Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric

Mike Graf
Institute of Information Security
University of Stuttgart
Stuttgart, Germany
mike.graf@sec.uni-stuttgart.de

Ralf Küsters
Institute of Information Security
University of Stuttgart
Stuttgart, Germany
ralf.kuesters@sec.uni-stuttgart.de

Daniel Rausch
Institute of Information Security
University of Stuttgart
Stuttgart, Germany
daniel.rausch@sec.uni-stuttgart.de

Abstract—While accountability is a well-known concept in distributed systems and cryptography, in the literature on blockchains (and, more generally, distributed ledgers) the formal treatment of accountability has been a blind spot: there does not exist a formalization let alone a formal proof of accountability for any blockchain yet.

Therefore, in this work we put forward and propose a formal treatment of accountability in this domain. Our goal is to formally state and prove that if in a run of a blockchain a central security property, such as consistency, is not satisfied, then misbehaving parties can be identified and held accountable. Accountability is particularly useful for permissioned blockchains where all parties know each other, and hence, accountability incentivizes all parties to behave honestly.

We exemplify our approach for one of the most prominent permissioned blockchains: Hyperledger Fabric in its most common instantiation.

Index Terms—Accountability, Blockchain, Distributed Ledger, Distributed Systems, Security, Hyperledger Fabric

1. Introduction

Since the invention of the blockchain in 2008 [51], interest in this concept in particular and distributed ledger technology in general has grown rapidly. Initially proposed as a solution for payment systems without a central authority, blockchains and distributed ledgers have evolved into multi-purpose distributed systems with strong security promises.

While there exists a wide range of established security notions and formal security analyses for blockchains [8], [9], [11], [19], [24], [35], [36], [53], there is an important blind spot in the formal security analysis of blockchains and distributed ledgers: *accountability* [43]. On a high level, accountability requires that if certain security goals of a protocol are violated, then misbehaving parties can be identified and held accountable for their misbehavior via undeniable cryptographic evidence, which incentivizes honest behavior.

Accountability itself is a well-known concept in distributed systems and cryptography that has already been applied to a wide variety of settings (see, e.g., [6], [10], [28], [39], [40], [42]–[45], [48], [49], [61]).

In the context of blockchains, accountability is often mentioned as a desirable property [12], [18], [22], [33],

[54] and some research papers on blockchains use the concept of accountability on an informal level [14], [34]. However, there does not exist a formalization let alone a formal proof of accountability for any blockchain or distributed ledger yet (see also Section 7).

Therefore, in this work we put forward and propose a formal treatment of accountability for blockchains, which we exemplify by a very prominent permissioned blockchain (see below). With this, we provide a new formal approach for designing and analyzing blockchains.

This approach is in fact orthogonal to established security notions for blockchains, such as consistency: these security notions [24], [53] require that security goals are always fulfilled under certain assumptions. For example, blockchains should always converge to a consistent state, typically under the assumption of a round-based network model with eventual message delivery and an honest majority of participants. In this sense, security (e.g., consistency) cannot be broken as long as certain assumptions are fulfilled. We call such security notions *strict* in what follows. In contrast, our accountability-based approach accepts that security goals can be violated. However, in such a case it is guaranteed that misbehaving parties can be identified and rightfully blamed for their misbehavior. Our approach can be used in two ways:

Accountability as a stand-alone property. On the one hand, accountability is an interesting stand-alone property that can be used instead of strict security notions to obtain (different) security guarantees. Particularly for permissioned blockchains, where parties know each other and often collaborate on a contractual basis, having the guarantee that if some central security goal breaks down, then misbehaving parties can be identified and held accountable will in many cases be considered a sufficiently strong property. As demonstrated in this paper and further detailed below, using accountability as a stand-alone property allows for new constructions of secure blockchains which can leverage one or more of the following features: *(i)* Accountability might already be achievable with weaker but more efficient components compared to what is needed for strict goals. For example, as illustrated in our case study, accountable blockchains can be built from efficient crash-fault tolerant consensus protocols instead of more complex Byzantine-fault tolerant consensus protocols in order to provide accountability w.r.t. consistency. That is, while a misbehaving party can break consistency,

this party can then be identified and held accountable. *(ii)* In order to fulfill accountability properties one might need less assumptions. For example, in order to show accountability w.r.t. consistency in our case study, we do not require an honest majority or networks with eventual message delivery; strict security notions have not been proven in this setting for any blockchain yet. *(iii)* Accountability properties might be stronger in some aspects than strict security notions: as explained for our case study, for our notion of accountability w.r.t. consistency we require (and prove for the considered protocol) consistency not only for blockchains of honest parties but also of dishonest ones.

