Intuitively, a successful solution can be seen as a *proof of work* that testifies to the fact that the miner presenting is 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. This creates for each miner a sequence of seen miner blocks. Moreover, a block is rendered valid by any miner only if it includes a hash-pointer the last valid block that thus miner has seen. Thus, the sequence of valid blocks forms a linked list which is often referred to as the *blockchain*.

The works of Garay, Kiayas, and Leonardos [15] and that of Pass, Seeman, and shelat [29] include a formal specification and security proof of the Bitcoin protocol.[17] However, the proved security in these works is property-based, i.e., informally, they prove that conditioned on the largest part of the network following the Bitcoin protocol (in fact and abstraction and generalization thereof), the output of bitcoin satisfies certain properties.

These properties are as follows:

- The *common prefix property* from [15] is a property of the linked list of blocks commonly referred to as the *blockchain* that is created by executing Bitcoin. It requires that if a block $\mathbf{B}$ is “deep enough” in a blockchain $\mathcal{C}$ that is considered valid by some honest party, then the prefix-subchain $\mathcal{C}|_{\mathbf{B}}$ of $\mathcal{C}$ that results by ignoring in $\mathcal{C}$ all blocks that are after $\mathbf{B}$ will eventually become a prefix-subchain of all honest parties. In [29] this property was refined (and augmented) by requiring a similar *consistency* property which in addition to the above mandates that for every honest party, cutting his chain at a deep enough block, yields a prefix-chain that will be prefix for ever. These properties are satisfied with overwhelming probability (in a security parameter κ), where deep enough is defined as $O(\kappa)$.
- The *chain growth*, which was implicitly defined in [15] and posted as a required property in [29], mandates, informally, that the blockchain of honest parties will increase with time. More concretely, it postulates a lower bound on the speed in which blocks are added in the blockchain. In fact, [29] introduced also the notion of a corresponding upper bound which, as the argue, might be useful for synchronization purposes.We observer that although [29] adopts a round-based model of execution (see below) which makes it unclear where or how such extra synchronization would help, the proposed property would indeed be useful to have if one was to remove that assumed synchronous structure from the model.[18]
- The *chain quality* [15, 29] property postulates that the honest miners get to insert a fraction of the blocks that eventually make it into the common prefix of honest parties. This, means that the adversary is not able to monopolize the blocks that are inserted in the blockchain, and every so often an honest miner is allowed to do so. This abstracts the property which was already postulated in the original blockchain whitepaper that the Bitcoin implements a lottery which decides (according to the computing power that each party contributes—alas not necessarily in a fair allocation manner [30]) who will insert that next block to the chain.

[17] An extension of [15] to allow capturing dynamically evolving miner-set was also recently posted on the IACR eprint [16].

[18] Such an adaptation would however harm several of the arguments and would require a new analysis and protocol.

Both [15] and [29] assume a multicast network—i.e., a network where a party sends messages to arbitrary other parties[19]—and abstract the hash as a random oracle. Furthermore, they both have an explicit round-based model of execution where parties proceed in rounds. The main differences between the two are: (1) that in [15] every party make q of hash-queries (i.e., q hash calls) in each round as opposed to [29] where every party exactly one hash-query per round; and (2) in [15] message sent in some round r are guaranteed to be delivered at the beginning of round $r+1$, whereas in [29] the adversary might choose to delay message delivery but the statements are proved assuming no message is delayed by more than Δ rounds. We note that Property 1 implies that both models assume that parties are fully synchronized, since in both models every party can simply count the number of queries it has made to the random oracle and decide in what round it is (and by assumption this decision will ensure that no party goes to round $r+1$ before all parties have finished round r). Notwithstanding, as we argue in Section 4 the network assumption in [29] in combination with the design of the protocol yields *strictly* weaker setting than the one in [15], i.e., one can achieve strictly more with the assumptions in [15] even assuming an external synchronization mechanism. We will discuss these difference further when we describe these network assumptions and in particular in Section 4 where we argue how these two models are captured in UC and how the corresponding protocols can be cast as special case of our UC protocol.

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 [2, 3, 7, 23, 21, 22, 20, 1]—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 one of the composable simulation-based frameworks, e.g., [10, 9, 11]. In such a framework, security of a protocol is defined as follows: First we specify the goal the protocol is supposed to achieve by means of a trusted third party, usually referred to as the *ideal functionality* $\mathcal{F}$. Then we prove that the protocol implements this functionality $\mathcal{F}$ which means that for any adversary $\mathcal{A}$ attacking the protocol, there exists an ideal adversary (aka a *simulator*) $\mathcal{S}$ that attacks an ideal invocation of $\mathcal{F}$ and emulates the attach that $\mathcal{A}$ launches to the protocol. We assume that the reader has some familiarity with simulation-based security, and in particular with the UC and GUC frameworks of Canetti et al. [10, 11].

The advantage of simulation-based security is that it often comes with a composition theorem which, intuitively, states that we can replace calls to a functionality with invocation of a protocol implementing it without worrying about the protocol's internals. Thus, in our case, a ledger functionality that is implemented by Bitcoin allows us to use this functionality in any protocol that wants to use Bitcoin as a resource, and the composition theorem will then imply that replacing the functionality with Bitcoin does not compromise security. Note that [19] already includes an attempt to define such a functionality, but as we argue here, the proposed functionality is too strong to be implemented from Bitcoin under standard assumptions.

B Model (Cont'd)

This section includes complementary material for Section 2.

B.1 The Asynchronous Channel

Functionality $\mathcal{F}_{\text{A-CH}}$

The functionality is parametrized with a receiver p_R, and and upper bound Δ on the delay of any channel. I keeps track of the set of possible senders $\mathcal{P}$. Any newly registered (resp. deregistered) party is added to (resp. deleted from) $\mathcal{P}$.

[19] Unlike [15] 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.

– Upon receiving a message ($\texttt{send}, m$) 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 $D^{MAX}_{\mathsf{mid}} = 1$, set $\vec{M} := \vec{M}||(m, \mathsf{mid}, D_{\mathsf{mid}})$, and send $(m, \mathsf{mid}, D_{\mathsf{mid}})$ to the adversary.
– Upon receiving a message ($\texttt{fetch}$) from p_r:
  1. For all regired 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 a message ($\texttt{delay}, T_{\mathsf{mid}}, \mathsf{mid}$) from the adversary, if $D^{MAX}_{\mathsf{mid}} + T_{\mathsf{mid}} \leq \Delta$ and mid is a message-ID registered in the current $\vec{M}$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} + T_{\mathsf{mid}}$ and $D^{MAX}_{\mathsf{mid}} := D^{MAX}_{\mathsf{mid}} + T_{\mathsf{mid}}$; otherwise, ignore the message.
– Upon receiving a message ($\texttt{swap}, \mathsf{mid}, \mathsf{mid}'$) 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 ($\texttt{swap-ok}$) to the adversary.

B.2 The Network

The asynchronous multicast network with bounded delivery is described in the following.

Functionality $\mathcal{F}_{\text{A-MC}}$

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

– *Honest sender multicast:*
  Upon receiving a message $(\text{MULTICAST}, \mathsf{sid}, m)$ from some $p_s \in \mathcal{P}$, where $\mathcal{P} = \{p_1, \ldots, p_n\}$ denotes the current party set, choose n new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$, initialize $2n$ new variables $D_{\mathsf{mid}_1} := D^{MAX}_{\mathsf{mid}_1} \ldots := D_{\mathsf{mid}_n} := D^{MAX}_{\mathsf{mid}_n} := 1$, set $\vec{M} := \vec{M}||(m, \mathsf{mid}_1, D_{\mathsf{mid}_1}, p_1)|| \ldots ||(m, \mathsf{mid}_n, D_{\mathsf{mid}_n}, p_n)$, and send $(\text{MULTICAST}, \mathsf{sid}, m, p_s, (p_1, \mathsf{mid}_1), \ldots, (p_n, \mathsf{mid}_n))$ to the adversary.
– *Adversarial sender (partial) multicast:*
  Upon receiving a message $(\text{MULTICAST}, \mathsf{sid}, (m_{i_1}, p_{i_1}), \ldots, (m_{i_\ell}, p_{i_\ell})$ from the adversary with $\{p_{i_1}, \ldots, p_{i_\ell}\} \subseteq \mathcal{P}$, choose ℓ new unique message-IDs $\mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_\ell}$, initialize ℓ new variables $D_{\mathsf{mid}_{i_1}} := D^{MAX}_{\mathsf{mid}_{i_1}} := \ldots := D_{\mathsf{mid}_{i_\ell}} := D^{MAX}_{\mathsf{mid}_{i_\ell}} := 1$, set $\vec{M} := \vec{M}||(m_{i_1}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}, p_{i_1})|| \ldots ||(m_{i_\ell}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_\ell}}, p_\ell)$, and send $((\text{MULTICAST}, \mathsf{sid}, (m_{i_1}, p_{i_1}, \mathsf{mid}_{i_1}), \ldots, (m_{i_\ell}, p_{i_\ell}, \mathsf{mid}_{i_\ell})$ to the adversary.
– *Honest party fetching:*
  Upon receiving a message $(\text{FETCH}, \mathsf{sid})$ from $p_i \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of p_i if p_i is 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 $\vec{M}^{p_i}_0$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \mathsf{mid}, D_{\mathsf{mid}}, p_i)$ with $D_{\mathsf{mid}} = 0$ (in the same order as they appear in $\vec{M}$). Delete all entries in $\vec{M}^{p_i}_0$ from $\vec{M}$, and send $\vec{M}^{p_i}_0$ to p_i.
– *Adding adversarial delays:*
  Upon receiving a message $(\text{DELAYS}, \mathsf{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell}))$ do the following for each pair $(T_{\mathsf{mid}_{i_j}}, \mathsf{mid}_{i_j})$ in this message:
  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.
– *Adversarially reordering messages:*
  Upon receiving a message $(\text{SWAP}, \mathsf{sid}, \mathsf{mid}, \mathsf{mid}')$ 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 $(\text{SWAP}, \mathsf{sid})$ to the adversary.

Implementing the Multicast network The asynchronous multicast network can be implemented by means of a synchronized message-diffusion protocol over a network of unicast channels. The parties in such a diffusion mechanism forward in each round all new messages they received in the previous round to all the unicast channels they are connected to as senders. (In order to ensure that parties can send the messages twice, a nonce is attached to each input message that is to be multicasted.[20] Let $G = (V, E)$ denote the (dynamically updatable) directed graph whose vertices V are the registered parties that are currently participating to this diffusion protocol 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 (who is given any message submitted to any unicast channel) needs to simply simulate when parties see the 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.

B.3 The Clock

The clock functionality. We describe the clock below. The global clock functionality, i.e., a shared clock that may interact with more than one protocol session, is denoted by $\bar{\mathcal{G}}_{\text{CLOCK}}$. The standard UC functionality is denoted by $\mathcal{G}_{\text{CLOCK}}$ (without the bar).

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

The functionality is available to all participants. The functionality is parametrized with variable τ, a set of parties $\mathcal{P}'$, and a set F of functionalities. For each party $p \in \mathcal{P}'$ it manages variable d_p. For each $\mathcal{F} \in F$ it manages variable $d_{\mathcal{F}}$

Initially, $\tau := 0$, $\mathcal{P}' := \emptyset$ and $F := \emptyset$.

Synchronization:

- Upon receiving $(\text{CLOCK-UPDATE}, \mathsf{sid}_\mathsf{C})$ from some party $p \in \mathcal{P}'$ set $d_p := 1$; execute *Round-Update* and forward $(\text{CLOCK-UPDATE}, \mathsf{sid}_\mathsf{C}, p)$ to $\mathcal{A}$.
- Upon receiving $(\text{CLOCK-UPDATE}, \mathsf{sid}_\mathsf{C})$ from some functionality $\mathcal{F} \in F$ set $d_{\mathcal{F}} := 1$, execute *Round-Update* and return $(\text{CLOCK-UPDATE}, \mathsf{sid}_\mathsf{C}, \mathcal{F})$ to $\mathcal{F}$.
- Upon receiving $(\text{CLOCK-READ}, \mathsf{sid}_\mathsf{C})$ from any participant (including the environment, the adversary, or any ideal—shared or local—functionality) return $(\text{CLOCK-READ}, \mathsf{sid}_\mathsf{C}, \tau)$ to the requestor.

Procedure Round-Update:
If $d_{\mathcal{F}} := 1$ for all $\mathcal{F} \in F$ and $d_p = 1$ for all honest p in $\mathcal{P}'$, then set $\tau := \tau + 1$ and reset $d_{\mathcal{F}} := 0$ and $d_p := 0$ for all parties in $\mathcal{P}'$.

B.4 The Random Oracle Functionality

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

The functionality is parametrized by a security parameter κ. It maintains a set of registered parties/miners $\mathcal{P}$ (initially set to $\emptyset$) and a (dynamically updatable) function table $\mathcal{T}$ (initially $\mathcal{T} = \emptyset$). For simplicity we write $T[x] = \perp$ to denote the fact that no pair of the form $(x, \dot{})$ is in $\mathcal{T}$.

