Bitcoin as a Transaction Ledger: A Composable Treatment

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

[1] ETH Zurich, Zurich, Switzerland
{christian.badertscher,maurer,tschudid}@inf.ethz.ch
[2] RPI, Troy, USA
vzikas@cs.rpi.edu

Abstract. Bitcoin is one of the most prominent examples of a distributed cryptographic protocol that is extensively used in reality. Nonetheless, existing security proofs are property-based, and as such they do not support composition.

In this work we put forth a universally composable treatment of the Bitcoin protocol. We specify the goal that Bitcoin aims to achieve as a ledger functionality 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 [4,14,15,18,30,33,34]. However, a core question remained:

What security goal does Bitcoin achieve 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 [24–26,31] is that it does not rely on having a known (permissioned) set of participants, but everyone can join and leave at any point in time. This is often referred to as the *permissionless* model. Consensus in this model is achieved by shifting from the traditional assumptions on the fraction of cheating versus honest participants, to assumptions on the collective computing power of the

The full version of this paper can be found at the Cryptology ePrint Archive [6].

V. Zikas—Research supported in part by IOHK.

© International Association for Cryptologic Research 2017
J. Katz and H. Shacham (Eds.): CRYPTO 2017, Part I, LNCS 10401, pp. 324–356, 2017.
DOI: 10.1007/978-3-319-63688-7_11

cheating participants compared to the total computing power of the parties that support the consensus mechanism. The core idea is that in order for a party's action to affect the system's behavior, it needs to prove that it is investing sufficient computing resources. In Bitcoin, these resources are measured by means of solutions to a presumably computation-intensive problem.

Although the above idea is implicit in [28], a formal description of Bitcoin's goal had not been proposed or known to be achieved (and under what assumptions) until the recent works of Garay et al. [16] and Pass et al. [29]. In a nutshell, these works set forth models of computation and, in these models, an abstraction of Bitcoin as a distributed protocol, and proved that the output of this protocol satisfies certain security properties, for example the *common prefix* [16] 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 into the blockchain of every honest party and will never be reversed.[1] In addition to the common prefix property, other quality properties of the output of the abstracted blockchain protocol were also defined and proved. A more detailed description of the security properties in [16,29] is included in Sect. 4.4.

Bitcoin as a Service for Cryptographic Protocols. 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 still being researched, and certain rational models predict it is not a stable solution, it is a fact that Bitcoin has not met any of these pessimistic predictions for several years and it is not clear it ever will do. And even if it does, the research community has produced and is testing several alternative decentralized cryptocurrencies, e.g., [7,9,27], 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 one can use this new reality to improve the security and/or efficiency of cryptographic protocols? First answers to this question were given in [1–3,8,20–23] 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 multiparty computation. But in order to formally define and prove the security of the above constructions in a widely accepted cryptographic framework for multiparty protocols, one needs to define what it means for these protocols to be run in a world that gives them access to the Bitcoin network as a resource to improve their security. In other words, the question now becomes:

What functionality can Bitcoin provide to cryptographic protocols?

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

To address this question, Bentov and Kumaresan [8] introduced a model of computation in which protocols can use a punishment mechanism to incentivize adversaries to adhere to their protocol instructions. As a basis, they use the universal composition framework of Canetti [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 that model.

In a different direction, Kiayias et al. [19] connected the above question with the original question of Bitcoin's security goal. More concretely, they proposed identifying the resource that Bitcoin (or other decentralized cryptocurrencies) offers to cryptographic protocols as its security goal, and expressing it in a standard language compatible with the existing literature on cryptographic multi-party protocols. More specifically, they modeled the ideal guarantees as a transaction-ledger functionality in the universal composition framework. To be more precise, 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 party which keeps a state of blocks of transactions and makes it available, upon request, to any party. Furthermore, it accepts 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. As proved in [19], giving multi-party protocols access to such a transaction-ledger functionality allows for formally capturing, within the composable (G)UC framework, the mechanism of leveraging security loss with coins. The proposed ledger functionality guarantees all properties that one could expect from Bitcoin and encompasses the properties in [16,29]. Therefore, it is natural to postulate that it is a candidate for defining the security goal of Bitcoin (and potentially other decentralized cryptocurrencies). However, the ledger functionality proposed by [19] was not accompanied by a security proof that any of the known cryptocurrencies implements it.

However, as we show, despite being a step in the right direction, the ledger proposed in [19] cannot be realized under standard assumptions about the Bitcoin network. On the positive side, we specify a new transaction ledger functionality which still guarantees all properties postulated in [16,29], and prove that a reasonable abstraction of the Bitcoin protocol implements this ledger. In our construction, we describe Bitcoin as a UC protocol which generalizes both protocols proposed in [16,29]. Along the way we devise a compound way of capturing in UC assumptions like the ones in [16,29], which enables us to compare the strengths of these models.

Related Literature. The security of Bitcoin as a cryptographic protocol was previously studied by Garay et al. [16] and by Pass et al. [29] who proposed and analyzed an abstraction of the core of the Bitcoin protocol in a *property-based* manner. The treatment of [16,29] does not offer composable security guarantees. More recently, Kiayias et al. [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]. However, the proposed ledger-functionality is too strong to be implementable by Bitcoin. We refer the interested reader to the full version [6] for the basic elements of these works and a discussion on simulation-based security in general. A formal comparison of our treatment with [16,29], which indicates how both these protocols and definitions can be captured as special cases of our security definition, is given in Sect. 4.4.

Our Results. We put forth the first universally composable (simulation-based) proof of security of Bitcoin in the (G)UC model of Canetti et al. [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 shares certain design properties with the proposal in [19] but which can be provably implemented by the Bitcoin protocol. As in [19], our proposed functionality can be used as a global setup to allow 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. The ledger is parametrized by a generic transaction validation predicate which enables it to capture decentralized blockchain protocols beyond Bitcoin. Our functionality allows for parties/miners to join and or leave the computation and allows for adaptive corruption.

Having defined our ledger functionality we next prove that for an appropriate validation predicate $\mathcal{G}_{\text{LEDGER}}$ is implemented by Bitcoin assuming that miners which deviate from the Bitcoin protocol do not control a majority of the total hashing power at any point. To this end, we describe an abstraction of the Bitcoin protocol as a synchronous UC protocol. Our protocol generalizes both [16,29]—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 on the network delay—and the assumption on the number of queries per round. To capture these assumptions in UC, we devise a methodology to formulate functionality wrappers to capture assumptions, and discuss the implications of such a method in preserving universal composability.

Our protocol works over a network of bounded-delay channels, where similar to [29], the miners are not aware of (an upper bound on) the actual delay that the network induces. We argue that such a network is strictly weaker than a network with known bounded delay, which is implicit in the synchrony assumptions of [16] (cf. Remark 1). Notwithstanding, unlike previous works, instead of starting from a complete network that offers multicast, we explain how such a network could be

implemented by running the message-diffusion mechanism of the Bitcoin network (which is run over a lower level network of unicast channels). Intuitively, this network is built by every miner, upon joining the system, choosing some existing miners of its choice to use them as relay-nodes.

Our security proof proposes a useful modularization of the Bitcoin protocol. Concretely, we first identify the part of the Bitcoin code which intuitively corresponds to the lottery aspect, provide an ideal UC functionality that reflects this lottery aspect, and prove that this part of the Bitcoin code realizes the proposed functionality. We then analyze the remainder of the protocol in the simpler world where the respective code that implements the lottery aspect is replaced by invocations of the corresponding functionality. Using the composition theorem, we can then immediately combine the two parts into a proof of the full protocol.