Accountability as an additional, orthogonal layer of defense. On the other hand, accountability can be used as an additional security requirement and another security layer on top of strict notions: Even in the strict setting, security can still be broken because the underlying assumptions, such as an honest majority, might not be fulfilled. In such a case, with the accountability property in place one can still identify and blame misbehaving parties. (As mentioned for the stand-alone case, in order for accountability to hold true one might need only weaker assumptions, e.g., a dishonest majority might be tolerated.) Such parties can then be excluded from future executions. Moreover, faced with the prospect of being held accountable, parties might refrain from misbehaving in the first place, making certain assumptions, such as honest majority, more likely to hold true.

Levels of accountability. We note that there are different levels of accountability (and hence levels of security) that a blockchain can provide. If, for example, for some violations of security goals only a group of parties, possibly from different companies collaborating on the same chain, can be identified to have misbehaved, but it is only clear that at least one of the parties within this group must have misbehaved, then this is an insufficient level of accountability: no individual party can for sure be held accountable, and hence, be blamed and punished. To be sufficient for practical purposes, it should be possible to hold at least one individual party accountable for misbehavior, a property called *individual accountability*.

Analyzing accountability for blockchains. Our approach for analyzing accountability for blockchains is based on a generic accountability framework from Küsters et al. [43], which we slightly adjust. A key task for instantiating this framework for the analysis of blockchains is to formalize appropriate security goals, for which parties can be held accountability if they are violated. It seems difficult, if not impossible, to formulate such goals in a way that applies to all kinds of blockchains and distributed ledgers. The reason is that even established security goals, such as consistency, mean different things in different systems: for some consistency might mean that the complete chains of different parties are prefixes of each other, for other systems this might only be required for certain parts of the chains, e.g., possibly modulo the last T blocks and/or modulo certain meta-data (where the definitions of these parts again depend on the protocols at hand), and yet for others, such as Corda [13], consistency might rather require that the combination of the partial states of different parties yields a consistent state.

We therefore chose to exemplify our approach on accountability for blockchains via a case study of one of the most important permissioned blockchains: Hyperledger Fabric [2], [31]. Analyzing Fabric within the above mentioned framework involves instantiating the framework by carrying out the following main tasks: *(i)* build an appropriate system model, including a model of corruption, *(ii)* formalize the security goals to be achieved, including, among others, consistency, *(iii)* specify who is to be blamed if a security goal is violated, including a procedure for how to identify misbehaving parties, and *(iv)* use the resulting instantiation of the accountability framework to formally analyze accountability of Fabric with respect to the specified security goals.

While the details of the above steps are specific to the protocol at hand, the case study nevertheless illustrates general techniques and serves as a blueprint for the formal analysis of accountability of other blockchains as well as distributed ledgers.

Hyperledger Fabric. Hyperledger Fabric [2], [31], or short Fabric, is a success story in the blockchain sector for the Linux Foundation [58]. This permissioned business blockchain is widely used and accepted by the industry. Forbes includes more than 20 important companies using Fabric in the Forbes Blockchain 50 list [20] — a list of leading companies that are implementing distributed ledger technology for production usage.

The driving reasons behind the increasing adoption of Fabric are several unique features tailored towards industrial contexts, such as a *execute-order-validate* architecture that allows for parallel executions of transactions and thus increases throughput in Fabric over other blockchains [21], [27] (cf. Section 2). Moreover, Fabric introduced modularity to blockchain architectures, including a pluggable consensus mechanism, i.e., the consensus algorithm is defined via a "plugin". This plugin can be chosen depending on the specific requirements at hand to create a customized instantiation of Fabric. Out of the box, Hyperledger ships Apache Kafka [3], [30] (in short, Kafka) and, very recently, Raft [30], [52] as the only production ready consensus mechanisms for Fabric, both of which are only crash-fault tolerant.

Our approach of accountability in blockchains can in principle be applied to all possible instantiations of Fabric. In particular, accountability can add a second security layer to Fabric instantiations based on Byzantine-fault tolerant consensus mechanisms (e.g., the prototype of the PBFT SMaRT-based consensus mechanism [16], [56]) which already provide strict security guarantees under certain assumptions. More interesting for this work, however, is an instantiation of Fabric using Apache Kafka: Not only is this the most widely used instantiation in production setups, it is, as mentioned, also based on a consensus mechanism that is only crash-fault tolerant, and hence, does not meet strict security notions in the presence of active attackers. Therefore, it is interesting to investigate in how far accountability as a stand-alone property can help to assess and possibly improve the security of this widely used instance of Fabric, for which the designers deliberately chose a crash-fault tolerant consensus mechanism.