- Upon receiving $(\text{EVAL}, \mathsf{sid}, x)$ from some party $p \in \mathcal{P}$ (or from $\mathcal{A}$ on behalf of a corrupted p), do the following:

[20] The relayers do not add another nonce to the message they relay.

1. If $H[x] = \bot$ sample a value y uniformly at random from $\{0,1\}^\kappa$, set $H[x] \leftarrow y$ and add $(x, T[x])$ to $\mathcal{T}$.
2. Return $(\textsc{eval}, \mathsf{sid}, x, H[x])$ to the requestor.

B.5 The Wrapped RO

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

The wrapper functionality is parametrized by an upper bound q which restricts the $\mathcal{F}$-evaluations of each corrupted party per round. (To keep track of rounds the functionality registers with the global clock $\mathcal{G}_{\text{CLOCK}}$.) The functionality manages the variable $\mathsf{counter}$ and the current set of corrupted miners $\mathcal{P}'$. For each party $p \in \mathcal{P}'$ it manages variables count_p.

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

General:

- The wrapper stops the interaction with the adversary as soon as the adversary tries to exceed its budget of q queries per corrupted party.

Relaying inputs to the random oracle:

- Upon receiving $(\textsc{eval}, \mathsf{sid}, x)$ from $\mathcal{A}$ on behalf of a corrupted party $p \in \mathcal{P}'$, then first execute *Round Reset*. Then, set $\mathsf{count}_p \leftarrow \mathsf{count}_p + 1$ and only if $\mathsf{count}_p \leq q$ forward the request to $\mathcal{F}_{\text{RO}}$ and return to $\mathcal{A}$ whatever $\mathcal{F}_{\text{RO}}$ returns.
- Any other request from any participant or the adversary is simply relayed to the corresponding hybrid functionality without any further action and the output is given to the destination specified by the hybrid functionality.

Standard UC Corruption Handling:

- Upon receiving $(\textsc{corrupt}, \mathsf{sid}, p)$ 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 $\mathsf{count}_p \leftarrow t$. Otherwise set $\mathsf{count}_p \leftarrow 0$.

Procedure Round-Reset:
Send $(\textsc{clock-read}, \mathsf{sid}_\mathsf{C})$ to $\bar{\mathcal{G}}_{\text{CLOCK}}$ and receive $(\textsc{clock-read}, \mathsf{sid}_\mathsf{C}, \tau)$ from $\bar{\mathcal{G}}_{\text{CLOCK}}$. If $|\tau - \mathsf{counter}| > 0$ and the new time τ is even (i.e., a new round started), then set $\mathsf{count}_p := 0$ for each participant $p \in \mathcal{P}'$ and set $\mathsf{counter} \leftarrow \tau$.

C Bitcoin Protocol (Cont'd)

This section includes complementary material for Section 4.1.

C.1 Algorithm isvalidstate

The algorithm $\mathsf{isvalidstate}$ takes a state $\vec{\mathtt{st}}$ as input and checks whether $\vec{\mathtt{st}}$ is valid with respect to $\mathsf{ValidTx}_{BC}$.

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

Let $\vec{\mathtt{st}} := \mathtt{st}_1 || \ldots || \mathtt{st}_n$
for each $\mathtt{st}_i$ **do**
  Extract the transaction sequence $\vec{\mathtt{x}}_i \leftarrow \mathtt{x}_{i,1}, \ldots, \mathtt{x}_{i,n_i}$ contained in $\mathtt{st}_i$
end for
$\vec{\mathtt{st}}' \leftarrow \mathtt{gen}$ // Initialize the genesis state
for $i = 1$ to n **do**
  if the first transaction in $\vec{\mathtt{x}}_i$ is not a coin-base transaction **return** false

$\vec{N}_i \leftarrow \mathtt{x}_{i,1}$
for $j = 2$ to $|\vec{\mathtt{x}}_i|$ **do**
  $\mathtt{st} \leftarrow \mathsf{blockify}_{BC}(\vec{N}_i)$
  if $\mathsf{ValidTx}_{BC}(\mathtt{x}_{i,j}, \vec{\mathtt{st}}'||\mathtt{st}) = 0$ **return** false
  $\vec{N}_i \leftarrow \vec{N}_i||\mathtt{x}_{i,j}$
end for
$\vec{\mathtt{st}}' \leftarrow \vec{\mathtt{st}}'||\mathtt{st}_i$
end for
return true

C.2 Algorithm extendchain

The algorithm 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 $\mathsf{extendchain}_D(\mathcal{C}, \mathtt{st}, q)$

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

C.3 Ledger-Protocol

The Bitcoin ledger protocol is described in the following. It assumes as hybrids a random oracle $\mathcal{F}_{\text{RO}}$, a network $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ for blockchains, a network $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$ for transcations, and clock $\mathcal{G}_{\text{CLOCK}}$. Note that the two networks are instances of $\mathcal{F}_{\text{A-MC}}$ and can be realized from a single network $\mathcal{F}_{\text{A-MC}}$ using different message-IDs. The protocol is parametrized by q, D, T where q is the number of mining attempts per round, D is the difficulty of the proof-of-work, and T is the number of blocks chopped off to get the ledger state. The registration process in the protocol works as follows. If a party receives (REGISTER, sid) from the environment it registers at the random oracle, the network, and the clock. Upon the next activation the party multicasts NEW-PARTY over the network. In the mining procedure parties check if they have received a NEW-PARTY message in the current round. If this is the case they multicast their transaction buffer in addition to their local chain. For the sake of simplicity of the presentation we omit the explicit registration process from the protocol description below.

Protocol $\mathsf{Ledger\text{-}Protocol}_{q,\mathsf{D},T}(p)$

Initialization:
We assume that the party p is registered to $\mathcal{F}_{\text{RO}}$, $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$, $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$, and $\mathcal{G}_{\text{CLOCK}}$.
The protocol manages the local state $\vec{\mathtt{st}}$ encoded in the chain $\mathcal{C}_{loc}$ which initially contains genesis block, i.e., $\mathcal{C}_{loc} \leftarrow (\mathbf{G})$.
The protocol also manages an initially empty buffer buffer which contains the received transactions. The buffer forms the information base to build new chain blocks.

$t \leftarrow 0$.

Clock:

- Upon receiving (CLOCK-READ, $\mathsf{sid_C}$) forward the query to $\mathcal{G}_{\text{CLOCK}}$ and return whatever is received as answer from $\mathcal{G}_{\text{CLOCK}}$
- Upon receiving (CLOCK-UPDATE, $\mathsf{sid_C}$), do the following: remember that a clock-update was received in the current mini-round.

Ledger:

- Upon receiving (SUBMIT, $\mathsf{sid}, \mathtt{x}$), set $\mathsf{buffer} \leftarrow \mathsf{buffer}||\mathtt{x}$, and send (MULTICAST, $\mathsf{sid}, \mathtt{x}$) to $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$.
- Upon receiving (READ, sid) send (CLOCK-READ, $\mathsf{sid_C}$) to $\mathcal{G}_{\text{CLOCK}}$, receive as answer (CLOCK-READ, $\mathsf{sid_C}, \tau$) and proceed as follows.

    // Fetch new states in update mini-rounds
    if τ corresponds to an update mini-round and $t < \tau$ **then**
        Execute **FetchInformation** and set $t \leftarrow \tau$.
    end if
    // Return current ledger state
    Return (READ, $\mathsf{sid}, \vec{\mathtt{st}}^{\lceil T}$).
- Upon receiving (MAINTAIN-LEDGER, $\mathsf{sid}, \mathsf{minerID}$) execute **MiningProcedure**

MiningProcedure:

Step 1: If a (CLOCK-UPDATE, $\mathsf{sid_C}$) has been received during this update mini-round then send (CLOCK-UPDATE, $\mathsf{sid_C}$) to $\mathcal{G}_{\text{CLOCK}}$ (if it hasn't been sent already in the current mini-round), and in the next activation go to the next step. Else in the next activation repeat this step.

Step 2: Send (CLOCK-READ, $\mathsf{sid_C}$) to $\mathcal{G}_{\text{CLOCK}}$, receive as answer (CLOCK-READ, $\mathsf{sid_C}, \tau$), and proceed as follows.

    if τ corresponds to a working mini-round **then**
        // Generate a new block: extract transactions and form a state-block
        Set $\mathsf{buffer}' \leftarrow \mathsf{buffer}$
        Parse buffer' as sequence $(\mathtt{x}_1, \ldots, \mathtt{x}_n)$
        Set $\vec{N} \leftarrow \mathtt{x}^{\text{coin-base}}_{\mathsf{minerID}}$
        Set $\mathtt{st} \leftarrow \mathsf{blockify}_{BC}(\vec{N})$
        repeat
            for $i = 1$ to n **do**
                if $\mathsf{ValidTx}_{BC}(\mathtt{x}_i, \vec{\mathtt{st}}||\mathtt{st}) = 1$ **then**
                    $\vec{N} \leftarrow \vec{N}||\mathtt{x}_i$
                    Remove $\mathtt{x}$ from buffer'
                    Set $\mathtt{st} \leftarrow \mathsf{blockify}_{BC}(\vec{N})$
                end if
            end for
        until $\vec{N}$ does not increase anymore
        Execute **ExtendState($\mathtt{st}$)** and go to step 3 in the next activation.
    else
        Go to the beginning of step 2 in the next activation.
    end if

Step 3:
If a (CLOCK-UPDATE, $\mathsf{sid_C}{}_C$) has been received during this working round then send (CLOCK-UPDATE, $\mathsf{sid_C}$) to $\mathcal{G}_{\text{CLOCK}}$, and in the next activation go to the next step. Else in the next activation repeat this step.

Step 4:
Send (CLOCK-READ, $\mathsf{sid_C}$) to $\mathcal{G}_{\text{CLOCK}}$, receive as answer (CLOCK-READ, $\mathsf{sid_C}, \tau$), and proceed as follows.

    if τ corresponds to an update mini-round **then**
        If $t < \tau$ execute **FetchInformation** and set $t \leftarrow \tau$.
        Go to step 1 in the next activation.

else

Go to the beginning of step 4 in the next activation.

end if

ExtendState($\mathtt{st}$):

$\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\mathsf{D}}(\mathcal{C}_{loc}, \mathtt{st}, q)$

if $\mathcal{C}_{\text{new}} \neq \mathcal{C}_{loc}$ **then**

Update the local chain, i.e., $\mathcal{C}_{loc} \leftarrow \mathcal{C}_{\text{new}}$.

end if

// Broadcast current chain

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

FetchInformation:

// Fetch new chains and update the local state

Send $(\text{FETCH}, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$; denote the response from $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ by $(\text{FETCH}, \mathsf{sid}, b)$.

Extract valid chains $\mathcal{C}_1, \ldots, \mathcal{C}_k$ from b.

Set $\mathcal{C}_{loc}$ to the longest valid chain in $\mathcal{C}_{loc}, \mathcal{C}_1, \ldots, \mathcal{C}_k$ (to resolve ties the ordering decides).

// Fetch new transactions and add them to the buffer

Send $(\text{FETCH}, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$; denote the response from $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$ by $(\text{FETCH}, \mathsf{sid}, b)$.

Extract received transactions $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ from b.

Set $\mathsf{buffer} \leftarrow \mathsf{buffer}||(\mathtt{x}_1, \ldots, \mathtt{x}_k)$.

Remove all transactions from buffer which are invalid with respect to $\vec{\mathtt{st}}^{\lceil T}$

We now show that the ledger protocol has a predictable synchronization pattern according to Definition 1.

Lemma 2. *The protocol* $\mathsf{Ledger\text{-}Protocol}_{q,\mathsf{D},T}$ *satisfies Definition 1.*

Proof (Sketch). Recall that we have to argue that there exists an algorithm $\mathtt{predict\text{-}time}_{\Pi}(\cdot)$ such that for any possible execution of Π (i.e., for any adversary and environment, and any choice of random coins) it holds the following: If $\vec{\mathcal{I}}_H^T = ((x_1, \mathtt{pid}_1, \tau_1), \ldots, (x_m, \mathtt{pid}_m, \tau_m))$ is the corresponding timed honest inputs sequence for this execution, then for any $i \in [m-1]$: $\mathtt{predict\text{-}time}_{\Pi}((x_1, \mathtt{pid}_1, \tau_1), \ldots, (x_i, \mathtt{pid}_i, \tau_i)) = \tau_{i+1}$. This is straightforward to see for our ledger protocol (and all protocols that share the same structure). Roughly, the predicate $\mathtt{predict\text{-}time}$ can be implemented as follows: browse through the entire sequence $\vec{\mathcal{I}}_H^T$ and determine how many times the clock advances. The clock advances for the first time, when all miners got a maintain command, followed by a clock-update command. By definition of Ledger-Protocol, in this each party will then send a clock-update to the clock. If every party has done that, 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 honest parties will provoke a clock update. Note that only honest parties can make the clock advance. □

C.4 The Bitcoin Ledger (Cont'd)

This section includes complementary material for Section 4.2.

In the following we 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 Validate and Blockify.