Similarly to the *backbone* protocol from [16] our above UC protocol description of Bitcoin relies only on proofs of work and not on digital signatures. As a result, it implements a somewhat weaker ledger, which does not guarantee that transactions submitted by honest parties will eventually make it into the blockchain.[2] As a last result, we show that (similarly to [16]) by incorporating public-key cryptography, i.e., taking signatures into account in the validation predicate, we can implement a stronger ledger that ensures that transactions issued by honest users—i.e., users who do not sign contradicting transactions and who keep their signing keys for themselves—are guaranteed to be eventually included into the blockchain. The fact that our protocol is described in UC makes this a straight-forward, modular construction using the proposed transaction ledger as a hybrid. In particular, we do not need to consider the specifics of the Bitcoin protocol in the proof of this step. This also allows us to identify the maximum (worst-case) delay a user needs to wait before being guaranteed to see its transaction on the blockchain and be assured that it will not be inverted.

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 the implicit assumptions from previous works. As we lay down the theoretical framework, we will also discuss these assumptions along with their strengths and differences.

Bitcoin miners are represented as players—formally Interactive Turing Machine instances (ITIs)—in a multi-party computation. They 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 hybrid functionalities. Before we provide the formal specification of such functionalities, we

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

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 subject to change throughout the lifecycle of the execution. Nonetheless, UC does provide support for functionalities in which the set of parties that might interact with the functionality is dynamic. This dynamic nature is an inherent feature 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:[3]

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

For simplicity in the description of the functionalities, for a party $p_i \in \mathcal{P}$ we will use p_i to refer to this party's ID.

In addition to the above registration instructions, global setups, i.e., shared functionalities that are available both in the real and in the ideal world and allow parties connected to them to share state [11], allow also UC functionalities to register with them.[4] Concretely, global setups include, in addition to the above party registration instructions, two registration/de-registration instructions for functionalities:

- Upon receiving (REGISTER, sid_C) from a functionality $\mathcal{F}$, set $F := F \cup \{\mathcal{F}\}$.
- Upon receiving (DE-REGISTER, sid_C) from a functionality $\mathcal{F}$, set $F := F \setminus \{\mathcal{F}\}$.
- Upon receiving (GET-REGISTERED-F, sid_C) from the adversary $\mathcal{A}$, the functionality returns (GET-REGISTERED-F, sid_C, F) to $\mathcal{A}$.

The above three (or six 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.

[3] Note that making the set of parties dynamic means that the adversary needs to be informed about which parties are currently in the computation so that he can chose how many (and which) parties to corrupt.

[4] Although we allow no communication between functionalities, we will allow functionalities to communicate with global setups. (They can use the interface of global setups to additional honest parties, which is anyway open to the environment.)

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 existing miners—and uses its neighbors to send messages to all the miners in the Bitcoin network. This corresponds to multicasting the message[5]. This is achieved by 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 about the communication channels in the above diffusion mechanism/protocol:

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

Our formal description of communication with eventual delivery within the UC framework builds on ideas from [5,13,17]. In particular, we capture such communication by assuming for each miner $p_j \in \mathcal{P}$ a multi-use *unicast* channel $\mathcal{F}_{\text{U-CH}}$ with receiver p_j, to which any miner $p_i \in \mathcal{P}$ can connect and input 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 provide any information to 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 message-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 allowed to deliver them in an arbitrary out-of-order manner.

To ensure that the adversary cannot arbitrarily delay the delivery of messages submitted by honest parties, we use the following idea: We first turn the UC channel-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

[5] In [16] 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.

functionality an integer value T_{mid}—the *delay* for message with ID mid. This will have the effect that the channel ignores the next T_{mid} fetch attempts, and only then allows the receipt 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 the delivery of messages, we allow him to input delays more than once, which are added to the current delay amount. If the adversary wants to deliver the message in the next activation, all he needs to do is submit a negative delay. Furthermore, we allow the adversary to schedule more than one messages to be delivered in the same "fetch" command. Finally, to ensure that the adversary is able to re-order such batches of messages arbitrarily, we allow $\mathcal{A}$ to send special $(\mathtt{swap}, \mathsf{mid}, \mathsf{mid}')$ commands that have as an effect to change the order of the corresponding messages. The detailed specification of the described channels, denoted $\mathcal{F}_{\text{U-CH}}$ is provided in the full version [6]. Note that in the descriptions throughout the paper, for a vector $\vec{M}$ we denote by the symbol $||$ the operation which adds a new element to $\vec{M}$.

From Unicast to Multicast. As already mentioned, the Bitcoin protocol uses the above asynchronous-and-bounded-delay unicast network as a basis to achieve a multicast mechanism. A multicast functionality with bounded delay can be defined similarly to the above unicast channel. The main difference is that once a message is inserted it is recorded once for each possible receiver. The adversary can add delays to any subset of messages, but again for any message the cumulative delay cannot exceed Δ. He is further allowed to do partial and inconsistent multicasts, i.e., where different messages are sent to different parties. This is the main difference of such a multicast network from a broadcast network. The detailed specification of the corresponding functionality $\mathcal{F}_{\text{N-MC}}$ is similar to that of $\mathcal{F}_{\text{U-CH}}$ and is provided in the full version [6]. There we also show how the simple round-based diffusion mechanism can be used to implement a multicast mechanism from unicast channels as long as the corresponding network among honest parties stays strongly connected. (A network graph is strongly connected if there is a directed path between any two nodes in the network, where the unicast channels are seen as the directed edges from sender to receiver.)

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

Synchrony. Katz et al. [17], proposed a methodology for casting synchronous protocols in UC by assuming they have access to an ideal functionality $\mathcal{G}_{\text{CLOCK}}$, *the clock*, that allows parties to ensure that they proceed in synchronized rounds.

Informally, the idea is that the clock keeps track of a round variable whose value the parties can request by sending it (CLOCK-READ, sid_C). This value is updated only once all honest parties sent the clock a (CLOCK-UPDATE, sid_C) command.

Given such a clock, the authors of [17] describe how synchronous protocols can maintain their necessary round structure in UC: For every round ρ each party first executes all its round-ρ instructions and then sends the clock a CLOCK-UPDATE command. Subsequently, whenever activated, it sends the clock a CLOCK-READ command and does not advance to round $\rho+1$ before it sees the clocks variable being updated. This ensures that no honest party will start round $\rho+1$ before every honest party has completed round ρ. 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. The detailed specification of $\mathcal{G}_{\text{CLOCK}}$ is given in the full version [6].

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 sends the clock an update command. We capture this property in a generic manner in Definition 1.

In order to make the definition better accessible, we briefly recall the mechanics of activations in UC. In a UC protocol execution, an honest party (ITI) gets activated either by receiving an input from the environment, or by receiving a message from one of its hybrid-functionalities (or from the adversary). Any activation results in the activated ITI performing some computation on its view of the protocol and its local state and ends with either the party sending a message to some of its hybrid functionalities or sending an output to the environment, or not sending any message. In either of this case, the party loses the activation.[6]

For any given protocol execution, we define the *honest-input 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. 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-input sequence,* denoted as $\vec{\mathcal{I}}_H^T$, to be the honest-input sequence augmented with the respective clock time when an input was given. If the timed honest-input sequence of an execution is $\vec{\mathcal{I}}_H^T = ((x_1, \mathtt{pid}_1, \tau_1), \ldots, (x_m, \mathtt{pid}_m, \tau_m))$, this means that $((x_1, \mathtt{pid}_1), \ldots, (x_m, \mathtt{pid}_m))$ is the honest-input sequence corresponding to this execution, and for each $i \in [n]$, τ_i is the time of the global clock when input x_i was handed to $\mathtt{pid}_i$.

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

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 $\vec{\mathcal{I}}_H^T = ((x_1, \texttt{pid}_1, \tau_1), \ldots, (x_m, \texttt{pid}_m, \tau_m))$ is the corresponding timed honest-input sequence for this execution, then for any $i \in [m-1]$:* $\texttt{predict-time}_{\Pi}((x_1, \texttt{pid}_1, \tau_1), \ldots, (x_i, \texttt{pid}_i, \tau_i)) = \tau_{i+1}$.