In our analysis, we observe that Fabric using Kafka provides a certain weak level of accountability with re-

spect to the most crucial security goal of blockchains: *consistency*. That is, if blockchains of different parties are inconsistent, one can blame a set of parties with the guarantee that at least one of them misbehaved. However, that level of accountability, as already argued above, it insufficient for practice. Hence, we propose small modifications to Fabric with Kafka and show that this slightly modified version, which we call Fabric*, indeed achieves *individual accountability w.r.t. consistency*. We also prove that Fabric* achieves accountability w.r.t. further security goals from the literature [2], [8], [9], [11], [19], [24], [35], [36], [53] (cf. Section 3 and Appendix C). Given the importance of Fabric in practice, besides illustrating the formal treatment of accountability, we consider this result to be of independent interest.

Our case study perfectly illustrates the aforementioned features of accountability when used as a stand-alone concept: *(i)* We obtain reasonable security guarantees for a blockchain that is based on a crash-fault tolerant consensus algorithm. More generally, the concept of accountability enables new types of blockchain constructions that provide different, but reasonable forms of security guarantees. *(ii)* We obtain security results without assuming honest majorities and in a fully asynchronous network, including message loss, without assuming a round-based model or a global clock. Established strict security notions have not been proven in this setting for any blockchain: proofs typically rely on the assumptions that the majority of the parties is honest and that messages are delivered eventually; assumptions that do not always hold true in reality. *(iii)* We show accountability with respect to consistency even for chains held by malicious parties. In contrast, existing strict security notions for consistency can only be shown for chains of honest parties as malicious parties can trivially change their own chains to be inconsistent. This shows that accountability allows for enforcing new security goals that are not covered by existing security notions.

We emphasize that we do not propose that accountability should replace strict security notions. On the contrary, as mentioned, it is highly desirable to have blockchains fulfilling both strict security notions *and* accountability, as these are orthogonal and complement each other. But in some cases, such as typical deployments of permissioned blockchains, accountability when used as a stand-alone security property can offer an alternative approach to strict security notions with several advantages.

Summary of our contributions.
- We put forward a formal treatment of accountability for blockchains.
- We illustrate the use and features of accountability as a stand-alone property by Hyperledger Fabric, one of the most prominent and widely used (permissioned) blockchains in practice.
- We formally define accountability w.r.t. consistency and other central security goals for Fabric with Kafka.
- We identify individual accountability as the necessary degree of accountability for Fabric.
- We argue that this instantiation of Fabric does not provide individual accountability (w.r.t. consistency) and then formally prove that Fabric*, a slight variant, satisfies individual accountability (w.r.t. consistency

and other central security goals).
- Our case study, besides being interesting on its own right given the importance of Fabric, can serve as a blueprint for formally analyzing accountability of other blockchains.

Structure of the paper. We introduce the Fabric protocol, including the Kafka-based consensus mechanism, in Section 2. In Section 3, we explain why Fabric using Kafka does not achieve consistency in the presence of an active adversary. We also give a first intuition of individual accountability w.r.t. consistency and explain why Fabric fails to achieve this security notion. We then propose Fabric*, obtained from slight modifications to Fabric, with a formal model of Fabric* described in Section 4. In Section 5 we introduce the accountability framework we use. Then, we formally define accountability w.r.t. consistency in Section 6 and prove that Fabric* satisfies this property. We also discuss other security notions besides consistency in Appendix C, with more details provided in the appendix. We discuss related work in Section 7. We conclude in Section 8. Full details are available in the full version of the paper [41].

2. Hyperledger Fabric

Fabric is designed as a *distributed operating system* [2]. Similar to Ethereum [60], the Fabric blockchain allows distributed code execution: all transactions in Fabric are calls to code stored in the blockchain. There is not just a single global instance of Fabric but there are multiple separate instances running in parallel. Each instance has its own set of participants, chain, code, ledger state (derived from the chain), and parameters, thus allowing different (groups of) companies to run their own separated Fabric blockchains. Such instances are called *channels*. Fabric is a framework that can be instantiated with different consensus algorithms. As already mentioned in the introduction, we focus on Fabric using Kafka.