Algorithm Extend Policy for $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$

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

Let τ_L be current ledger time (computed from $\vec{\mathcal{I}}_H^T$)

if $|\mathsf{state}| \geq \mathtt{slack}$ **then**

Set $\tau_{\mathtt{slack}} \leftarrow \vec{\tau}_{\mathsf{state}}[|\mathsf{state}| - \mathtt{slack} + 1]$

else

Set $\tau_{\texttt{slack}} \leftarrow 0$
end if// $\tau_{\texttt{slack}}$ is the time of the state block which is `slack` behind the head of the current state
// Now, create an honest client block as alternative
Create a local copy of buffer' (such that the original buffer does not change)
Set $\vec{N} \leftarrow \mathtt{x}^{\text{coin-base}}_{\mathsf{minerID}}$ of an honest miner
Sort buffer' according to time stamps.
Let $\vec{\mathtt{x}} = (\mathtt{x}_1, \ldots, \mathtt{x}_n)$ be the transactions in buffer'
Set $\mathtt{st} \leftarrow \mathsf{blockify}_{BC}(\vec{N})$
repeat
  for $i = 1$ to n **do**
    if $\mathsf{ValidTx}_{BC}(\mathtt{x}_i, \mathsf{state}||\mathtt{st}) = 1$ **then**
      $\vec{N} \leftarrow \vec{N}||\mathtt{x}_i$
      Remove $\mathtt{x}_i$ from $\vec{\mathtt{x}}$
      Set $\mathtt{st} \leftarrow \mathsf{blockify}_{BC}(\vec{N})$
    end if
  end for
until $\vec{N}$ does not increase anymore
// Possibly more than one block should be added
Parse `NxtBC` as a vector $\mathtt{NxtBC} = (\mathtt{NxtBC}_1, \cdots, \mathtt{NxtBC}_n)$
$\vec{B} \leftarrow \varepsilon$ // Initialize Result
for each list $\mathtt{NxtBC}_i$ of transaction IDs **do**
  $\vec{B}_i \leftarrow \varepsilon$
  // Verify validity of $\mathtt{NxtBC}_i$ and compute content
  Use the `txid` contained in $\mathtt{NxtBC}_i$ to determine the list of transactions
  Let $\vec{\mathtt{x}} = (\mathtt{x}_1, \ldots, \mathtt{x}_{|\mathtt{NxtBC}_i|})$ denote the transactions of $\mathtt{NxtBC}_i$
  if $\mathtt{x}_1$ is not a coin-base transaction **then**
    return $\vec{N}$
  else
    $\vec{B}_i \leftarrow \mathtt{x}_1$
    for $j = 2$ to $|\mathtt{NxtBC}_i|$ **do**
      Set $\mathtt{st}_i \leftarrow \mathsf{blockify}_{BC}(\vec{B}_i)$
      if $\mathsf{ValidTx}_{BC}(\mathtt{x}_j, \mathsf{state}||\mathtt{st}_i) = 0$ **then**
        return $\vec{N}$
      end if
      $\vec{B}_i \leftarrow \vec{B}_i||\mathtt{x}_j$
    end for
    Set $\mathtt{st}_i \leftarrow \mathsf{blockify}_{BC}(\vec{B}_i)$
  end if
  // Test that all old valid transaction are included
  if the coin-base transaction of $\vec{B}_i$ denotes an honest party **then**
    for each $\mathtt{BTX} = (\mathtt{x}, \mathtt{txid}, \tau', p_i) \in \mathsf{buffer}$ with time $\tau' \leq \tau_L - \frac{\mathtt{Delay}}{2}$ **do**
      if $\mathsf{ValidTx}_{BC}(\mathtt{x}, \mathsf{state}'||\mathtt{st}_i) = 1$ but $\mathtt{x} \notin \vec{B}_i$ **then**
        return $\vec{N}$
      end if
    end for
  end if
  // Must not proceed too fast, too slow, or with too many adversarial blocks.
  if $\tau_L - \tau_{\texttt{slack}} < \mathsf{window}^{\text{slack}}_{\text{maxGrow}}$ **then**
    return ε
  else if $\tau_L - \tau_{\texttt{slack}} > \mathsf{window}^{\text{slack}}_{\text{minGrow}}$ **and** `NxtBC` is empty **then**
    return $\vec{N}$
  else if `NxtBC` is non-empty **and** corrupted party set as coinbase **then**
    Determine the most recent honest block $\mathtt{st}_i$ in state using $\vec{\tau}_{\texttt{state}}$.
    if $|\mathsf{state}| - i \geq \eta$ **then**

return $\vec{N}$
end if
end if
$\vec{B} \leftarrow \vec{B}||\vec{B}_i$
$\mathsf{state}' \leftarrow \mathsf{state}'||\mathtt{st}_i$
end for
return $\vec{B}$
end function

D Security Analysis (Cont'd)

This section includes complementary material for Section 4.3.

D.1 Modular-Ledger-Protocol

The Modular-Ledger-Protocol uses the same hybrids as our original protocol Ledger-Protocol but abstracts the lottery implemented by the mining process by making calls to the above state exchange functionality $\mathcal{F}^{p_H,p_A}_{\text{StX-ed}}$. The protocol is parameterized by T which is the number of blocks chopped off to get the ledger state. The registration process work as in the Ledger-Protocol.

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

Initialization:

We assume that the party p is registered to $\mathcal{F}^{p_H,p_A}_{\text{StX-ed}}$, $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$, and $\mathcal{G}_{\text{clock}}$.
The protocol manages the local ledger state $\vec{\mathtt{st}}$ which initially is the genesis state, i.e. , $\vec{\mathtt{st}} \leftarrow (\mathtt{gen})$.
The protocol also manages an initially empty buffer buffer which contains the received transactions. The buffer forms the information base to build new state blocks.
$t \leftarrow 0$.

Clock:

- Upon receiving $(\text{clock-read}, \mathsf{sid}_\mathsf{C})$ forward the query to $\mathcal{G}_{\text{clock}}$ and return whatever is received as answer from $\mathcal{G}_{\text{clock}}$
- Upon receiving $(\text{clock-update}, \mathsf{sid}_\mathsf{C})$, do the following: remember that a clock-update was received in the current mini-round.

Ledger:

- Upon receiving $(\text{submit}, \mathsf{sid}, \mathtt{x})$, set $\mathsf{buffer} \leftarrow \mathsf{buffer}||\mathtt{x}$, and send $(\text{multicast}, \mathsf{sid}, \mathtt{x})$ to $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$.
- Upon receiving $(\text{read}, \mathsf{sid})$ send $(\text{clock-read}, \mathsf{sid}_\mathsf{C})$ to $\mathcal{G}_{\text{clock}}$, receive as answer $(\text{clock-read}, \mathsf{sid}_\mathsf{C}, \tau)$ and proceed as follows.
  // Fetch new states in update mini-rounds
  if τ corresponds to an update mini-round and $t < \tau$ **then**
  Execute **FetchInformation** and set $t \leftarrow \tau$.
  end if
  // Return current ledger state
  Return $(\text{read}, \mathsf{sid}, \vec{\mathtt{st}}^{\lceil T})$
- Upon receiving $(\text{maintain-ledger}, \mathsf{sid}, \mathsf{minerID})$ execute **MiningProcedure**

MiningProcedure:

Step 1:
If a $(\text{clock-update}, \mathsf{sid}_\mathsf{C})$ has been received during this update mini-round then send $(\text{clock-update}, \mathsf{sid}_\mathsf{C})$ to $\mathcal{G}_{\text{clock}}$ (if it hasn't been sent already in the current mini-round), and in the next activation go to the next step. Else in the next activation repeat this step.

Step 2:

Send ($\textsc{clock-read}, \mathsf{sid_C}$) to $\mathcal{G}_{\textsc{clock}}$, receive as answer ($\textsc{clock-read}, \mathsf{sid_C}, \tau$), and proceed as follows.

if τ corresponds to a working mini-round **then**
// Generate a new block: extract transactions and form a state-block
Set $\mathsf{buffer}' \leftarrow \mathsf{buffer}$
Parse buffer' as sequence $(\mathtt{x}_1, \ldots, \mathtt{x}_n)$
Set $\vec{N} \leftarrow \mathtt{x}^{\text{coin-base}}_{\mathsf{minerID}}$
Set $\mathtt{st} \leftarrow \mathsf{blockify}_{BC}(\vec{N})$
repeat
for $i = 1$ to n **do**
if $\mathsf{ValidTx}_{BC}(\mathtt{x}, \vec{\mathtt{st}}||\mathtt{st}) = 1$ **then**
$\vec{N} \leftarrow \vec{N}||\mathtt{x}$
Remove $\mathtt{x}$ from buffer'
Set $\mathtt{st} \leftarrow \mathsf{blockify}_{BC}(\vec{N})$
end if
end for
until $\vec{N}$ does not increase anymore
Execute **ExtendState(st)** and go to step 3 in the next activation.
else
Go to the beginning of step 2 in the next activation.
end if

Step 3:

If a ($\textsc{clock-update}, \mathsf{sid}_{CC}$) has been received during this working round then send ($\textsc{clock-update}, \mathsf{sid_C}$) to $\mathcal{G}_{\textsc{clock}}$, and in the next activation go to the next step. Else in the next activation repeat this step.

Step 4:

Send ($\textsc{clock-read}, \mathsf{sid_C}$) to $\mathcal{G}_{\textsc{clock}}$, receive as answer ($\textsc{clock-read}, \mathsf{sid_C}, \tau$), and proceed as follows.

if τ corresponds to an update mini-round **then**
if $t < \tau$ **then**
Execute **FetchInformation** and set $t \leftarrow \tau$.
end if
Go to step 1 in the next activation.
else
Go to the beginning of step 4 in the next activation.
end if

ExtendState(st):

Send ($\textsc{submit-new}, \mathsf{sid}, \vec{\mathtt{st}}, \mathtt{st}$) to $\mathcal{F}_{\textsc{StX-ed}}$.
Denote the response by ($\textsc{success}, \mathsf{sid}, B$) of $\mathcal{F}_{\textsc{StX-ed}}$.
if $B = 1$ **then**
Update the local state, i.e., $\vec{\mathtt{st}} \leftarrow \vec{\mathtt{st}}||\mathtt{st}$.
end if
// Broadcast current state using $\mathcal{F}_{\textsc{StX-ed}}$.
Send ($\textsc{continue}, \mathsf{sid}$) to $\mathcal{F}_{\textsc{StX-ed}}$

FetchInformation:

// Fetch new states and update the local state
Send ($\textsc{fetch-new}, \mathsf{sid}$) to $\mathcal{F}_{\textsc{StX-ed}}$.
Denote the response from $\mathcal{F}_{\textsc{StX-ed}}$ by ($\textsc{fetch-new}, \mathsf{sid}, (\vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k)$).
Set $\vec{\mathtt{st}}$ to the longest state in $\vec{\mathtt{st}}, \vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k$ (to resolve ties the ordering decides).
// Fetch new transactions and add them to the buffer
Send ($\textsc{fetch}, \mathsf{sid}$) to $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$; denote the response from $\mathcal{F}^{\mathsf{tx}}_{\text{A-MC}}$ by ($\textsc{fetch}, \mathsf{sid}, b$).
Extract received transactions $(\mathtt{x}_1, \ldots, \mathtt{x}_k)$ from b.
Set $\mathsf{buffer} \leftarrow \mathsf{buffer}||(\mathtt{x}_1, \ldots, \mathtt{x}_k)$.
Remove all transactions from buffer which are invalid with respect to $\vec{\mathtt{st}}^{\lceil T}$

D.2 Proof of Lemma 1

In the following we provide a proof for Lemma 1.

Lemma (1). *The protocol* $\mathsf{Ledger\text{-}Protocol}_{q,\mathsf{D},T}$ *UC emulates the protocol* $\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T$ *with* $\mathcal{F}^{p_H, p_A}_{\text{StX-ED}}$ *where* $p_A := \frac{\mathsf{D}}{2^\kappa}$ *and* $p_H = 1 - (1 - p_A)^q$.

In a first step, we describe protocol $\mathsf{StateExchange\text{-}Protocol}$ and show that it UC-realizes the $\mathcal{F}_{\text{StX-ED}}$ functionality in the $\mathcal{F}_{\text{RO}}$, $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ hybrid world. Note that $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ is a (named) instance of the $\mathcal{F}_{\text{A-MC}}$ functionality. The protocol is parametrized by q and D where q is the number of mining attempts per submission attempt and D is the difficulty of the proof-of-work.

Protocol $\mathsf{StateExchange\text{-}Protocol}_{q,\mathsf{D}}(p)$

Initialization:

We assume that the party p is registered to $\mathcal{F}_{\text{RO}}$ and $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$.
The protocol maintains a tree $\mathcal{T}$ of all valid chains. Initially it contains the genesis chain $(\mathbf{G})$.