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 can or cannot do. For example, a standard assumption in [16,29] is that in each round the adversary cannot do more calls to the random oracle than what the honest parties (collectively) can do. This can be captured by assuming a restricted environment and adversary which balances the amount of times that the adversary queries the random oracle. In a property-based treatment such as [16,29] 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, for example by assigning 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 $\mathcal{G}_{\text{CLOCK}}$. We refer the reader to [6] for a detailed specification of such a wrapped random-oracle functionality $\mathcal{W}^q(\mathcal{F}_{\text{RO}})$.

Remark 1 (Functionally Black-box Use of the Network (Delay)). A key difference between the models in [16,29] 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, 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 ledger functionality, denoted as $\mathcal{G}_{\text{LEDGER}}$, which can, for example, be achieved 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 with known approaches in the cryptographic literature), our ledger is also designed for this synchronous model. Nonetheless, several of our modeling choices are made with the foresight of removing or limiting the use of the clock and leaving room for less synchrony.

At a high level, our ledger $\mathcal{G}_{\text{LEDGER}}$ has a similar structure as the ledger proposed in [19]. Concretely, anyone (whether an honest miner or the adversary) might submit a transaction which is validated by means of a predicate $\mathsf{Validate}$, and if it is found valid it is added to a buffer buffer. The adversary $\mathcal{A}$ is informed that the transaction was received and is given its contents.[7] Informally, this buffer also contains transactions that, although validated, are not yet deep enough in the blockchain to be considered out-of-reach for a adversary.[8] Periodically, $\mathcal{G}_{\text{LEDGER}}$ fetches some of the transactions in the buffer, and using an algorithm $\mathsf{Blockify}$ creates a block including these transactions and adds this block to its permanent state state, which is a data structure that includes the part of the blockchain the adversary can no longer change. This corresponds to the *common prefix* in [16,29]. Any miner or the adversary is allowed to request a read of the contents of the state.

This sketched specification is simple, but in order to have a ledger that can be implemented by existing blockchain protocols, we need to relax this functionality by giving the adversary more power to interfere with it and influence its behavior. Before sketching the necessary relaxations we discuss the need for a new ledger definition and it potential use as a global setup.

Remark 2 (Impossibility to realize the ledger of [19]*).* The main reasons why the ledger in [19] is not realizable by known protocols under reasonable assumptions are as follows: first, their ledger guarantees that parties always obtain the same common state. Even with strong synchrony assumptions, this is not realizable since an adversary, who just mined a new block, is not forced to inform each party instantaneously (or at all) and thus could, e.g., make parties observe different lengths of the same prefix. Second, the adversarial influence is restricted to permuting the buffer. This is too optimistic, as in reality the adversary can try to mine a new block and possibly exclude certain transactions. Also, this excludes any possibility to quantify quality. Third, letting the update rate be fixed does not adequately reflect the probabilistic nature of blockchain protocols.

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

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

Remark 3 (On the sound usage of a ledger as a global setup). As presented in [19], a UC ledger functionality $\mathcal{G}_{\text{LEDGER}}$ can be cast as a global setup [11] which allows different protocols to share state. This is true for any UC functionality as stated in [11,12]. Nonetheless, as pointed out in the recent work of Canetti et al. [12], one needs to be extra careful when replacing a global setup by its implementation, e.g., in the case of $\mathcal{G}_{\text{LEDGER}}$ by the UC Bitcoin protocol of Sect. 4. Indeed, such a replacement does not, in general, preserve a realization proof of some ideal functionality $\mathcal{F}$ that is conducted in a ledger-hybrid world, because the simulator in that proof might rely on specific capabilities that are not available any more after replacement (as the global setup is also replaced in the ideal world). The authors of [12] provide a sufficient condition for such a replacement to be sound. This condition is generally too strong to be satisfied by any natural ledger implementation, which opens the question of devising relaxed sufficient conditions for sound replacements in an MPC context. As this work focuses on the realization of ledger functionalities per se, we can treat $\mathcal{G}_{\text{LEDGER}}$ as a standard UC functionality.

In the following, we review the necessary relaxations to obtain a realizable ledger. We conclude this section with the specification of our generic ledger functionality.

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 receipt of a transaction, Validate checks that it is not in conflict with the state and the buffer, and if so the transaction is added to the buffer. However, in reality we do not know how to implement such a strong filter, as different miners might be working on different, potentially conflicting sets of transactions. The only time when it becomes clear which of these conflicting transactions will make it into the state is once one of them has been inserted into a block which has made it deep enough into the blockchain (i.e., has become part of state). Hence, given that the buffer includes all transactions that might end up in the state, it might at some point include both conflicting transactions.

To enable us for a provably implementable ledger, in this work we take a different approach. The validate predicate will be less restrictive as to which transactions make it into the buffer. Concretely, at the very least, Validate will enforce the invariant that no single transaction in the buffer contradicts the state state, while different transactions in buffer might contradict each other. Looking ahead, a stronger version that is achievable by employing digital signatures (presented in Sect. 5), could enforce that no submitted transaction contradicts other submitted 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 re-validation to be

generic, transactions that are added to the buffer are accompanied by certain metadata, i.e., the identity of the submitter, a unique transaction ID `txid` [9], or 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 a block. Concretely, instead of assuming that the state is extended in fixed time intervals, we allow the adversary to define when this update occurs. This is done by allowing the adversary, at any point, to propose what we refer to as the next-block candidate `NxtBC`. This is a data structure containing the contents of the next block that $\mathcal{A}$ wants to have inserted into the state. Leaving `NxtBC` empty can be interpreted as the adversary signaling that it does not want the state to be updated in the current clock tick.

Of course allowing the adversary to always decide what makes it into the state `state`, or if anything ever does, yields a very weak ledger. Intuitively, this would be a ledger that only guarantees the common prefix property [16] but no liveness or chain quality. Therefore, to enable us to capture also stronger properties of blockchain protocols we parameterize the ledger by an algorithm `ExtendPolicy` that, informally, enforces a state-update policy restricting the freedom of the adversary to choose the next block and implementing an appropriate compliance-enforcing mechanism in case the adversary does not follow the policy. This enforcing mechanism simply returns a default policy-complying block using the current contents of the buffer. We point out that a good simulator for realizing the ledger will avoid triggering this compliance-enforcing mechanism, as this could result in an uncontrolled update of the state which would yield a potential distinguishing advantage. In other words, a good simulator, i.e., ideal-world adversary, always complies with the policy.

In a nutshell, `ExtendPolicy` takes the current contents of the buffer `buffer`, along with the adversary's recommendation `NxtBC`, and the *block-insertion times vector* $\vec{\tau}_{\mathtt{state}}$. The latter is a vector listing the times when each block was inserted into `state`. The output of `ExtendPolicy` is a vector including the blocks to be appended to the state during the next state-extend time-slot (where again, `ExtendPolicy` outputting an empty vector is a signal to not extend). To ensure that `ExtendPolicy` can also enforce properties that depend on who inserted how many (or which) blocks into the state—e.g. the so-called *chain quality* property from [16]—we also pass to it the timed honest-input sequence $\vec{\mathcal{I}}_H^T$ (cf. Sect. 2).

Some examples of how `ExtendPolicy` allows us to define ways that the protocol might restrict the adversary's interference in the state-update include the following properties from [16]:

– *Liveness* corresponds to `ExtendPolicy` enforcing the following policy: If the state has not been extended for more that a certain number of rounds and the

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

simulator keeps recommending an empty `NxtBC`, `ExtendPolicy` can choose some of the transactions in the buffer (e.g., those that have been in the buffer for a long time) and add them to the next block. Note that a good ideal-world adversary will never allow for this automatic update to happen and will make sure that he keeps the state extend rate within the right amount.