In this section, we first describe the general and abstract protocol of Fabric independently of specific consensus algorithms. We then explain the Apache Kafka-based consensus mechanism as one possible instance of a consensus algorithm. In our description of Fabric, we follow the terminology and presentation of Fabric by Androulaki et al. [2], which differs slightly from Fabric's read-the-docs [32].

2.1. Roles in Fabric

In Fabric, all participants have a *role* and are assigned to an *organization*. One can think of an organization as a company that takes part in the Fabric blockchain. The central roles are the following:

Membership Service Providers (MSP) provide access to the Fabric blockchain. More specifically, MSPs issue certificates to participants, which include the participant's role, organization, and public key.

Clients initiate transactions. They typically obtain inputs for transactions from end users. Similar to other blockchains, clients do not keep a copy of the chain.

Endorsing Peers or Peers are essentially the "miners" and "full-nodes" of Fabric. They execute transactions from

clients. They also verify and replicate the chain, i.e., they keep a copy of the full Fabric chain and allow clients to query data from the chain.

An Ordering Service is an abstract concept that provides a "consensus service" in Fabric. The ordering service is typically instantiated via a distributed protocol potentially using multiple different sub-roles, but it can also be implemented via a single centralized machine (cf. the *Solo* ordering service [30]). This service is one of the major differences compared to other blockchains. With this service, consensus and transaction execution are separated and an *execute-order-validate architecture* is established. Transactions from clients are first executed by (some or all of the) peers; afterwards, transactions and execution results are forwarded by the clients to the ordering service. The ordering service forms blocks from the incoming transactions without re-executing transactions or checking validity. The service then distributes the blocks to all peers.

Besides these four central roles (MSP, client, peer, ordering service), there are some additional (sub-)roles in Fabric, namely administrators, leading peers, anchor peers, and committing peers. Administrators are allowed to, e.g., update smart contracts and to add organizations. The various peer types describe different peers with specific tasks. For example, leading peers directly receive blocks from the ordering service and then distribute these blocks to other peers. While, for simplicity of presentation, we subsume all peer subroles into a single peer role and omit administrators, all of our results also hold true when these (sub-)roles are taken into account.

2.2. The Fabric Protocol

We now describe the Fabric protocol in detail, including the precise interactions between the roles described above.

A set of organizations creates a new Fabric channel by first externally agreeing on a genesis block. The genesis block contains an initial setup of the Fabric channel, including a channel ID, organizations and thus participants that are allowed to access the channel. For simplicity of presentation, we also include in the genesis block an initial state of the channel, smart contracts available in the channel (i.e., deterministic programs which are called *chaincode* in Fabric), and a so-called endorsement policy for each chaincode; formally, all of this information is also agreed upon externally but would be installed separately from the genesis block. We explain endorsement policies in the protocol description below.

In what follows, we exemplify the execution of the Fabric protocol by following the path of a transaction from its initial creation until it becomes part of the blockchain. Figure 1 depicts the corresponding message flow.

1. Transaction Proposal. To interact with the chain, e.g., to read from the chain or to change the state stored in the chain, a client calls a chaincode by initiating a *transaction proposal*. The proposal contains the client's identity, the chaincode the client wants to invoke, input data for the chaincode call, a sequence number, and the client's signature over this data. The hash over all of this data (excluding the signature) is used as transaction ID. A
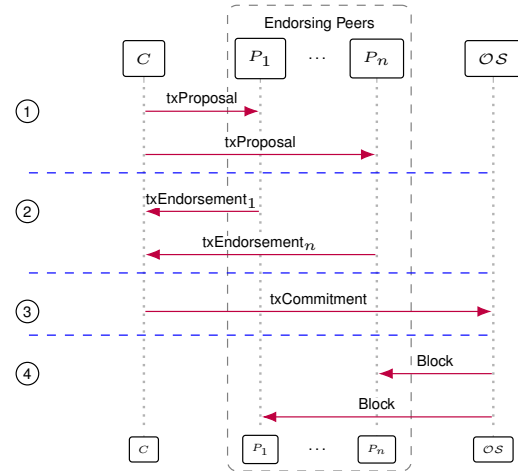

Figure 1. Example Message Flow in Fabric with Client C, (endorsing) peers $P_1, \ldots, P_n$, and ordering service $\mathcal{OS}$.

client requests a subset of peers to execute her transaction proposal (cf. ① in Figure 1). The endorsement policy of the chaincode specifies, besides other things, which peers and/or how many peers have to execute a proposal. After having distributed the proposal, clients wait for the results of the executions from the peers, the so-called *endorsements*.