Message Exchange:

- Upon receiving $(\textsc{submit-new}, \mathsf{sid}, \vec{\mathtt{st}}, \mathtt{st})$ do
  // Check if there exists a chain in $\mathcal{T}$ which contains the state $\vec{\mathtt{st}}$
  if $\mathsf{isvalidstate}(\vec{\mathtt{st}}||\mathtt{st}) = 1$ **then**
    if there exists $\mathcal{C} \in \mathcal{T}$ with $\vec{\mathtt{st}}$ **then**
      // Try to extend the chain
      $\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\mathsf{D}}(\mathcal{C}, \mathtt{st}, q)$
      if $\mathcal{C}_{\text{new}} \neq \mathcal{C}$ **then**
        Update the local tree, i.e., add $\mathcal{C}_{\text{new}}$ to $\mathcal{T}$
        Output $(\textsc{success}, \mathsf{sid}, 1)$ to p.
      else
        Output $(\textsc{success}, \mathsf{sid}, 0)$ to p.
      end if
      // Broadcast current chain
      On response $(\textsc{continue}, \mathsf{sid})$ send $(\textsc{multicast}, \mathsf{sid}, \mathcal{C}_{\text{new}})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$.
    end if
  end if
- Upon receiving $(\textsc{fetch-new}, \mathsf{sid})$ if do the following:
  Send $(\textsc{fetch}, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ and denote the response by $(\textsc{fetch}, \mathsf{sid}, b)$.
  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 3. *Let* $p := \frac{\mathsf{D}}{2^\kappa}$. *The protocol* $\mathsf{StateExchange\text{-}Protocol}_{q,\mathsf{D}}$ *UC-realizes functionality* $\mathcal{F}^{p_H, p_A}_{\text{StX-ED}}$ *in the* $(\mathcal{F}_{\text{RO}}, \mathcal{F}_{\text{A-MC}})$*-hybrid model where* $p_A := p$ *and* $p_H = 1 - (1 - p_A)^q$.

Proof. We consider the following simulator.

Simulator $\mathsf{sim}_{\text{stx}}$

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`.
The simulator manages a set $\mathcal{P}_{RO}$ of parties registered to the random oracle and a set of parties $\mathcal{P}_{net}$ registered to the network.

Random Oracle:

– Upon receiving (EVAL, sid, v) for $\mathcal{F}_{\text{RO}}$ from $\mathcal{A}$ on behalf of corrupted $p \in \mathcal{P}_{RO}$ do the following.
  1. If $H[v]$ is already defined, output (EVAL, $\mathsf{sid}, v, H[v]$).
  2. If v is of the form $(\mathsf{s}, \mathtt{st}, \mathsf{n})$ and there exists[a] a chain $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ such that $H[\mathbf{B}_n] = \mathsf{s}$ proceed as follows. If $\mathcal{C} \notin \mathcal{T}$ abort with `tree-error`. Otherwise continue. Extract the state $\vec{\mathtt{st}}$ from $\mathcal{C}$ and extract the state block $\mathtt{st}$ from v. Send (SUBMIT-NEW, $\mathsf{sid}, \vec{\mathtt{st}}, \mathtt{st}$) to $\mathcal{F}_{\text{StX-ED}}$ and denote by (SUCCESS, B) the output of $\mathcal{F}_{\text{StX-ED}}$. If $B = 1$ set $H[v]$ to a uniform random value in $\{0,1\}^\kappa$ strictly smaller[b] 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, $\mathsf{sid}, v, H[v]$).
  3. Otherwise set v to a uniform random value in $\{0,1\}^\kappa$ and output (EVAL, $\mathsf{sid}, v, H[v]$).

Network:

– Upon receiving (MULTICAST, $\mathsf{sid}, (m_{i_1}, p_{i_1}), \ldots, (m_{i_\ell}, p_{i_\ell})$) for $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ from $\mathcal{A}$ on behalf of corrupted $p \in \mathcal{P}_{net}$ with $\{p_{i_1}, \ldots, p_{i_\ell}\} \subseteq \mathcal{P}_{net}$ proceed as follows.
  1. Choose ℓ new unique message-IDs $\mathsf{mid}_{i_1}, \ldots, \mathsf{mid}_{i_\ell}$, initialize ℓ new variables $D_{\mathsf{mid}_{i_1}} := \ldots := D_{\mathsf{mid}_{i_\ell}} := 1$, set $\vec{M} := \vec{M}||(m_{i_1}, \mathsf{mid}_{i_1}, D_{\mathsf{mid}_{i_1}}, p_{i_1})|| \ldots ||(m_{i_\ell}, \mathsf{mid}_{i_\ell}, D_{\mathsf{mid}_{i_\ell}}, p_\ell)$.
  2. For each (m_{i_j}, p_{i_j}) where m_{i_j} is a chain $\mathcal{T}$ extract the state $\vec{\mathtt{st}}_{i_j}$ from m_{i_j}, and send (SEND, $\mathsf{sid}, \vec{\mathtt{st}}, p_{i_j}$) to $\mathcal{F}_{\text{StX-ED}}$. Store the message-ID $\widehat{\mathsf{mid}_{i_j}}$ returned by $\mathcal{F}_{\text{StX-ED}}$ with mid_{i_j}.
  3. Output (MULTICAST, $\mathsf{sid}, (m_{i_1}, p_{i_1}, \mathsf{mid}_{i_1}), \ldots, (m_{i_\ell}, p_{i_\ell}, \mathsf{mid}_{i_\ell})$) to $\mathcal{A}$.
– Upon receiving (FETCH, sid) for $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ from $\mathcal{A}$ on behalf of corrupted $p \in \mathcal{P}_{net}$ proceed as follows.
  1. For all tuples $(m, \mathsf{mid}, D_{\mathsf{mid}}, p) \in \vec{M}$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} - 1$.
  2. Let $\vec{M}_0^p$ denote the subvector $\vec{M}$ including all tuples of the form $(m, \mathsf{mid}, D_{\mathsf{mid}}, p)$ with $D_{\mathsf{mid}} = 0$ (in the same order as they appear in $\vec{M}$). 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, $\mathsf{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell})$) 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 $\mathcal{F}_{\text{StX-ED}}$-message-ID $\widehat{\mathsf{mid}}$ for mid send (DELAY, $\mathsf{sid}, T_{\mathsf{mid}}, \widehat{\mathsf{mid}}$) to $\mathcal{F}_{\text{StX-ED}}$.
– Upon receiving a message (SWAP, $\mathsf{sid}, \mathsf{mid}_1, \mathsf{mid}_2$) 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 corresponding tuples in $\vec{M}$.
  2. If the simulator knows for both mid_1 and mid_2 $\mathcal{F}_{\text{StX-ED}}$-message-IDs $\widehat{\mathsf{mid}_1}$ and $\widehat{\mathsf{mid}_2}$ send (SWAP, $\mathsf{sid}, \widehat{\mathsf{mid}_1}, \widehat{\mathsf{mid}_2}$) to $\mathcal{F}_{\text{StX-ED}}$.
  3. Output (SWAP, sid) to $\mathcal{A}$.

State Exchange Functionality :

– Upon receiving (SUBMIT-NEW, $\mathsf{sid}, \vec{\mathtt{st}}, p_s, (p_1, \widehat{\mathsf{mid}_1}), \ldots, (p_n, \widehat{\mathsf{mid}_n})$) from $\mathcal{F}_{\text{StX-ED}}$ 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, $\mathsf{sid}, \mathcal{C}, p_s, (p_1, \mathsf{mid}_1), \ldots, (p_n, \mathsf{mid}_n)$) to the adversary.
  2. Otherwise find a chain $\mathcal{C}'$ in $\mathcal{T}$ with state $\mathtt{st}_1, \ldots, \mathtt{st}_{k-1}$[c]. Choose a random nonce n and set $\mathbf{B}_k = (H[\mathbf{B}_{k-1}], \mathtt{st}_k, \mathsf{n})$ and set $H[\mathbf{B}_k]$ to a uniform random value in $\{0,1\}^\kappa$ strictly smaller than 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, $\mathsf{sid}, \mathcal{C}, p_s, (p_1, \mathsf{mid}_1), \ldots, (p_n, \mathsf{mid}_n)$) to the adversary.

---

[a] This can be checked efficiently using H under the assumption that there are no collisions.
[b] Can be done efficiently using rejection sampling.
[c] Such a chain must exist as $\mathtt{st}_1, \ldots, \mathtt{st}_{k-1}$ is a successfully submitted state in $\mathcal{F}_{\text{StX-ED}}$ in which case the simulator knows a corresponding chain.

The proof works similar as the one for Lemma 5.1 in [29].Denote by $\text{EXEC}_{Real,\mathcal{A},\mathcal{E}}$ the joint view of all parties in the execution of $\mathsf{StateExchange\text{-}Protocol}$ for adversary $\mathcal{A}$ and environment $\mathcal{E}$. Denote by $\text{EXEC}_{Ideal,\mathcal{A},\mathcal{E}}$ the joint view of all parties for $\mathcal{F}_{\text{STX-ED}}$ with simulator $\mathsf{sim}_{\text{stx}}$.

Define $\text{HYB}_{\mathcal{A},\mathcal{E}}$ the same as $\text{EXEC}_{Real,\mathcal{A},\mathcal{E}}$ except that the random oracle aborts on collisions with `collision-error` and that adversarial oracle queries are emulated as in $\mathsf{sim}_{\text{stx}}$. The only difference is thus that in $\text{HYB}_{\mathcal{A},\mathcal{E}}$ we may abort with `collision-error` or `tree-error`.

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. 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 $\text{HYB}_{\mathcal{A},\mathcal{E}}$ will not abort with `collision-error`.

Let $\mathsf{event2}$ be the event that some party makes a query $H[(\mathsf{s}, \cdot, \cdot)]$ where no v exists such that $H[v] = \mathsf{s}$, but later some party makes a query v' such that $H[v'] = \mathsf{s}$. The probability that any query $H[(\mathsf{s}, \cdot, \cdot)]$ a later query returns 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 $\text{HYB}_{\mathcal{A},\mathcal{E}}$ conditioned under $\mathsf{event1}$ and $\mathsf{event2}$ not happening. Assume for contradiction that $\text{HYB}_{\mathcal{A},\mathcal{E}}$ 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 $\text{HYB}_{\mathcal{A},\mathcal{E}}$ 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 $\mathbf{B}_i = (\mathsf{s}_i, x_i, \mathsf{n}_i)$. Since $\mathcal{C}$ is a valid chain we have $H[(\mathsf{s}_n, x_n \mathsf{n}_n)] < D$. But at the time $\mathbf{B}_n$ was added to H no valid chain existed where the last block has hash value s_n (otherwise $\mathcal{C}$ would be in $\mathcal{T}$). This implies that no earlier query to H could have returned 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 $\text{HYB}_{\mathcal{A},\mathcal{E}}$ proceeds as $\text{EXEC}_{Real,\mathcal{A},\mathcal{E}}$. It follows that $\text{EXEC}_{Real,\mathcal{A},\mathcal{E}}$ and $\text{HYB}_{\mathcal{A},\mathcal{E}}$ are statistically close.

Now we compare $\text{HYB}_{\mathcal{A},\mathcal{E}}$ and $\text{EXEC}_{Ideal,\mathcal{A},\mathcal{E}}$. Consider the event where a honest miner queries a block $(\mathsf{s}, x, \mathsf{n})$ and fails, i.e. where $H[(\mathsf{s}, x, \mathsf{n})] > \mathsf{D}$. In $\text{HYB}_{\mathcal{A},\mathcal{E}}$ this query is stored in the random oracle table while the simulator in $\text{EXEC}_{Ideal,\mathcal{A},\mathcal{E}}$ does not store the query in the random oracle table. Under the condition that such failed queries are never queried again $\text{HYB}_{\mathcal{A},\mathcal{E}}$ and $\text{EXEC}_{Ideal,\mathcal{A},\mathcal{E}}$ are identically distributed as the network simulation in $\mathsf{sim}_{\text{stx}}$ is perfect. Note that the nonce n in a 'failed' query $(\mathsf{s}, x, \mathsf{n})$ is choosen uniform at random from $\{0, 1\}^{\kappa}$. This implies that with probability $\mathsf{poly}(\kappa) \cdot 2^{-\kappa}$ it was never queried before. As honest miner discard 'failed' queries it also follows that except with probability $\mathsf{poly}(\kappa) \cdot 2^{-\kappa}$ the query will not be queried again (by honest parties or $\mathcal{A}$). 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 $\text{HYB}_{\mathcal{A},\mathcal{E}}$ and $\text{EXEC}_{Ideal,\mathcal{A},\mathcal{E}}$ are statistically close.

We conclude the proof for Lemma 1, by giving a game-hopping argument to show that $\mathsf{Ledger\text{-}Protocol}$ UC emulates the protocol $\mathsf{Modular\text{-}Ledger\text{-}Protocol}$. We start with the original $\mathsf{Ledger\text{-}Protocol}$ and consider the protocol part in Figure 1 where will alter the protocol.

Protocol Original Protocol Part

Initialization:

The protocol manages the local state $\vec{\mathtt{st}}$ encoded in the chain $\mathcal{C}_{loc}$ which initially contains genesis block, i.e., $\mathcal{C}_{loc} \leftarrow (\mathbf{G})$.