– *Chain quality* corresponds to `ExtendPolicy` enforcing the following policy: `ExtendPolicy` looks into the blocks of state for a special type of transaction (corresponding to a so-called coinbase transaction) and parses the state (using the sequence of honest inputs $\vec{\mathcal{I}}_H^T$ and the block-insertion times vector $\vec{\tau}_{\mathsf{state}}$) 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 `ExtendPolicy`), then `ExtendPolicy` forces the coinbase transaction of the next block to have as the miner ID the ID submitted by some honest miner.

In addition to the above standard properties, `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 Sect. 4 where we prove the security of Bitcoin, we will provide the concrete specification of Validate and `ExtendPolicy` for which the Bitcoin protocol realizes our ledger.

Output Slackness and Sliding Window of State Blocks. The common prefix property guarantees that blocks which are sufficiently deep in the blockchain of an honest miner will eventually be included in the blockchain of every honest miner. Stated differently, if an honest miner receives as output from the ledger a state state, every honest miner will eventually receive state as its output. However, in reality we cannot guarantee that at any given point in time all honest miners see exactly the same blockchain length; this is especially the case when network delays are incorporated into the model, but it is also true in the zero-delay model of [16]. 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 a 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 allows the adversary to only move the pointers forward and it forbids the adversary to define pointers for honest miners that are too far apart, i.e., more than `windowSize` state blocks. The parameter $\mathtt{windowSize} \in \mathbb{N}$ denotes a core parameter of the ledger. In particular, the parameter `windowSize` reflects the similarity of the blockchain to the dynamics of a so-called *sliding window*, where the window of size `windowSize`

contains the possible views of honest miners onto state and where the head of the window advances with the head of the state. In addition, it is convenient to express security properties of concrete blockchain protocols, including the properties discussed above, as assertions that hold within such a sliding window (for any point in time).

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 $\mathcal{G}_{\text{CLOCK}}$ 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 our 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 wants 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[10] (where 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 $\{1, \ldots, n\}$.

[10] For technical reasons described in Sect. 4.1, Δ rounds in the protocol correspond to 2Δ clock-ticks.

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

$\mathcal{G}_{\text{LEDGER}}$ is parametrized by four algorithms, $\mathsf{Validate}$, $\mathtt{ExtendPolicy}$, $\mathsf{Blockify}$, and $\mathtt{predict\text{-}time}_{BC}$, along with two parameters: $\mathtt{windowSize}, \mathtt{Delay} \in \mathbb{N}$. The functionality manages variables state, $\mathtt{NxtBC}$, buffer, τ_L, and $\vec{\tau}_{\mathsf{state}}$, as described above. The variables are initialized as follows: $\mathsf{state} := \vec{\tau}_{\mathsf{state}} := \mathtt{NxtBC} := \varepsilon$, $\mathsf{buffer} := \emptyset$, $\tau_L = 1$.

The functionality maintains the set of registered parties $\mathcal{P}$, the (sub-)set of honest parties $\mathcal{H} \subseteq \mathcal{P}$, and the (sub-set) of de-synchronized honest parties $\mathcal{P}_{DS} \subset \mathcal{H}$. The set $\mathcal{P}, \mathcal{H}, \mathcal{P}_{DS}$ are all 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 $\mathcal{P}_{DS}$.

For each party $p_i \in \mathcal{P}$ the functionality maintains a pointer $\mathtt{pt}_i$ (initially set to 1) and a current-state view $\mathsf{state}_i := \varepsilon$ (initially set to empty). The functionality also keeps track of the timed honest-input sequence in a vector $\vec{\mathcal{I}}_H^T$ (initially $\vec{\mathcal{I}}_H^T := \varepsilon$).

Upon receiving any input I from any party or from the adversary, send $(\text{CLOCK-READ}, \mathsf{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$ and upon receiving response $(\text{CLOCK-READ}, \mathsf{sid}_C, \tau)$ 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 $\vec{N} = (\vec{N}_1, \ldots, \vec{N}_\ell) := \mathtt{ExtendPolicy}(\vec{\mathcal{I}}_H^T, \mathsf{state}, \mathtt{NxtBC}, \mathsf{buffer}, \vec{\tau}_{\mathsf{state}})$ and if $\vec{N} \neq \varepsilon$ set $\mathsf{state} := \mathsf{state} || \mathsf{Blockify}(\vec{N}_1) || \ldots || \mathsf{Blockify}(\vec{N}_\ell)$ and $\vec{\tau}_{\mathsf{state}} := \vec{\tau}_{\mathsf{state}} || \tau_L^\ell$, where $\tau_L^\ell = \tau_L || \ldots, || \tau_L$.
   (c) For each $\mathtt{BTX} \in \mathsf{buffer}$: if $\mathsf{Validate}(\mathtt{BTX}, \mathsf{state}, \mathsf{buffer}) = 0$ then delete $\mathtt{BTX}$ from buffer.
   (d) If there exists $j \in [\ell]$ with $p_{i_j} \in \mathcal{H} \setminus \mathcal{P}_{DS} : |\mathsf{state}| - \hat{\mathtt{pt}}_i \leq \mathtt{windowSize}$ or $\hat{\mathtt{pt}}_{i_j} \geq |\mathsf{state}_{i_j}|$, then for every $j \in [\ell]$ set $\mathtt{pt}_i := |\mathsf{state}| - |\vec{\mathbf{N}}|$ 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:
     (a) Choose a unique transaction ID $\mathtt{txid}$ and set $\mathtt{BTX} := (x, \mathtt{txid}, \tau_L, p_i)$
     (b) If $\mathsf{Validate}(\mathtt{BTX}, \mathsf{state}, \mathsf{buffer}) = 1$, then $\mathsf{buffer} := \mathsf{buffer} \cup \{\mathtt{BTX}\}$.
     (c) 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) $\texttt{predict-time}(\vec{\mathcal{I}}_H^T) = \hat{\tau} > \tau_L$ then send $(\text{CLOCK-UPDATE}, \mathsf{sid}_C)$ to $\mathcal{G}_{\text{CLOCK}}$. Else send I to $\mathcal{A}$.
- *The adversary proposing the next block:*
  If $I = (\text{NEXT-BLOCK}, (\texttt{txid}_1, \ldots, \texttt{txid}_\ell))$ is sent from the adversary, update $\texttt{NxtBC}$ as follows:
  (a) Set $\texttt{NxtBC} := \varepsilon$.
  (b) For $i = 1, \ldots, \ell$ do: if there exists $\texttt{BTX} := (x, \texttt{txid}, \mathsf{minerID}, \tau_L, p_i) \in \mathsf{buffer}$ with ID $\texttt{txid} = \texttt{txid}_i$ then set $\texttt{NxtBC} := \texttt{NxtBC}||\texttt{txid}_i$.
  (c) Output $(\text{NEXT-BLOCK}, ok)$ to $\mathcal{A}$.
- *The adversary setting state-slackness:*
  If $I = (\text{SET-SLACK}, (p_{i_1}, \hat{\texttt{pt}}_{i_1}), \ldots, (p_{i_\ell}, \hat{\texttt{pt}}_{i_\ell}))$, with $\{p_{i_1}, \ldots, p_{i_\ell}\} \subseteq \mathcal{H} \setminus \mathcal{P}_{DS}$ is received from the adversary $\mathcal{A}$ do the following:
  (a) If for all $j \in [\ell]$: $|\mathsf{state}| - \hat{\texttt{pt}}_{i_j} \leq \texttt{windowSize}$ and $\hat{\texttt{pt}}_{i_j} \geq |\mathsf{state}_{i_j}|$, set $\texttt{pt}_{i_1} := \hat{\texttt{pt}}_{i_1}$ for every $j \in [\ell]$ and return $(\text{SET-SLACK}, ok)$ to $\mathcal{A}$.
  (b) Otherwise set $\texttt{pt}_{i_j} := |\mathsf{state}|$ 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 realizes an instantiation of the ledger functionality from the previous section. More concretely, we cast the Bitcoin protocol as a UC protocol, where consistent with 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 first describe the UC protocol, denoted Ledger-Protocol, in Sect. 4.1 which abstracts the components of Bitcoin that are relevant for the construction of such a ledger—similar to how the backbone protocol [16] captures core Bitcoin properties in their respective model of computation. Later, in Sect. 4.2, we specify the ledger functionality $\mathcal{G}_{\text{LEDGER}}^{\mathbb{B}}$ 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 $\texttt{ExtendPolicy}$. In fact, for the sake of generality, we specify generic classes of Validate and Blockify and parameterize our Ledger-Protocol with these classes, so that the security statement still stays generic. We then prove our main theorem (Theorem 1) which can be described informally as follows:

Theorem (*informal*)**.** *Let 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* $\texttt{windowSize} = \omega(\log \kappa)$*, κ being the length of the outputs of the random oracle. Then, for an appropriate* $\texttt{ExtendPolicy}$ *and for any function* Blockify*, the protocol* Ledger-Protocol *instantiated with algorithms* Validate

and Blockify *securely realizes a ledger functionality* $\mathcal{G}_{\text{LEDGER}}^{\mathbb{P}}$ *(the generic ledger instantiated with the above functions) 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 be formally defined later):*

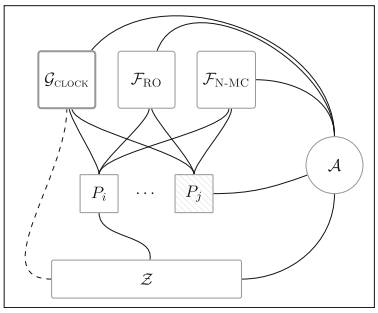
- *In any round of the protocol execution, the collective mining power of the adversary, contributed by corrupted and temporarily de-synchronized miners, does not exceed the mining power of honest (and synchronized) parties in that round. The exact relation additionally captures the (negative) impact 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 (cf. Fig. 1). Informally, this modularization distills out form the protocol a reactive *state-extend* subprocess which captures the lottery that decides which miner gets to advance the blockchain next and additionally the process of propagating this state to other miners. Lemma 1 shows that the state-extend module/subprocess implements an appropriate reactive UC functionality $\mathcal{F}_{\text{STX}}$. We can then use the UC composition theorem which allows us to argue security of Ledger-Protocol in a simpler hybrid world where, instead of using this subprocess, parties make calls to the functionality $\mathcal{F}_{\text{STX}}$. We conclude this section (Subsect. 4.4) by showing how both the GKL and PSs protocols can be cast as special cases of our protocol which provides the basis for comparing the different models and their respective assumptions.

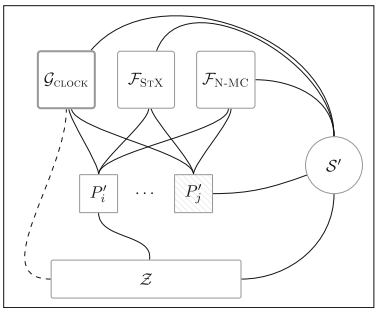
4.1 The Bitcoin Ledger as a UC Protocol

In the following we provide the formal description of protocol Ledger-Protocol. The protocol assumes as hybrids the multi-cast network $\mathcal{F}_{\text{N-MC}}$ (recall that we assume that this network does have an upper bound Δ on the delay unknown to the protocol) 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 use throughout this section. For compatibility with existing work, wherever it does not overload notation, we use some of the terminology and notation from [16].

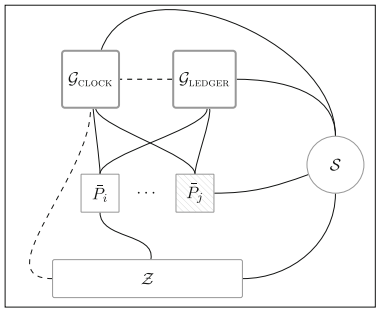
Blockchain. A *blockchain* $\mathcal{C} = \mathbf{B}_1, \ldots, \mathbf{B}_n$ is a (finite) sequence of blocks where each *block* $\mathbf{B}_i = \langle \mathsf{s}_i, \mathtt{st}_i, \mathsf{n}_i \rangle$ is a triple consisting of the *pointer* s_i, the *state block* $\mathtt{st}_i$, and the *nonce* n_i. A special block is the *genesis block* $\mathbf{G} = \langle \bot, \mathtt{gen}, \bot \rangle$ which contains the genesis state $\mathtt{gen}$. 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* $\vec{\mathtt{st}}$ corresponding to $\mathcal{C}$ is defined as a sequence of the corresponding state blocks, i.e., $\vec{\mathtt{st}} := \mathtt{st}_1 || \ldots || \mathtt{st}_n$. In other words, one should think of the blockchain $\mathcal{C}$ as an encoding of its underlying state $\vec{\mathtt{st}}$; such an encoding might, e.g., organize $\mathcal{C}$ is an efficient searchable data structure as is



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



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



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

Fig. 1. Modularization of the Bitcoin protocol.

the case in the Bitcoin protocol where a blockchain is a linked list implemented with hash-pointers.

In the protocol, the blockchain is the data structure storing a sequence of entries, often referred to as transactions. Furthermore, as in [19], in order to capture blockchains with syntactically different state encoding, we use an algorithm $\mathsf{blockify}_{\mathbb{P}}$ to map a vector of transactions into a state block. Thus, each block $\mathtt{st} \in \vec{\mathtt{st}}$ (except the genesis state) of the state encoded in the blockchain has the form $\mathtt{st} = \mathsf{Blockify}(\vec{N})$ 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 $\mathbf{B}_i = \langle \mathsf{s}_i, \mathtt{st}_i, \mathsf{n}_i \rangle$ 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 $\mathsf{D} \in [\kappa]$, where

κ is the security parameter, and a given hash function $H(\cdot) : \{0,1\}^* \to \{0,1\}^\kappa$; it requires that, for each $i > 1$, the value s_i contained in $\mathbf{B}_i$ satisfies $\mathsf{s}_i = H[\mathbf{B}_{i-1}]$ and that additionally $H[\mathbf{B}_i] < \mathsf{D}$ holds.

The semantic validity on the other hand is defined on the state $\vec{\mathtt{st}}$ encoded in the blockchain $\mathcal{C}$ and specifies whether this content is valid (which might depend on a particular application). The validation predicate $\mathsf{Validate}$ defined in the ledger functionality (cf. Sect. 3) plays a similar role. In fact, the semantic validity of the blockchain can be defined using an algorithm that we denote $\mathsf{isvalidstate}$ which is builds upon the $\mathsf{Validate}$ predicate. The idea is that for any choice of $\mathsf{Validate}$, the blockchain protocol using $\mathsf{isvalidstate}$ for semantic validation of the chain implements the ledger parametrized with $\mathsf{Validate}$. More specifically, algorithm $\mathsf{isvalidstate}$ checks that a given blockchain state can be built in an iterative manner, such that each contained transaction is considered valid according to $\mathsf{Validate}$ upon insertion. It further ensures that the state starts with the genesis state and that state blocks contain a special *coin-base* transaction $\mathtt{x}^{\mathsf{coin\text{-}base}}_{\mathsf{minerID}}$ which assigns them to a miner. We remark that this only works for predicates $\mathsf{Validate}$ which ignore all information other than the state and transaction that is being validated.[11] To avoid confusion, throughout this section we use $\mathsf{ValidTx}_{\mathbb{B}}$ to refer to the validate predicate with the above restriction. The pseudo-code of the algorithm $\mathsf{isvalidstate}$ which builds upon $\mathsf{ValidTx}_{\mathbb{B}}$ is provided in the full version [6]. We succinctly denote by $\mathsf{isvalidchain}_D(\mathcal{C})$ the predicate that returns true iff chain $\mathcal{C}$ satisfies syntactic and semantic validity as defined above.