ExtendState(st):

$\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\mathsf{D}}(\mathcal{C}_{loc}, \mathtt{st}, q)$
if $\mathcal{C}_{\text{new}} \neq \mathcal{C}_{loc}$ **then**
  Update the local chain, i.e., $\mathcal{C}_{loc} \leftarrow \mathcal{C}_{\text{new}}$.
end if
// Broadcast current chain
Send ($\textsc{multicast}, \mathsf{sid}, \mathcal{C}_{loc}$) to $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$

FetchInformation:

// Update the local state
Send ($\textsc{fetch}, \mathsf{sid}$) to $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$; denote the response from $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ by ($\textsc{fetch}, \mathsf{sid}, b$).
Extract valid chains $\mathcal{C}_1, \dots, \mathcal{C}_k$ from b.
Set $\mathcal{C}_{loc}$ to the longest valid chain in $\mathcal{C}_{loc}, \mathcal{C}_1, \dots, \mathcal{C}_k$ (to resolve ties the ordering decides).
// Fetch new transactions and add them to the buffer
...

Fig. 1. The original protocol part from Ledger-Protocol

The first modification of the protocol (cf. Figure 2) 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 $\vec{\mathtt{st}}||\mathtt{st}$ is a valid state and that there exists a chain in $\mathcal{T}$ which encodes the state $\vec{\mathtt{st}}$. We observe that the protocol calls **ExtendState(st)** only with st where $\vec{\mathtt{st}}||\mathtt{st}$ is a valid state. This implies that the first check is always satisfied. Moreover, not that the current local chain $\mathcal{C}_{loc}$ which encodes state $\vec{\mathtt{st}}$ is at any time stored in the tree $\mathcal{T}$. The second check is therefore also always satisfied. Hence, the modified protocol has the same input/output behavior has Ledger-Protocol.

Protocol Modification 1

Initialization:

The protocol manages the local state $\vec{\mathtt{st}}$ encoded in the chain $\mathcal{C}_{loc}$ which initially contains 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 isvalidstate(st⃗||st) = 1 then
    if there exists C ∈ T which encodes st⃗ then
        C_new ← extendchain_D(C_loc, st, q)
        if C_new ≠ C_loc then
            Update the local chain, i.e., C_loc ← C_new.
            Add C_loc to T
        end if
        // Broadcast current chain
        Send (MULTICAST, sid, C) to F^bc_A-MC
    end if
end if
```

FetchInformation:

```
// Update the local state
Send (FETCH, sid) to F^bc_A-MC; denote the response from F^bc_A-MC by (FETCH, sid, b).
Extract all valid chains C_1, ..., C_k from b and add them to T.
Set C_loc to the longest valid chain in C_loc, C_1, ..., C_k (to resolve ties the ordering decides).
// Fetch new transactions and add them to the buffer
...
```

Fig. 2. Modification 1: Added a chain history $\mathcal{T}$ and modified **ExtendState(st)**

In the next modification (cf. Figure 3) the local state $\vec{\mathtt{st}}$ 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.

In the next modification (cf. Figure 4) 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.

Finally consider Modular-Ledger-Protocol in Figure 5 which is the same as **Modification** 3 except that the chain storage $\mathcal{T}$ and the calls to sub-procedures SUBMIT-NEW, CONTINUE, and FETCH-NEW are replaced by the calls to $\mathcal{F}_{\text{STX-ED}}$. Lemma 3 implies that the behavior of **Modification** 3 and Modular-Ledger-Protocol is the same. This concludes the game-hopping argument.

Protocol Modification 2

Initialization:

The protocol manages the local state $\vec{\mathtt{st}}$ which initially is set to 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 isvalidstate(st⃗||st) = 1 then
    if there exists C ∈ T which encodes st⃗ then
        C_new ← extendchain_D(C, st, q)
        if C_new ≠ C then
            Add C to T
            Update the local state, i.e., st⃗ ← st⃗||st.
        end if
        // Broadcast current chain
        Send (MULTICAST, sid, C) to F^bc_A-MC
    end if
end if
```

FetchInformation:

// Update the local state
Send $(\text{FETCH}, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$; denote the response from $\mathcal{F}^{\mathsf{bc}}_{\text{A-MC}}$ by $(\text{FETCH}, \mathsf{sid}, b)$.
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 $\vec{\mathtt{st}}$ to the longest state in $\vec{\mathtt{st}}, \vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k$ (to resolve ties the ordering decides).
// Fetch new transactions and add them to the buffer
...

Fig. 3. Modification 2: State $\vec{\mathtt{st}}$ is stored directly

Protocol Modular-Ledger-Protocol **Part**

Initialization:

The protocol manages the local state $\vec{\mathtt{st}}$ encoded in the chain $\mathcal{C}_{loc}$ which initially contains genesis block, i.e., $\mathcal{C}_{loc} \leftarrow (\mathbf{G})$.

ExtendState(st):

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

FetchInformation:

// Fetch new states and update the local state
Send $(\text{FETCH-NEW}, \mathsf{sid})$ to $\mathcal{F}_{\text{StX-ED}}$.
Denote the response from $\mathcal{F}_{\text{StX-ED}}$ by $(\text{FETCH-NEW}, \mathsf{sid}, (\vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k))$.
Set $\vec{\mathtt{st}}$ to the longest state in $\vec{\mathtt{st}}, \vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_k$ (to resolve ties the ordering decides).
// Fetch new transactions and add them to the buffer
...

Fig. 5. The protocol part from Modular-Ledger-Protocol

Protocol Modification 3

Initialization:

The protocol manages the local state $\vec{\mathtt{st}}$ encoded in the chain $\mathcal{C}_{loc}$ which initially contains 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($\mathtt{st}$):

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

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

if $\mathsf{isvalidstate}(\vec{\mathtt{st}}||\mathtt{st}) = 1$ **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
    $\mathcal{C}_{\text{new}} \leftarrow \mathsf{extendchain}_{\mathsf{D}}(\mathcal{C}, \mathtt{st}, q)$
    if $\mathcal{C}_{\text{new}} \neq \mathcal{C}$ **then**
      Add $\mathcal{C}$ to $\mathcal{T}$
      return 1
    end if
    return 0
  end if
end if

Procedure CONTINUE:

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

FetchInformation:

// Update the local state
$(\vec{\mathtt{st}}_1, \dots, \vec{\mathtt{st}}_k) \leftarrow$ FETCH-NEW
Set $\vec{\mathtt{st}}$ to the longest state in $\vec{\mathtt{st}}, \vec{\mathtt{st}}_1, \dots, \vec{\mathtt{st}}_k$ (to resolve ties the ordering decides).
// Fetch new transactions and add them to the buffer
...

Procedure FETCH-NEW:

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

Fig. 4. Modification 3 Modularization into sub-procedures

D.3 Proof of Theorem 1

In the following we provide a proof for Theorem 1.

Theorem (1). *Let the functions* $\mathsf{ValidTx}_{BC}$, $\mathsf{blockify}_{BC}$, *and* $\mathtt{ExtendPolicy}$ *be as defined above. Let* $p \in (0,1)$, *integer* $q \geq 1$ *and let* $p_H = 1-(1-p)^q$ *and* $p_A = p$. *Let* $\Delta \geq 1$ *be the upper bound on the network delay.*

Consider $\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T$ *in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{F}_{\text{STX-ED}}^{p_H,p_A}, \mathcal{F}_{\text{A-MC}})$*-hybrid world. If, for some* $\lambda > 1$, *the relation*

$$\alpha \cdot (1 - 2 \cdot (\Delta + 1) \cdot \alpha) \geq \lambda \cdot \beta \tag{2}$$

is satisfied in any real-world execution, where α *and* β *are defined as above, then the protocol* $\mathsf{Modular\text{-}Ledger\text{-}Protocol}_T$ *UC-realizes* $\mathcal{G}_{\text{LEDGER}}^{\mathcal{B}}$ *for any parameters in the range*

$$\mathtt{slack} = T \quad \text{and} \quad \mathtt{Delay} = 4\Delta,$$

$$\mathsf{window}^{\text{slack}}_{\text{minGrow}} \geq \frac{\mathtt{slack}}{(1-\delta)\cdot\gamma} \quad \text{and} \quad \mathsf{window}^{\text{slack}}_{\text{maxGrow}} \leq \frac{\mathtt{slack}}{(1+\delta)\cdot \max_r \mathsf{T_{mp}}(r)},$$

$$\eta > (1+\delta)\cdot \mathtt{slack} \cdot \frac{\beta}{\gamma},$$

where $\gamma := \frac{\alpha}{1+\Delta\alpha}$ *and* $\delta > 0$ *is an arbitrary constant. In particular, the realization is perfect except with probability* $R \cdot \mathsf{negl}(T)$, *where* R *denotes the upper bound on the number of rounds.*

Proof. In order to show the theorem we specify the simulator for the ideal world $\mathsf{sim}_{\text{ledg}}$. $\mathsf{sim}_{\text{ledg}}$ is specified below as pseudo-code. Let us explain the general structure: 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 $\mathcal{F}_{\text{STX-ED}}$, then the simulator will assemble the views of its simulated honest 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 AdjustVisibility). 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 start by analyzing the real-world execution $\text{EXEC}_{\mathsf{Modular\text{-}Ledger\text{-}Protocol},\mathcal{A},\mathcal{E}}$. We follow the detailed analysis provided by Pass, Seeman, and shelat to analyze the real-world execution. The analysis is divided into six different claims about the real-world execution. They include properties such as a lower-bound on the chain growth (Claim 2.), the chain quality (Claim 3.), or an upper-bound on the chain growth (Claim 6.). For completeness, we prove the Claims 1.-6. below, which represent the core arguments (with some slight adaptations) also in our setting. 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 $R \cdot \mathsf{negl}(T)$, where R denotes the number of rounds the protocol is running and $\mathsf{negl}(\cdot)$ denotes a negligible function in the parameter T.

By abusing a bit of notation, use the term $\text{EXEC}_{Real,\mathcal{A},\mathcal{E}}$ to refer to the entire random experiment defined by the UC execution of a protocol with adversary $\mathcal{A}$ and environment $\mathcal{E}$ (instead of only to the output of $\mathcal{E}$). We then denote by $f(\text{EXEC}_{Real,\mathcal{A},\mathcal{E}})$ the induced random variable where f is a function on the entire view of an execution. For notational simplicity, for Claims 1 to 6, if we speak of *honest* miners, we always mean *honest and synchronized* miners.

Claim. 1. State dissemination: *Let p_i and p_j be miners, and let $r \geq 0$. Assume p_i is honest in round r, and the longest state received or stored by p_j has length ℓ. For any honest miner p_j in round $r + \Delta$, it holds that the longest state received or stored by p_j has length at least ℓ.*

<u>Proof</u>: By assumption, all messages, and in particular transmitted states of honest miners, are delayed maximally by Δ rounds. Thus, if an honest miner receives a state of length ℓ, then any other honest miner will receive this state within the next Δ rounds. Additionally, if a honest miner successfully mines a block, this new state will arrive at any honest miner at latest after Δ rounds. By then, any honest miner will have adopted a chain of length at least ℓ. ■

Claim. 2. Minimal number of mined blocks: *Let p_i be a miner, and let $r \geq 0$. Assume p_i is honest 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 $R \cdot \mathsf{negl}(T)$, that the length of the longest state (received or stored) of at least one honest miner p_j in that round has length at least $\ell + T$ if $t \geq \frac{T}{(1-\delta)\cdot\gamma}$.*

<u>Proof</u>: 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 starting and prove that in a real-world execution with $\mathcal{A}'$ the expected state length of an honest 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 $r + t$.

Construction of $\mathcal{A}'$: Given adversary $\mathcal{A}$, the adversary $\mathcal{A}'$ works as follows. It internally runs $\mathcal{A}$ until round r without any modifications. At round r, $\mathcal{A}'$ first delays all current messages in the network to the maximally possible delay. Also, from round r onward, whenever an honest 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 starting from round r.

We define the function $\textsc{Len}_i^r(\text{EXEC}_{Real,\mathcal{A}(\sigma),\mathcal{E}})$ to be the length of the longest chain (honest) miner i in round r in the real world experiment. Further, let $\mathcal{A}(\sigma)$ denote the behavior of $\mathcal{A}$ on (internal) randomness σ, further, denote by $Real(\sigma')$ the real-system, where the internal randomness of $\mathcal{F}_{\text{StX-ED}}$ is fixed to σ'. We give an inductive proof to show that for any $r > 0$, $\textsc{Len}_i^{r+t}(\text{EXEC}_{Real(\sigma'),\mathcal{A}(\sigma),\mathcal{E}}) \geq \textsc{Len}_i^{r+t}(\text{EXEC}_{Real(\sigma'),\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{E}})$.

Base Case, $t = 0$: Since adversary $\mathcal{A}'$ and $\mathcal{A}'$ behave identical up to and including round $r-1$, 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 σ is fixed. Furthermore, when the randomness σ' of $\mathcal{F}_{\text{StX-ED}}$ 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, $\textsc{Len}_i^r(\text{EXEC}_{Real(\sigma'),\mathcal{A}(\sigma),\mathcal{E}}) < \textsc{Len}_i^r(\text{EXEC}_{Real(\sigma'),\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{E}})$ only if player i receives a longer state in round r. Since $\mathcal{A}'$ additionally maximally delays messages, 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 base case.