The Ledger Protocol. We are now ready to formally define our blockchain protocol $\mathsf{Ledger\text{-}Protocol}_{q,\mathsf{D},T}$ (we usually omit the parameters when clear from the context). The protocol allows an arbitrary number of parties/miners to communicate by means of a multicast network $\mathcal{F}_{\text{N-MC}}$. Note that this means that the adversary can send different messages to different parties. New miners might dynamically joint or leave the protocol by means of the registration/de-registration commands: when they join they register with all associated functionalities and when they leave they deregister.[12]

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 $\mathsf{ValidTx}_{\mathbb{B}}$) and creates a new state block $\mathtt{st}$ using $\mathsf{blockify}_{\mathbb{B}}$. 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 as inputs a chain $\mathcal{C}$, a state block $\mathtt{st}$, and the number q of attempts. The core

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

[12] Note that when a party registers to a local functionality such as the network or the random oracle it does not lose its activation token. This is a subtle point to ensure that the real and ideal worlds are in-sync regarding activations.

idea of the algorithm is to find a proof-of-work which allows to extend $\mathcal{C}$ by a block which encodes $\mathtt{st}$. The pseudo-code of this algorithm is provided in the full version [6]. After each mining attempt parties will multicast their current chain. A party will replace its local chain if it receives a longer chain. When queried to output the state of the ledger, Ledger-Protocol outputs the state of its longest chain, where it first chops-off the most recent T blocks. This behavior will ensure that all honest parties output a consistent 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 correspond to two sub-rounds (also known as mini-rounds). To avoid confusion we refer to clock rounds as *clock-ticks*. 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 two clock-ticks per round is the way to ensure in synchronous UC that messages (e.g., a block) sent within a round are delivered at the beginning of the next round. The idea is that each round is divided into 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 fetch messages from the network to obtain messages sent in the previous round. The pseudo-code of this UC blockchain protocol, denoted as Ledger-Protocol, is provided in the full version [6], 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 Sect. 3 with appropriate parameters so that it is implemented by protocol Ledger-Protocol. To define this Bitcoin ledger $\mathcal{G}_{\text{LEDGER}}^{\mathbb{B}}$, we need to give specific instantiations of the three functions Validate, Blockify, and $\mathtt{ExtendPolicy}$.

As mentioned above, in case of Validate we use the same predicate as the protocol uses to validate the states: For a given transaction $\mathtt{x}$ and a given state state, the predicate decides whether this transaction is valid with respect to state. Given such a validation predicate, the ledger validation predicate takes a specific simple form which, excludes dependency on anything other than the transaction $\mathtt{x}$ and the state state, i.e., for any values of $\mathtt{txid}, \tau_L, p_i$, and buffer:

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

Blockify can be an arbitrary algorithm, and if the same algorithm is used in Ledger-Protocol the security proof will go through. However, as discussed below (in 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 $\mathtt{ExtendPolicy}$. At a high level, upon receiving a list of possible candidate blocks which should go into the state of the ledger, $\mathtt{ExtendPolicy}$ 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, $\mathtt{ExtendPolicy}$ ensures the following property:

1. The speed of the ledger is not too fast. This is implemented by defining a bound $\texttt{minTime}_{\text{window}}$ on the time (number of rounds) within which no more than `windowSize` state blocks can be added.
2. The speed of the ledger is not too slow. This is implemented by defining a bound $\texttt{maxTime}_{\text{window}}$ within which at least `windowSize` 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 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 in conflict. As we show in Sect. 5, 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 an ideal way, and whenever it catches the adversary in trying to propose blocks that do not obey the policies, it punishes the adversary by proposing its own generated block. The formal description of the extend policy (as pseudo-code) for $\mathcal{G}_{\text{LEDGER}}^{\hat{\mathcal{B}}}$ is given in the full version [6].

On the Relation Between Blockify and Validate. As already discussed above, `ExtendPolicy` guarantees that the adversary cannot block the extension of the state indefinitely, and that occasionally an honest miner will receive the block reward (via the coin-base) transaction. These correspond to the chain-growth and chain-quality properties from [16]. 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 [16], is achieved by adding digital signatures.

To see where this could be problematic in general, consider a blockify that, at a certain point, creates a block that renders all possible future transactions invalid. Observe that this does not mean that our protocol is insecure and that this is as well possible for the protocols of [16,29]; 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 a certain relation with each other. Informally, this relation should ensure that the above sketched 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 *if there exists an efficiently computable function* Dec *mapping outputs of* Blockify *to vectors* $\vec{N}$ *such that there exists a validate predicate* Validate′ *for which the following properties hold for any possible state* $\mathsf{state} = \mathtt{st}_1||\ldots||\mathtt{st}_\ell$*, buffer* buffer*, vectors* $\vec{N} := (x_1, \ldots, x_m)$*, and transaction* x*:*

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

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 $\mathsf{ValidTx}_{\mathbb{B}}$ and $\mathsf{blockify}_{\mathbb{B}}$ 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 distill out from the protocol Ledger-Protocol a state-extend 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 $\mathcal{F}_{\text{STX}}$ that is realized by StateExchange-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 $\mathcal{F}_{\text{STX}}$. 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 $\mathcal{F}_{\text{STX}}$, implements the Bitcoin ledger described in Sect. 4.2.

Step 1. The state-exchange functionality $\mathcal{F}_{\text{STX}}$ allows parties to submit ledger states which are accepted with a certain probability. Accepted states are then multicast to all parties. Informally, it can be seen as lottery on which (valid) states are exchanged among the participants. Parties can use $\mathcal{F}_{\text{STX}}$ to multicast a valid state, but instead of accepting any submitted state and sending it to all (registered) parties, $\mathcal{F}_{\text{STX}}$ keeps track of all states that it ever saw, and implements the following mechanism upon submission of a new ledger state $\vec{\mathtt{st}}$ and a state block $\mathtt{st}$ from any party: If $\vec{\mathtt{st}}$ was previously submitted to $\mathcal{F}_{\text{STX}}$ and $\vec{\mathtt{st}}||\mathtt{st}$ is a valid state, then $\mathcal{F}_{\text{STX}}$ 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 $\mathcal{F}_{\text{LTX}}^{\Delta,p_H,p_A}$

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 $(\text{CONTINUE}, \mathsf{sid})$ where $\mathcal{P} = \{p_1, \ldots, p_n\}$ choose n new unique message-IDs $\mathsf{mid}_1, \ldots, \mathsf{mid}_n$, initialize n new variables $D_{\mathsf{mid}_1} := D_{\mathsf{mid}_1}^{MAX} := \ldots := D_{\mathsf{mid}_n} := D_{\mathsf{mid}_n}^{MAX} := 1$ set $\vec{M} := \vec{M}||(\vec{\mathtt{st}}_{new}, \mathsf{mid}_1, D_{\mathsf{mid}_1}, p_1)|| \ldots ||(\vec{\mathtt{st}}_{new}, \mathsf{mid}_n, D_{\mathsf{mid}_n}, p_n)$, and send $(\text{SUBMIT-NEW}, \mathsf{sid}, \vec{\mathtt{st}}_{new}, p_s, (p_1, \mathsf{mid}_1), \ldots, (p_n, \mathsf{mid}_n))$ to the adversary.
- Upon receiving $(\text{FETCH-NEW}, \mathsf{sid})$ 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 $(\text{SEND}, \mathsf{sid}, \vec{\mathtt{st}}, p')$ 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 $(\text{SEND}, \mathsf{sid}, \vec{\mathtt{st}}, p', \mathsf{mid})$ to $\mathcal{A}$.