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

$$\textsc{Len}_i^{r+t}(\text{EXEC}_{Real(\sigma'),\mathcal{A}(\sigma),\mathcal{E}}) \geq \textsc{Len}_i^{r+t}(\text{EXEC}_{Real(\sigma'),\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{E}}).$$

We argue that $\textsc{Len}_i^{r+t+1}(\text{EXEC}_{Real(\sigma'),\mathcal{A}(\sigma),\mathcal{E}}) \geq \textsc{Len}_i^{r+t+1}(\text{EXEC}_{Real(\sigma'),\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{E}})$ holds as well. Assume not, then, by the above reasoning, it can only be due to miner i receiving a state in round $r + t + 1$ that increases $\textsc{Len}_i^{r+t+1}(\text{EXEC}_{Real(\sigma'),\mathcal{A}'(\mathcal{A}(\sigma)),\mathcal{E}})$ but not $\textsc{Len}_i^{r+t+1}(\text{EXEC}_{Real(\sigma'),\mathcal{A}(\sigma),\mathcal{E}})$ (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. Taking the expectation over the randomness σ and σ', we conclude that for any round r, any $c \geq 0$, for the real UC-execution, it holds that

$$\begin{aligned}&\Pr[\textsc{Len}_i^{r+t}(\text{EXEC}_{Real,\mathcal{A},\mathcal{E}}) \leq \textsc{Len}_i^{r}(\text{EXEC}_{Real,\mathcal{A},\mathcal{E}}) + c] \\ &\leq \Pr[\textsc{Len}_i^{r+t}(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) \leq \textsc{Len}_i^{r}(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) + c].\end{aligned}$$

We say a round r' is *uniform* if $\textsc{Len}_i^{r'}(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) = \textsc{Len}_j^{r'}(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}})$ for all honest 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 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 $r + \Delta$.

Fix some round r. If in round $s = r + t$, the length increase of the overall longest state of an honest miner is less than c blocks, then at most $c \cdot \Delta$ non-uniform rounds occurred. Hence, there were at least $t - c \cdot \Delta$ uniform rounds. The probability that less than c new blocks are mined by honest miners (i.e., that less than c successful queries by honest miners to $\mathcal{F}_{\text{StX-ed}}$ extended the state by one block) is thus $\Pr[\sum_{i=1}^{t-c\Delta} X_i < c]$, where X_i is a boolean random variable with mean α. Let $X := \sum_{i=1}^{t-c\Delta} X_i$ with mean $E[X] = \alpha \cdot t - a \cdot c \cdot \Delta$. To get an appropriate tail-estimate, we set $c = \frac{\alpha t}{1+\alpha\Delta}$ to obtain $E[X] = c$ and can apply the Chernoff bound to get

$$\begin{aligned}
&\Pr[\textsc{Len}_i^s(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) \leq \textsc{Len}_i^r(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) + (1-\delta)c] \\
&\leq \Pr[\textsc{Len}^s(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) \leq \textsc{Len}^r(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) + (1-\delta)c] \\
&\leq \Pr[X < (1-\delta)c] = \Pr[X < (1-\delta)\gamma t] \leq \exp\Big(\frac{-\delta^2}{2} \cdot \gamma t\Big).
\end{aligned}$$

where we plugged-in the definition of γ.

By Claim 1 (chain dissemination), we know that if some honest party has a state in round r, then any honest miner will have a state of at least this length in round $r + \Delta$. Hence, for realizing the ledger, we see that state extend happens if at least one honest miner has a new state which happens at a rate at least γt (except with probability $\mathsf{negl}(\gamma t) = \mathsf{negl}(T)$ by the lower bound on t).

By Claim 1, we see that if an honest miner knows some state, then within Δ rounds, every other honest miner will be aware of that. A similar calculation shows that the lower bound on the time to have a state increase by T blocks by all honest parties follows the same law (and hence the perceived ledger speed is the same). By requiring $s = r + t - \Delta$ above (such that in round $r + t$ all honest miners have a state that increased by at least $(1-\delta)\gamma t$ since round r), and letting $Y := \sum_{i=1}^{t-\Delta-c\Delta} X_i$ (for X_i as above), we get

$$\begin{aligned}
&\Pr[\textsc{Len}^s(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) \leq \textsc{Len}^r(\text{EXEC}_{Real,\mathcal{A}',\mathcal{E}}) + (1-\delta)c] \\
&\leq \Pr[Y < (1-\delta)\gamma(t-\Delta)] \leq \exp\Big(\frac{-\delta^2}{2} \cdot \gamma(t-\Delta)\Big). \qquad (3)
\end{aligned}$$

Since $\gamma t - 1 < \gamma t - \gamma\Delta < \gamma t$, this implies that $\Pr[Y < (1-\delta')\gamma t] \leq \exp\Big(\frac{-\delta^2}{2} \cdot \gamma t\Big)$ for any δ' by choosing a sufficiently small constant δ in Equation 3, yielding a negligible function in $\gamma t \geq T$. Finally, since α (and thus γ) is the lower bound for any round r, taking the union bound over the polynomial number of rounds yields Claim 2. ■

The following claims use the fact that Equation 1 implies that there exists $0 < \hat{\delta} < 1$ such that 1.) $\alpha > \gamma > (1+\hat{\delta})\beta$, and 2.) $(1-\hat{\delta}) > (\Delta+1)\alpha$. This is proven in [29] (Claim 6.12).

Claim. 3. Fraction of honest blocks: *Let p_i be a miner, and let $r \geq 0$. Assume p_i is honest in round r, and 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* $\min\{1, (1+\delta) \cdot \frac{\beta}{\gamma}\}$ *except with probability* $R \cdot \mathsf{negl}(T)$ *for any* $\delta > 0$.

Proof: Let us assume that at round r', the state of 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 miners (unless $j + T$ is the maximum length of any state known to $\mathcal{F}^{\delta}_{\text{StX-BD}}$). Otherwise, one can enlarge T to meet this condition, as any state is finite and starts with the genesis block. We further assume that $\vec{\texttt{st}}^{<j}$ is mined at round r, and that in round $r+t$, the state $\vec{\texttt{st}}^{>j+T}$ appears for the first time as the state, or the prefix of a state, of at least one honest miner. We conclude that if an adversary successfully extended the state during some round by a new state block $\texttt{st}_{j+s}$ of the above sequence $\texttt{st}_{j+1}, \ldots, \texttt{st}_{j+T}$, 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 miner adopted a state containing $\texttt{st}_{j+1}$, we can invoke Claim 2, to conclude that the probability that the condition $t > \frac{T}{(1-\delta')\gamma}$ occurs in such an execution is at most $\mathsf{negl}(T)$. We hence have $t \leq \frac{T}{(1-\delta')\gamma}$ with overwhelming probability in T.

On the other hand, we can lower bound the number of rounds needed to generate a state increase by T blocks by standard Chernoff bound: using the relation of $p_H = 1-(1-p)^q$ our assumed $\mathcal{F}_{\text{StX-ED}}$, we have $\mathsf{T_{mp}}(r) \leq (q_A^{(r)} + q \cdot q_H^{(r)}) \cdot p$. Let $\mathsf{T_{mp}} := \max_{r \in [R]} \mathsf{T_{mp}}(r)$ and denote by $q_{\max}$ be the corresponding upper bound on the query power. The event that during t rounds, more than $T = (1+\delta) \cdot q_{\max} \cdot p \cdot t$ times a successful state extension happens (via a query to $\mathcal{F}_{\text{StX-ED}}$), occurs with probability $\exp\Big(-\frac{\delta^2}{2} q_{\max} pt\Big) \in \mathsf{negl}(T)$ only. Stated differently, with overwhelming probability, for the assumed sequence of T state blocks, the number of rounds t needed to mine the new state is at least $\frac{T}{(1+\delta)q_{\max}p}$.

Additionally, we know that the upper bound on the expected number of adversarial successes to extend a state in one round is β, and the upper bound on the expected number of successes (i.e., newly minded state blocks) within t rounds by the adversary, denoted as the random variable N_A^t, is thus $t \cdot \beta$ by linearity of expectation. The random variable N_A^t is hence the sum of $q'_{\max}$ binary random variables (where $q'_{\max}$ is the maximum number of queries contributed by corrupted as well as by honest but de-synchronized miners) each being successful with probability p. By the Chernoff bound, we get a tail-estimate of

$$\Pr[N_A^t > (1+\delta)t \cdot \beta] \leq \exp\Big(\frac{-\delta^2}{2} t\beta\Big).$$

By the above lower bound on t and since β is the maximum expected value over all rounds, and recall that β is a ρ fraction of the maximum mining power $\mathsf{T_{mp}}$ of a round within this t-round interval, we conclude that $\frac{\rho T}{1+\delta} \leq \beta t$ holds with overwhelming probability. Hence, the above function is indeed a negligible function in T. Therefore, 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

$$N_A^{\frac{T}{(1-\delta')\gamma}} \leq \frac{1+\delta}{1-\delta'} \cdot T \cdot \frac{\beta}{\gamma}.$$

Hence, the fraction of adversarial blocks within T consecutive blocks cannot be more than $f = \min\{1, (1+\delta'')\frac{\beta}{\gamma}\}$ for any δ'' and sufficiently small constants $\delta, \delta' > 0$, except with negligible probability in the length T of the sequence. By Equation 1, we even have $f < 1$ (for an appropriate choice of δ''). Since β is the maximum expected value in any round, the proof is concluded by taking the union bound over the number of such sequences (which is in the order of number of rounds). ■

Claim. 4. Withholding of adversarial block: *For any round r, the probability that a new state accepted by an honest miner was mined before round $r - \omega t$ happens with probability $\mathsf{negl}(\beta t)$, for any $0 < \omega < 1$.*

Proof: Let us define $\vec{st}_r = \texttt{st}_0 || \ldots || \texttt{st}_k$ to be the longest state known to $\mathcal{F}_{\text{StX-ED}}$ at round r. Let $\vec{\texttt{st}}_h$ be the longest prefix of $\vec{\texttt{st}}_r$ such that $\vec{\texttt{st}}_{r'}$ was mined by an honest miner or it is the genesis block. This state was known to at least one honest party by round $r' \leq r$. Now, assume that $r - r' \geq \omega t$. Then, by the chain growth lowerbound, we know that except with probability $\mathsf{negl}(\gamma t)$, $|\vec{\texttt{st}}_r| - |\vec{\texttt{st}}_{r'}| \geq (1-\delta) \cdot \gamma \omega t$. Since $\gamma t > \beta t$, we have that this holds except with probability $\mathsf{negl}(\beta t)$.

Analogously to the previous claim, the number of new states mined by the adversary is upper bounded by $(1+\delta') \cdot \beta \omega t$ (except with probability $\mathsf{negl}(\beta t)$). Since we assumed that all $|\vec{\texttt{st}}_r| - |\vec{\texttt{st}}_{r'}|$ blocks are mined by the adversary we have $|\vec{\mathtt{st}}_r| - |\vec{\mathtt{st}}_{r'}| \leq (1+\delta') \cdot \beta\omega t$. We get

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

which, for sufficiently small δ, δ' implies that $\gamma < (1+\delta'')\beta$ for any δ''. This contradicts Equation 1 and the claim follows. ■

Claim. 5. Consistent states: *Let p_i and p_j be miners, and let $r' \geq r \geq 0$. Assume p_i is honest in round r, and p_j is honest 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 $R \cdot \mathsf{negl}(T)$.*

Proof: We again follow the exposition in [29]. Since an inconsistency at round r implies an inconsistency at round $r' > r$, if the claim is proven for the case that $r \leq r' \leq r+1$, then by an inductive argument, the claim holds for any $r' \geq r$.

The protocol for the honest miners truncates the T newest blocks from the current respective state of each miner. 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 miner. Suppose we are at round r' in the protocol, then the time it takes to generate the last T blocks is at least $t \geq \frac{T}{(1+\delta)q_{\max}p}$ except with negligible probability as argued in the previous claim, where $q_{\max}$ is the maximum number of queries per round of all registered miners (maximum over all rounds). We can thus follow the argument by [29] to conclude that the probability that the states of two honest miners diverge at round $s := r - t$ is a negligible function $\mathsf{negl}(\beta t)$ and thus also a negligible function in T. The last step follows as in the previous claim by observing that $\beta t \geq \frac{\rho}{(1+\delta)}T$, where β is the maximum expected value of adversarially mined blocks (over all rounds).

In the interval between s and r', the expected number of rounds, where at lest one honest miner is successful, is at least αt. Thus, by a standard Chernoff bound, the probability that the number of these successful rounds is smaller than $\bar{q}_{min} := (1-\delta') \cdot \alpha t$ is no more than $\exp\Big(-\frac{\delta'^2}{2}\alpha t\Big)$ in the real-world UC random experiment. Since Equation 1 implies that there exists a $\hat{\delta} \in (0,1)$ such that $\alpha > \beta \cdot (1+\hat{\delta})$, this probability is upper bounded by $\mathsf{negl}(\beta t)$.

In addition, 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 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. By a union bound we have that $\Pr[X_i = 0] = \Pr[E_1^i \cup E_2^i] \leq \Delta\alpha + \alpha$, hence $\Pr[X_i = 1] \geq 1 - \alpha(\Delta+1)$. Let $X := \sum_{i=1}^{\bar{q}_{min}} X_i$, where $E[X] = (1-\alpha(\Delta+1)) \cdot \bar{q}_{min}$. Let us define $\bar{q}'_{min} := (1-\delta'') \cdot (1-\alpha(\Delta+1)) \cdot \bar{q}_{min}$. Since the random variables X_i are independently distributed, it follows that $\Pr[X \leq \bar{q}'_{min}] \leq \exp\Big(-\frac{\delta''^2}{2}(1-\alpha(\Delta+1)) \cdot \bar{q}_{min}\Big)$. Aside of $\alpha > \beta$, Equation 1 implies that there exists some $0 < \hat{\delta} < 1$ such that $\alpha(\Delta+1) < 1 - \hat{\delta}$. We conclude that $(1-\alpha(\Delta+1)) \cdot \bar{q}_{min} > \hat{\delta}(1-\delta') \cdot \alpha t > \hat{\delta}(1-\delta') \cdot \beta t$. Thus, $\Pr[X \leq \bar{q}'_{min}] \leq \exp\Big(-\frac{\delta''^2}{2}\hat{\delta}(1-\delta') \cdot \beta t\Big) \in \mathsf{negl}(\beta t)$.

We are interested in the number of times that $X_i = X_{i+1} = 1$. Intuitively, this means that we have a situation, in which for Δ rounds, no honest miner is successful, then exactly one honest miner is successful, and afterwards, we again have Δ rounds where no honest miner is successful. This is denoted in [29] 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 miners during this period, all honest 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.

Since with overwhelming probability, there are at least $\bar{q}_{min}$ successful rounds, among which at most $\bar{q}_{min} - \bar{q}'_{min}$ could prevent a convergence opportunity, the number of convergence opportunities C is, except with probability $\mathsf{negl}(\beta t)$, lower bounded by $C \geq \bar{q}_{min} - 2(\bar{q}_{min} - \bar{q}'_{min}) = 2\bar{q}'_{min} - \bar{q}_{min} > (1-\epsilon)(1-2\alpha(\Delta+1))\alpha t$, for any constant ϵ (by picking δ' and δ'' sufficiently small).

The final argument is a counting argument. Let us denote by $\mathcal{S}_r$ the set of maximal states known to $\mathcal{F}^{\delta}_{\text{STX-BD}}$ 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 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 $\mathcal{F}^{\delta}_{\text{STX-BD}}$) between round s and r' is $T_{\mathcal{A}} < C$. Then, by the pigeonhole principle, for all $\vec{\mathtt{st}} \in \mathcal{S}_r$, it holds that there is at least one block $\mathtt{st}_k$, such that functionality $\mathcal{F}_{\text{STX-ED}}$ is successfully queried by exactly one 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 miners have a state whose first $k+1$ blocks are $\vec{\mathtt{st}}_i = \mathtt{st}_0 \ldots, \mathtt{st}_k$. Unless the adversary provides an alternative state with a prefix $\vec{\mathtt{st}}'_i$ of length $k+1$, such that $\mathtt{st}'_l \neq \mathtt{st}_l$ for at least one index $0 < l \leq k$, no honest miner will ever mine on a state whose first $k+1$ blocks do not agree with $\vec{\mathtt{st}}_i$.

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 βt.

First, for any $\omega > 0$, the probability that any new state accepted by an honest miner during the period of $(t+1)$ rounds is mined before round $s - \omega t$ is at most $\mathsf{negl}(\beta t)$. Analogously to the previous claim, the number of adversarial blocks (i.e., successful state extensions by $\mathcal{A}$) generated within $(1+\omega)(t+1)$ rounds is (except with probability $\mathsf{negl}(\beta t)$) upper bounded by $T_{\mathcal{A}} \leq (1+\delta)(1+\omega)(t+1)\beta \leq \frac{(1+\delta)(1+\omega)}{\lambda}(t+1)\alpha \cdot (1-2\alpha \cdot (\Delta+1))$, where the last inequality follows from Equation 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 $\mathsf{negl}(\beta t)$. Since β is the maximal value over all β_r for any round, we again apply the union bound over the number of rounds and get the desired claim. ■

Claim. 6. Maximal number of mined blocks *Let p_i be a miner, and let $r \geq 0$. Assume p_i is honest 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 $R \cdot \mathsf{negl}(T)$, that the length of the longest state (received or stored) of at least one honest miner p_j in that round has length at most $\ell + T$ if $t \leq \frac{T}{(1+\delta)\cdot p \cdot q_{\max}}$, where $q_{\max}$ is the maximum query power in any round of this interval.*

Proof: To upper bound the number of accepted blocks, we have to combine two observations made above: we have seen that the time it takes to generate T blocks is at least $t \geq \frac{T}{(1+\delta)q_{\max}p}$ except with probability $\mathsf{negl}(\beta t)$. Hence, with overwhelming probability, in $t \leq \frac{T}{(1+\delta)\cdot p \cdot q_{\max}}$, no more than T blocks are mined. Furthermore, we have seen in Claim 4 that for any round r, the probability that a new state is accepted by an honest miner but this state was mined before round $r - \omega t$ happens with probability $\mathsf{negl}(\beta t)$, for any $0 < \omega < 1$. Thus, in the interval between r and t, for t as bound in the Claim statement, the state can increase by at most T state blocks except with probability $\mathsf{negl}(T)$, since we have again, $\beta t \geq \frac{\rho}{(1+\delta)}T$. Since β is the maximum expected value of adversarially mined blocks (over all rounds), the claim follows by taking the union bound over all rounds. ■

Finally, we conclude the proof by noting that after a delay of Δ rounds, all honest transactions are known to all honest miners, so, as soon as an honest miner mines the next state block, he for sure puts all these transactions into his next blocks if they are 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 ExtendPolicy of $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$. 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. Finally, setting $\texttt{slack} \geq T$, follows from the arguments of Claim 4 and 5, since the adversary cannot have mined a state that is T blocks larger than some chain from a honest and synchronized party. But this would be a necessary condition to provoke a slackness that exceeds T. This concludes the proof. □

It follows the formal specification of the simulator.

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

Initialization:

The simulator manages internally a simulated state-exchange functionality $\mathcal{F}_{\text{StX-ED}}$, a simulated network $\mathcal{F}_{\text{A-MC}}$. An honest miner p registered to $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$ is assumed to be registered in all simulated functionalities. Moreover, the simulator maintains the local state $\vec{\texttt{st}}_p$ and the buffer of transactions buffer_p of such a party. Upon any activation, the simulator will query the current party set from the ledger (and simulate the corresponding message they sent out to the network upon 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) real-world hybrid functionalities. The ideal world consists of the ledger functionality and the clock.

Interaction with the Clock:

- Upon receiving ($\text{CLOCK-READ}, \mathsf{sid}_C$) from $\mathcal{A}$ forward it to $\mathcal{G}_{\text{CLOCK}}$. Forward the returned ($\text{CLOCK-READ}, \mathsf{sid}_C, \tau$) to $\mathcal{A}$.
- Upon receiving ($\text{CLOCK-UPDATE}, \mathsf{sid}_C, p$) from $\mathcal{G}_{\text{CLOCK}}$ send ($\text{CLOCK-UPDATE}, \mathsf{sid}_C, p$) to $\mathcal{A}$.

Interaction with the Ledger:

- Upon receiving ($\text{SUBMIT}, \texttt{BTX}$) from $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$ where $\texttt{BTX} := (\texttt{x}, \texttt{txid}, \tau, p)$ forward ($\text{MULTICAST}, \mathsf{sid}, \texttt{x}$) to the simulated network $\mathcal{F}_{\text{A-MC}}$ in the name of p. Output the answer of $\mathcal{F}_{\text{A-MC}}$ to the adversary.
- Upon receiving ($\text{MAINTAIN-LEDGER}, \mathsf{sid}, \mathsf{minerID}$) from $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$, extract from $\vec{\mathcal{I}}_H^T$ the party p_i that issued this query. If p_i has already done its instructions for the current mini-round, then ignore the request. Otherwise, do:
  1. Execute SimulateMining($p_{\mathsf{minerID}}, \tau$) and if this was the last maintain command in a working mini-round, then execute ExtendLedgerState before giving the activation to $\mathcal{A}$.
  2. In addition, remember that party p_i is done with mining in the current mini-round.
- Upon any activation of the simulator, the simulator inspects the entire sequence of inputs by honest parties to the ledger $\vec{\mathcal{I}}_H^T$ and does the following:
  1. For any input, $I = (\text{READ}, \mathsf{sid})$ of party P, if the current round is an update mini-round, then execute Step 4 of the mining procedure as below in SimulateMining
  2. Remember that the update for party P is done for this round.

Simulation of the State Exchange Functionality:

- Upon receiving ($\text{SUBMIT-NEW}, \mathsf{sid}, \vec{\texttt{st}}, \texttt{st}$) from $\mathcal{A}$ on behalf of a corrupted $p \in \mathcal{P}_{stx}$, then relay it to the simulated $\mathcal{F}_{\text{StX-ED}}$ and do the following: , then give this back to $\mathcal{A}$. If $\mathcal{A}$ replies with ($\text{CONTINUE}, \mathsf{sid}$), input ($\text{CONTINUE}, \mathsf{sid}$) to the simulated $\mathcal{F}_{\text{StX-ED}}$ and do the following:
  1. If $\mathcal{F}_{\text{StX-ED}}$ returns ($\text{SUCCESS}, B$) give this reply to $\mathcal{A}$

2. If $\mathcal{A}$ replies with (CONTINUE, sid), input (CONTINUE, sid) to the simulated $\mathcal{F}_{\text{StX-ED}}$
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 $\mathcal{F}_{\text{StX-ED}}$ 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 $\mathcal{F}_{\text{StX-ED}}$.
  2. If the current mini-round is an update mini-round, then execute EXTENDLEDGERSTATE
  3. Return to $\mathcal{A}$ the return value from $\mathcal{F}_{\text{StX-ED}}$.
- Upon receiving (SWAP, $\mathsf{sid}, \mathsf{mid}, \mathsf{mid}'$) from $\mathcal{A}$, forward the request to the simulated $\mathcal{F}_{\text{StX-ED}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (DELAY, $\mathsf{sid}, T, \mathsf{mid}$) from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\text{StX-ED}}$ and do the following:
  1. Query the ledger state state
  2. Execute ADJUSTVISIBILITYstate
  3. Return to $\mathcal{A}$ the output of $\mathcal{F}_{\text{StX-ED}}$

Simulation of the Network (over which transactionss are sent) :

- Upon receiving (MULTICAST, $\mathsf{sid}, (m_{i_1}, p_{i_1}), \ldots, (m_{i_\ell}, p_{i_\ell})$ 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 $\mathtt{txid}$
  2. Forward the request to the internally simulated $\mathcal{F}_{\text{A-MC}}$, which replies for each message with a message-ID mid
  3. Remember the association between each mid and the corresponding $\mathtt{txid}$
  4. Provide $\mathcal{A}$ with whatever the network outputs.
- Upon receiving (FETCH, sid) from $\mathcal{A}$ on behalf some *corrupted* $P \in \mathcal{P}_{net}$ forward the request to the simulated $\mathcal{F}_{\text{A-MC}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (DELAYS, $\mathsf{sid}, (T_{\mathsf{mid}_{i_1}}, \mathsf{mid}_{i_1}), \ldots, (T_{\mathsf{mid}_{i_\ell}}, \mathsf{mid}_{i_\ell})$) from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\text{A-MC}}$ and return whatever is returned to $\mathcal{A}$.
- Upon receiving (SWAP, $\mathsf{sid}, \mathsf{mid}, \mathsf{mid}'$) from $\mathcal{A}$ forward the request to the simulated $\mathcal{F}_{\text{A-MC}}$ and return whatever is returned to $\mathcal{A}$.

procedure SIMULATEMINING(P, τ)
    Simulate the mining procedure of P of the protocol:
    if time-tick τ corresponds to a working sub-round **then**
        Execute Step 2 of the mining protocol. This includes:
            -Define the next state block $\mathtt{st}$ using the transaction set Txs_P
            -Send (SUBMIT-NEW, $\mathsf{sid}, \vec{\mathtt{st}}_P, \mathtt{st}$) to simulated functionality $\mathcal{F}_{\text{StX-ED}}$.
            -If successful, store $\vec{\mathtt{st}}_P || \mathtt{st}$ as the new $\vec{\mathtt{st}}_P$
            -If successful, distribute the new state via $\mathcal{F}_{\text{StX-ED}}$.
    else if time-tick τ corresponds to an update sub-round **then**
        Execute Step 4 of the mining protocol. This means that if the new information has not been fetched in this round already, then the following is executed:
            -Fetch transactions $(\mathtt{tx}_1, \ldots, \mathtt{tx}_u)$ (on behalf of P) from simulated $\mathcal{F}_{\text{A-MC}}$ and add them to Txs_P.
            -Fetch states $\vec{\mathtt{st}}_1, \ldots, \vec{\mathtt{st}}_s$ (on behalf of P) from the simulated $\mathcal{F}_{\text{StX-ED}}$ and update $\vec{\mathtt{st}}_P$ to the largest state among $\vec{\mathtt{st}}_P$ and $\vec{\mathtt{st}}_i$.
    end if
end procedure