Further adversarial influence on the network:

- Upon receiving $(\text{SWAP}, \mathsf{sid}, \mathsf{mid}, \mathsf{mid}')$ from $\mathcal{A}$, if mid and mid' are message-IDs registered in the current $\vec{M}$, swap the corresponding tuples in $\vec{M}$. Return $(\text{SWAP}, \mathsf{sid})$ to $\mathcal{A}$.
- Upon receiving $(\text{DELAY}, \mathsf{sid}, T, \mathsf{mid})$ from $\mathcal{A}$, if T is a valid delay, mid is a message-ID for a tuple $(\vec{\mathtt{st}}, \mathsf{mid}, D_{\mathsf{mid}}, p)$ in the current $\vec{M}$ and $D_{\mathsf{mid}}^{MAX} + T \leq \Delta$, set $D_{\mathsf{mid}} := D_{\mathsf{mid}} + T$ and set $D_{\mathsf{mid}}^{MAX} := D_{\mathsf{mid}}^{MAX} + T$.

The Modular-Ledger-Protocol uses the same hybrids as Ledger-Protocol but abstracts the lottery implemented by the mining process by making calls to

the above state exchange functionality $\mathcal{F}_{\text{StX}}^{\Delta,p_H,p_A}$. The detailed specification of the Modular-Ledger-Protocol protocol can be found in the full version [6]. Note that the only remaining parameter of Modular-Ledger-Protocol is the chop-off parameter T, the rest is part of $\mathcal{F}_{\text{StX}}^{\Delta,p_H,p_A}$. The following Lemma states that our Bitcoin protocol implements the above modular ledger protocol. The proof appears in [6].

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

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 used in both, 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 $\mathcal{F}_{\text{StX}}$ 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 interval of, say, 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 try to extend the state). 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 $\mathcal{F}_{\text{StX}}$. 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 for example quantify the overall expectation by $\mathsf{T}_{mp}^{\text{total}}(t_{rc}) := \sum_{r=1}^{t_{rc}} \mathsf{T}_{mp}(r)$. This allows to formulate the goal of a re-calibration of the difficulty parameter as requiring that this value should be 2016 blocks for t_{rc} corresponding a desired time bound (such as roughly two weeks), which is part of future work.

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.

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

Recall that a party is considered desynchronized for 2Δ rounds after its registration.

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 $\mathsf{mp}_A(r) \leq \rho \cdot \mathsf{T}_{mp}(r)$ holds. We then define $\beta_r := \rho \cdot \mathsf{T}_{mp}(r)$. 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 $\mathcal{F}_{\text{StX}}$. 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 synchronized*, uncorrupted miner successfully queries $\mathcal{F}_{\text{StX}}$ 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 flat models of Bitcoin, parties are expected to be synchronized and are otherwise counted as dishonest and the quantity $(1 - \rho)$ 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, then

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

Remark 4. 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, as the value β (as a fraction of the total mining power) could not be reasonably bounded. We are now ready to state the main theorem. The proof of the theorem can be found in the full version [6].

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

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

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* $\mathcal{G}^{\mathcal{B}}_{\text{LEDGER}}$ *for any parameters in the range*

$$\texttt{windowSize} = T \quad \text{and} \quad \texttt{Delay} = 4\Delta,$$
$$\texttt{maxTime}_{\text{window}} \geq \frac{\texttt{windowSize}}{(1-\delta)\cdot\gamma} \quad \text{and} \quad \texttt{minTime}_{\text{window}} \leq \frac{\texttt{windowSize}}{(1+\delta)\cdot \max_r \mathsf{T}_{mp}(r)},$$
$$\eta > (1+\delta)\cdot \texttt{windowSize} \cdot \frac{\beta}{\gamma},$$

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 5. It is worth noting the implications of Eq. 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 canceling out the term mp and letting p be sufficiently small (compared to $\frac{1}{\Delta m}$), Eq. 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 $p > \frac{1}{mp}$, Eq. 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 all honest (and synchronized) miners that

it simulates and determine their common prefix of states, i.e., 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. To reflect that not all parties have the same view on this common prefix, the simulator can adjust the state pointers accordingly. This simulation is perfect and corresponds to an emulation of real-world processes. What possibly prevents 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 et al. [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 [6]. □

Note that the theorem statement a-priori holds for any environment (but simply yields a void statement if the conditions are not met). In order to turn this into a composable statement, we follow the approach proposed in Sect. 2 and model restrictions as wrapper functionalities to ensure the condition of the theorem. We review two particular choices in Sect. 4.4. The general conceptual principle behind this is the following: For the hybrid world, that consists of a network $\mathcal{F}_{\text{N-MC}}$, a clock $\mathcal{G}_{\text{CLOCK}}$ and a random oracle $\mathcal{F}_{\text{RO}}$ with output length κ (or alternatively the state-exchange functionality $\mathcal{F}_{\text{STX}}$ instead of the random oracle), define a wrapper functionality $\mathcal{W}$ which ensures the condition in Eq. 1 and (possibly) additional conditions on minimal (honest) and maximal (dishonest) mining power. This can be done by enforcing appropriate restrictions along the lines of the basic example in Sect. 2 (e.g., imposing an upper bound on parties, or RO queries per round etc.). We provide the details and the specification of such a general random-oracle wrapper $\mathcal{W}^{\Delta,\lambda,\mathsf{T}_{mp}}_{\alpha,\beta,\mathsf{D}}(\mathcal{F}_{\text{RO}})$ with its parameters[13] in the full version of this work [6]. For this wrapper 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. *The UC blockchain protocol* $\mathsf{Ledger\text{-}Protocol}_{q,D,T}$ *that is executed in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{F}^{\Delta}_{\text{N-MC}}, \mathcal{W}^{\Delta,\lambda,\mathsf{T}_{mp}}_{\alpha,\beta,D}(\mathcal{F}_{\text{RO}}))$*-hybrid world, UC-realizes functionality* $\mathcal{G}^{\mathbb{D}}_{\text{LEDGER}}$ *(with the respective parameters assured by Theorem 1).*

[13] The parameters are the ones introduced in this section: a lower bound on honest mining power (per round) α, an upper bound on adversarial mining power (per round) β, the total mining power (per round) T_{mp}, the network delay Δ, the difficulty parameter D (that influences the probability of a successful PoW), and finally a value $\lambda > 1$ describing the required gap between honest and dishonest mining power.

4.4 Comparison with Existing Work

We demonstrate how the protocols, assumptions, and results from the two existing works analyzing security of Bitcoin (in a property based manner) can be cast as special cases of our construction.

We start with the result in [16], which is the so-called flat and synchronous model[14] with instant delivery and a constant number of parties n (i.e., Bitcoin is seen as an n-party MPC protocol).[15] 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 $\mathcal{F}_{\text{STX}}$ 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 $p_A = p$. We can further conclude that $\mathsf{T}_{mp}^{\text{GKL}}(r) \leq p \cdot q \cdot n$.
- The adversary gets (at most) q queries per corrupted party with probability $p_A = p$ and one query per honest but desynchronized party with success probability $p_H = 1 - (1-p)^q$. If t_r denotes the number of corrupted or desynchronized parties in round r, we get $\mathsf{mp}_A^{\text{GKL}}(r) \leq t_r \cdot q \cdot p$ and thus $\beta_r^{\text{GKL}} = p \cdot q \cdot (\rho \cdot n)$, where ρn is the (assumed) upper bound on the number of miners contributing to the adversarial mining power (independent of r).
- 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 now easily specify a wrapper $\mathcal{W}_{\text{GKL}}$ as special case of the above general wrapper. In the hybrid world $(\mathcal{G}_{\text{CLOCK}}, \mathcal{W}_{\text{GKL}}(\mathcal{F}_{\text{STX}}^{\Delta,p_H,p_A}), \mathcal{F}_{\text{N-MC}}^{\Delta})$ this ensures the condition of Theorem 1 and we arrive at the following composable statement:

Corollary 2. *The protocol* Modular-Ledger-Protocol$_T$ *UC-realizes the functionality* $\mathcal{G}_{\text{LEDGER}}^{\mathbb{B}}$ *in the* $(\mathcal{G}_{\text{CLOCK}}, \mathcal{W}_{\text{GKL}}(\mathcal{F}_{\text{STX}}^{1,p_H,p_A}), \mathcal{F}_{\text{N-MC}}^{1})$*-hybrid model (setting delay* $\Delta = 1$*) for the parameters assured by Theorem 1 for the above choice:*

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

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

- For each corrupted (and desynchronized) party, the adversary gets at most one query per round. Each honest miner makes exactly one query per round. This means that $q_A^{(r)} + q_H^{(r)} = n_r$.