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 state of the ledger
if $|\mathsf{state}| > |\vec{\mathtt{st}}^{\lceil T}|$ **then**
Execute AdjustVisibility(state)
end if
if state is not a prefix of $\vec{\mathtt{st}}^{\lceil T}$ **then**
Abort the simulation (due to inconsistency)
end if
Define the difference diff to be the block sequence s.t. $\mathsf{state}||\mathsf{diff} = \vec{\mathtt{st}}^{\lceil T}$.
Let $n \leftarrow |\mathsf{diff}|$
for each block diff_j, $j = 1$ to n **do**
Map each transaction $\mathtt{x}$ in this block to its unique transaction ID $\mathtt{txid}$
If a transaction does not yet have an $\mathtt{txid}$, then submit it to the ledger and receive the corresponding $\mathtt{txid}$ from $\mathcal{G}^{B}_{\text{LEDGER}}$
Let $\mathsf{list}_j = (\mathtt{txid}_{j,1}, \ldots, \mathtt{txid}_{j,\ell_j})$ be the corresponding list for this block
end for
Output (Next-Block, $(\mathsf{list}_1, \ldots, \mathsf{list}_n)$) to $\mathcal{G}^{B}_{\text{LEDGER}}$.
Execute AdjustVisibility($\mathsf{state}||\mathsf{diff}$)
end procedure

procedure AdjustVisibility(state, τ)
$\mathsf{pointers} \leftarrow \varepsilon$
for each honest and synchronized party p_i **do**
Using the simulated functionality $\mathcal{F}_{\text{StX-ed}}$ 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)
end if
if Party p_i has not executed step 4 of the mining protocol in this current mini-round **then**
$\mathsf{pointers} \leftarrow \mathsf{pointers}||(p_i, \mathtt{pt}_i)$
end if // As otherwise, the new state is only fetched in the next round
end for
Output (set-slack, $\mathsf{pointers}$) to $\mathcal{G}^{B}_{\text{LEDGER}}$
$\mathsf{pointers} \leftarrow \varepsilon$
$\mathsf{desyncStates} \leftarrow \varepsilon$
for each honest but de-synchronized party p_i **do**
Using the simulated functionality $\mathcal{F}_{\text{StX-ed}}$ do the following:
- Let $\vec{\mathtt{st}}$ be the longest state among $\vec{\mathtt{st}}_{p_i}$ and those contained in the receiver buffer $\vec{M}_{p_i}$ with delay 1

if Party p_i has not executed step 4 of the mining protocol in this current mini-round **then**
Set the pointer $\mathtt{pt}_i$ to be $|\vec{\mathtt{st}}|$
$\mathsf{pointers} \leftarrow \mathsf{pointers}||(p_i, \mathtt{pt}_i)$
$\mathsf{desyncStates} \leftarrow \mathsf{desyncState}||(p_i, \vec{\mathtt{st}})$
end if // As otherwise, the new state is only fetched in the next round
Output (set-slack, $\mathsf{pointers}$) to $\mathcal{G}^{B}_{\text{LEDGER}}$
Output (desync-state, $\mathsf{desyncStates}$) to $\mathcal{G}^{B}_{\text{LEDGER}}$
end for
end procedure

E Implementing a Stronger Ledger (Cont'd)

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

Looking ahead, the goal of this is the following: Assume that for the validation predicate $\mathsf{ValidTx}_{BC}$ it holds that if a transaction $(\mathsf{AccountID}, \mathtt{x})$ is valid relative to a state, then the only reason why it can get invalid is due to the presence of another transaction with the same account. If we think of wallets, if a miner can spend his coins at current time, then only another transaction by himself can invalidate that (by spending the same coins, which the Bitcoin network will refuse). In combination with the unforgeability of signatures, no adversary can ever render a valid transaction invalid. Due to the guarantee of the liveness guarantee in Bitcoin, that guarantees that if a transaction too old, but valid relative to the state, then it will enter the state.

We now show how to implement this account management in the $\mathcal{G}^{\text{B}}_{\text{LEDGER}}$ hybrid world to achieve a stronger ledger that formalizes account management in an ideal manner. Our protocol makes use of an existentially unforgeable digital signatures scheme $\text{DSS} := (K, S, V)$.

Definition 3. *A* digital signature scheme $\text{DSS} := (K, S, V)$ *for a message space $\mathcal{M}$ and signature space Ω consists of a (probabilistic) key generation algorithm K that returns a key pair (sk, vk), a (possibly probabilistic) signing algorithm S, that given a message $m \in \mathcal{M}$ and the signing key sk returns a signature $s \leftarrow S_{sk}(m)$, and a (possibly probabilistic, but usually deterministic) verification algorithm V, that given a message $m \in \mathcal{M}$, a candidate signature $s' \in \Omega$, and the verification key vk returns a bit $V_{vk}(m, s')$. The bit 1 is interpreted as a successful verification and 0 as a failed verification. It is required that $V_{vk}(m, S_{sk}(m)) = 1$ for all m and all (vk, sk) in the support of K. We call a* DSS *secure if it existentially unforgeable under chosen message attacks.*

A digital signatures scheme is existentially unforgeable under chosen message attacks if no efficient adversary A can win the following game $\mathbf{G}^{\mathsf{EU-CMA}}$ better than with negligible probability. $\mathbf{G}^{\mathsf{EU-CMA}}$ first chooses a key pair $(sk, vk) \leftarrow K$. Then it acts as a signing oracle, receiving messages $m \in \mathcal{M}$ at its interface and responding with $S_{sk}(m)$. At any point, A can undertake a forging attempt by providing a message m' and a candidate signature s' to $\mathbf{G}^{\mathsf{EU-CMA}}$. The game is won if and only if $V_{vk}(m', s') = 1$ and m' was never queried before by A.

E.1 The protocol for Account Management

Hybrid ledger functionality Let $\mathsf{ValidTx}_{BC}$ and $\mathsf{blockify}_{BC}$ be as in the previous section but with the following additional property: each transaction is a pair $\mathtt{x} = (\mathsf{AccountID}, \mathtt{x}')$ where the first part is bitstring of fixed length and the second part is an arbitrary transaction. In addition we require the following property: for any state state and any transaction $\mathtt{x}$ it holds that $\mathsf{ValidTx}_{BC}(\mathtt{x}, \mathsf{state}) = 1$ implies, for any state extension $\mathsf{state}||\mathtt{st}'$, that $\mathsf{ValidTx}_{BC}(\mathtt{x}, \mathsf{state}||\mathtt{st}') = 1$, if $\mathtt{st}'$ does not contain a transaction with the same identifier $\mathsf{AccountID}$. Recall that we assume that Definition 2 is satisfied.

We assume the Bitcoin ledger functionality with the following validation predicate, which is defined relative to a collision-resistant hash function H, and a signature scheme DSS.

Algorithm to describe the assumed validation predicate

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

The protocol. The protocol is straightforward: whenever the protocol is given an input of the form $(\mathsf{AccountID}, \mathtt{tx})$ it first checks that it is the party with this account 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 $\mathsf{accountMgmt}(p)$

Initialization:

This protocol talks to the $\mathcal{G}_{\text{LEDGER}}$, but only changes the behavior of on 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 $\mathcal{E}$. The protocol keeps a vector $\mathsf{submitted}$ of inputs submitted to the ledger which are not yet contained in the state of the ledger.

Account Management:

- Upon receiving $(\text{CREATEACCOUNT}, \mathsf{sid})$, execute $(sk, vk) \leftarrow K$. Set $\mathsf{AccountID} \leftarrow H(vk)$ and Return $(\text{CREATEACCOUNT}, \mathsf{sid}, \mathsf{AccountID})$

Ledger Read and Write:

- Upon receiving $(\text{SUBMIT}, \mathsf{sid}, \mathtt{x})$, check that $\mathtt{x} = (\mathsf{AccountID}, \mathtt{x}')$ for the generated account. If the check fails, ignore the input. Otherwise, do the following:
  1. Read the state state from $\mathcal{G}_{\text{LEDGER}}$
  2. If $\mathsf{ValidTx}_{BC}(\mathtt{x}, \mathsf{state}||\mathtt{st}') = 1$, then sign the input by $\sigma \leftarrow S_{sk}(\mathtt{x})$ and send $(\text{SUBMIT}, \mathsf{sid}, (\mathtt{x}, vk, \sigma))$
- Upon receiving $(\text{READ}, \mathsf{sid})$ send $(\text{READ}, \mathsf{sid})$ to $\mathcal{G}_{\text{LEDGER}}$ and receive as answer the current $\mathsf{state} = \mathtt{st}_1|| \dots ||\mathtt{st}_n$. Then do the following:

  $\mathsf{state}' \leftarrow \mathtt{st}_1$ // Genesis state
  for $i = 2$ to n **do**
    From state block $\mathtt{st}_i$, extract $(\mathtt{x}_1, vk_1, \sigma_1)|| \dots ||(\mathtt{x}_n, vk_n, \sigma_n)$
    From state block $\mathtt{st}_i$, extract time-stamp τ_i (as encoded by $\mathsf{Blockify}$)
    $\vec{x}' \leftarrow \mathtt{x}_1|| \dots ||\mathtt{x}_n$
    $\mathsf{state}' \leftarrow \mathsf{state}||\mathsf{Blockify}(\vec{x}, \tau_i)$
  end for
  Return $(\text{READ}, \mathsf{sid}, \mathsf{state}')$

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. In particular, we get the following enhanced functionality:

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

$\mathcal{G}_{\text{LEDGER}}^{\text{B}+}$ is identical to $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$ except with the following additional capabilities:

Difference to standard Ledger:

- Upon receiving $(\textsc{CreateAccount}, \mathsf{sid})$ from party p_i, send $(\textsc{AccountReq}, \mathsf{sid}, p_i)$ to $\mathcal{A}$ and upon receiving a reply $(\textsc{AccountReq}, \mathsf{sid}, p_i, \mathsf{AccountID})$ do the following:
  1. If AccountID is not yet associated to any party, store the pair $(p_i, \mathsf{AccountID})$ internally and return $(\textsc{CreateAccount}, \mathsf{sid}, \mathsf{AccountID})$ to P_i.
  2. If AccountID is already associated to a party, then output $(\textsc{CreateAccount}, \mathsf{sid}, \text{Fail})$ to p_i.

Standard Bitcoin Ledger:

- Identical to $\mathcal{G}_{\text{LEDGER}}$ with validation predicate $\mathsf{Val}_{\text{strong}}$ and thus omitted from this description.

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

Algorithm to define the strong validation

```
function Val_strong(BTX, state, buffer)
    Let BTX = (x, txid, τ_L, p_i)
    if x = (AccountID, x′) and AccountID is associated with p_i then
        return Validate(x, state)
    else
        return 0
    end if
end function
```

The stronger guarantee for honestly submitted transactions stems from two facts.

1. By Definition 2, the state blocks contain transactions beyond coin-base transactions.
2. Since a transaction of a party is associated with its account, and cannot be invalidated by another transaction with a different account, this implies that the transaction remains valid relative to state. Hence, if a honest party submits a transaction that is valid according to state, by definition of `ExtendPolicy`, this transaction is guaranteed to enter the state after staying in the the buffer for more than `Delay` rounds and an honest party mines the block after this delay, as enforced by `ExtendPolicy`. Overall, this means that after `Delay` rounds, the transaction has to appear within the next window of `slack` blocks that form the head of the state.

Looking at the ledger abstraction, we can directly compute the following worst-case upper bound for any miner: after submitting the transaction, the transaction will appear within the next $3 \cdot \texttt{slack}$ blocks after submitting the transaction. The reason is that upon submitting, the miner submitting the transaction actually be `slack` blocks behind the current state ledger, at most `slack` blocks are added after the transaction has been in the buffer for more than `Delay` time, during which no more than `slack` blocks are mined. Recall that in the real Bitcoin, the expected time until a transaction appears in the confirmed part of the state, i.e., in any block which is $T = 6$ blocks behind the head of the state, is approximately one hour. By composition, the correspondence of the ledger parameter `slack` with the protocol parameter T (as proven in the previous section), our worst-case bound suggests that three hours is the upper bound for Bitcoin (except with negligible probability) for the case that transactions are correctly signed and are not invalidated due to other transactions with the same account. The following lemma following along the ways of the above argument. The formal proof will be included in the full version.

Lemma 4. *Let* DSS *be a secure digital signature scheme and let H be a collision resistant hash function. Then the protocol* accountMgmt *in the $\mathcal{G}_{\text{LEDGER}}^{\text{B}}$-hybrid world UC-realizes ledger $\mathcal{G}_{\text{LEDGER}}^{\text{B}+}$, where the functionalities are instantiated as described above.*