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

[15] In a recent paper, the authors of [16] 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.

– In the PSs model, $p_H = p_A = p$ and hence $\mathsf{T}_{mp}^{\mathrm{PSs}}(r) \leq p \cdot n_r = p \cdot n$, where n is as above. With these values we get $\mathsf{mp}_A^{\mathrm{PSs}}(r) = p \cdot n_r^{\mathrm{corr}}$ and consequently $\beta_r^{\mathrm{PSs}} = p \cdot (\rho \cdot n)$, where ρn denotes the upper bound on corrupted parties in any round. Putting things together, we also have $\alpha_r^{\mathrm{PSs}} = 1 - (1-p)^{(1-\rho)\cdot n}$. 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.

We can again specify a wrapper functionality $\mathcal{W}_{\mathrm{PSs}}$ as above (where the restriction is 1 query per corrupted instead of q). We again have that the hybrid world $(\mathcal{G}_{\mathrm{CLOCK}}, \mathcal{W}_{\mathrm{PSs}}(\mathcal{F}_{\mathrm{STX}}^{\Delta,p,p}), \mathcal{F}_{\mathrm{N\text{-}MC}}^{\Delta})$ will ensure the condition of the theorem and directly yields the following composable statement.

Corollary 3. *The protocol* Modular-Ledger-Protocol$_T$ *UC-realizes* $\mathcal{G}_{\mathrm{LEDGER}}^{\mathbb{B}}$ *in the* $(\mathcal{G}_{\mathrm{CLOCK}}, \mathcal{W}(\mathcal{F}_{\mathrm{STX}}^{\Delta,p,p}), \mathcal{F}_{\mathrm{N\text{-}MC}}^{\Delta})$*-hybrid model (with network delay* $\Delta \geq 1$*) for the parameters assured by Theorem 1 for the above choice:*

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

5 Implementing a Stronger Ledger

As already observed in [16], the Bitcoin protocol makes use of digital signatures to protect transactions which allows it to achieve stronger guarantees. Informally, the stronger guarantee ensures that every transaction submitted by an honest miner will eventually make it into the state. Using our terminology, this means that by employing digital signatures, Bitcoin implements 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 $\mathcal{G}_{\mathrm{LEDGER}}^{\mathbb{B}}$.

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 as pseudo-code are provided in the full version [6].

Realized Ledger. The realized ledger abstraction, denoted by $\mathcal{G}_{\text{LEDGER}}^{\hat{\mathcal{B}}+}$, is formally specified in [6]. 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, $\mathcal{G}_{\text{LEDGER}}^{\hat{\mathcal{B}}+}$ only accepts a transaction containing the $\mathsf{AccountID}$ that was previously associated to p_i and ensures that parties are restricted to issue transactions using their own accounts.

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 $\mathcal{G}_{\text{LEDGER}}^{\hat{\mathcal{B}}+}$, 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 $2 \cdot \texttt{windowSize}$ new state blocks in the worst case. Relative to the current view of p_i this is no more than within the next $3 \cdot \texttt{windowSize}$ blocks as argued in [6].

References

1. Andrychowicz, M., Dziembowski, S.: PoW-based distributed cryptography with no trusted setup. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 379–399. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48000-7_19
2. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, Ł.: Fair two-party computations via bitcoin deposits. In: Böhme, R., Brenner, M., Moore, T., Smith, M. (eds.) FC 2014. LNCS, vol. 8438, pp. 105–121. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44774-1_8
3. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 443–458. IEEE Computer Society Press, May 2014
4. Babaioff, M., Dobzinski, S., Oren, S., Zohar, A.: On bitcoin and red balloons. SIGecom Exch. **10**(3), 5–9 (2011)
5. Backes, M., Hofheinz, D., Müller-Quade, J., Unruh, D.: On fairness in simulatability-based cryptographic systems. In: FMSE 2005 (2005)
6. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: a composable treatment. Cryptology ePrint Archive, Report 2017/149 (2017)
7. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press, May 2014
8. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_24
9. Buterin, V.: A next-generation smart contract and decentralized application platform (2013). https://github.com/ethereum/wiki/wiki/White-Paper

10. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: 42nd FOCS, pp. 136–145. IEEE Computer Society Press, October 2001
11. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). doi:10.1007/978-3-540-70936-7_4
12. Canetti, R., Shahaf, D., Vald, M.: Universally composable authentication and key-exchange with global PKI. In: Cheng, C.-M., Chung, K.-M., Persiano, G., Yang, B.-Y. (eds.) PKC 2016. LNCS, vol. 9615, pp. 265–296. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49387-8_11
13. Coretti, S., Garay, J., Hirt, M., Zikas, V.: Constant-round asynchronous multi-party computation based on one-way functions. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10032, pp. 998–1021. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53890-6_33
14. Eyal, I.: The miner's dilemma. In: 2015 IEEE Symposium on Security and Privacy, pp. 89–103. IEEE Computer Society Press, May 2015
15. Eyal, I., Sirer, E.G.: Majority is not enough: bitcoin mining is vulnerable. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 436–454. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45472-5_28
16. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: analysis and applications. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46803-6_10
17. Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Universally composable synchronous computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 477–498. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36594-2_27
18. Kiayias, A., Koutsoupias, E., Kyropoulou, M., Tselekounis, Y.: Blockchain mining games. In: EC (2016)
19. Kiayias, A., Zhou, H.-S., Zikas, V.: Fair and robust multi-party computation using a global transaction ledger. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 705–734. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49896-5_25
20. Kumaresan, R., Bentov, I.: How to use bitcoin to incentivize correct computations. In: Ahn, G.-J., Yung, M., Li, N. (eds.) ACM CCS 2014, pp. 30–41. ACM Press, November 2014
21. Kumaresan, R., Bentov, I.: Amortizing secure computation with penalties. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 418–429. ACM Press, October 2016
22. Kumaresan, R., Moran, T., Bentov, I.: How to use bitcoin to play decentralized poker. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 15, pp. 195–206. ACM Press, October 2015
23. Kumaresan, R., Vaikuntanathan, V., Vasudevan, P.N.: Improvements to secure computation with penalties. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 406–417. ACM Press, October 2016
24. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
25. Lamport, L.: Paxos made simple, fast, and byzantine. In: OPODIS (2002)
26. Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)

27. Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: anonymous distributed e-cash from bitcoin. In: 2013 IEEE Symposium on Security and Privacy, pp. 397–411. IEEE Computer Society Press, May 2013
28. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). http://bitcoin.org/bitcoin.pdf
29. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 643–673. Springer, Cham (2017). doi:10.1007/978-3-319-56614-6_22
30. Pass, R., Shi, E.: FruitChains: a fair blockchain. Cryptology ePrint Archive, Report 2016/916 (2016)
31. Rabin, M.O.: Randomized byzantine generals. In: FOCS (1983)
32. Rosulek, M.: Must you know the code of f to securely compute f? In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 87–104. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32009-5_7
33. Sompolinsky, Y., Zohar, A.: Secure high-rate transaction processing in bitcoin. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 507–527. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47854-7_32
34. Zohar, A.: Bitcoin: under the hood. Commun. ACM **58**(9), 104–113 (2015)