2. Transaction Endorsement. Upon receiving a new transaction proposal, peers first check whether the client may actually call the chaincode from the proposal. This access control mechanism is also specified in the endorsing policy. If the proposal fulfills the policy, then peers execute the proposal, i.e., they run the chaincode with the input provided in the proposal. The execution of a proposal is bounded by a threshold computation time to ensure termination of executions. At the end of a successful execution, a peer generates an *endorsement*, which is a confirmation of the peer that the transaction and its results should become part of the blockchain. An endorsement contains the identity of the peer, the original transaction proposal,[1] the data that was read from the ledger state (as defined by the blockchain held by the peer) during the execution of the transaction proposal (called *readset*), and the changes in the state caused by the execution of the transaction (called *writeset*). The endorsement is then sent to the client that initiated the proposal (cf. ② in Figure 1).

Note that peers do not apply the writesets to their chain/ledger state yet, i.e., in this phase the execution of a proposal does not change the local state of the peer. Also note that, since the ledger state of different peers may differ (e.g., because a peer has not received the most recent blocks yet), the writesets computed by different peers for the same proposal can differ as well.

3. Transaction Commitment. The client receives the endorsements. More specifically, she keeps collecting endorsements for her transaction proposal until she has a set

---

1. According to the specification of Fabric [32], an endorsement *has to* contain the transaction ID of the proposal. The endorsement *should* contain the transaction proposal, which is what we assume in our model. However, all of our results also hold true if transaction proposals themselves are not part of endorsements.

of endorsements such that all of them include the same readset and writeset and the set of endorsements meets the endorsement policy (i.e., the set includes sufficiently many endorsements from the expected set of peers). Then, the client broadcasts the initial transaction proposal and the set of endorsements as a so-called *transaction commitment* to the ordering service (cf. ③ in Figure 1). A transaction commitment where the set of endorsements does not meet the above mentioned requirements is considered invalid. While invalid transaction commitments sent to the ordering service can also be included in the chain (see the next step below), their validity will later be checked during block verification. Such invalid transaction commitments are then ignored for the purpose of deriving the ledger state from the chain. Hence, clients would typically not submit such commitments.

4. Block Generation and Distribution. The ordering service is supposed to create blocks from the transaction commitments it receives. As mentioned before, for now we consider the ordering service to be a black box, with the most commonly used concrete instantiation based on Apache Kafka described in Section 2.4.

When an ordering service receives a transaction commitment, it stores the commitment for inclusion in an upcoming block. A block in Fabric consists of a block number (the ID of the block), a block body, i.e., an ordered sequence of transaction commitments (including all endorsements), the hash of the previous block, the identity of the participant of the ordering service that formed the block (recall that the ordering service is usually a distributed system consisting of different entities), and a signature of that participant over the full block (excluding the signature field). The hash of the previous block is computed over the previous block, excluding the block creator's identity and the signature. Blocks are created using sequential block numbers starting at 1. When a new block is created, the ordering service removes some of its transaction commitments from its storage, orders them, and then adds them to the body of a new block. The resulting block is then sent to the peers in the channel (cf. ④ in Figure 1).

Importantly, to increase throughput, ordering services generally do not validate transactions. Thus invalid and malformed transactions can be part of a block in Fabric; these are filtered out in the next step.

5. Block Validation and Ledger State Update. Upon receiving a new block, peers validate whether they accept the block, i.e., whether the block has the expected format, contains a valid signature from an orderer identity, and the block number is as expected (i.e., it increments the number of the previously accepted block by one). If peers accept a block, they update their local copy of the ledger state according to the new block. More specifically, peers sequentially process transactions from the new block. For each transaction, they first check validity of the transaction, i.e., whether it has the expected format and the transaction commitment meets the endorsing policy. Valid transactions are then applied to the local ledger state. In contrast to, e.g., Ethereum, this is not done by re-executing the transaction, but rather by using the precomputed read- and writesets from the endorsements of the transactions. Peers first check whether the readsets match their current

ledger state (if not, the transaction is marked as invalid and is not applied to the state) and then apply the changes from the writesets.

We emphasize the following fundamental differences between Fabric and other classical blockchains:

Firstly, the separation of *(i)* execution of transactions (Step **2.**), *(ii)* transaction ordering and block generation (Step **4.**), and *(iii)* transaction validation (Step **5.**) is the core of the *execute-order-validate* paradigm. In contrast, typical blockchains use the *order-execute* paradigm where *(i)* miners order the transactions into blocks and propagate the blocks to all full-nodes and *(ii)* full-nodes validate and execute transactions if they accept a new block. The execute-order-validate paradigm enables a high throughput in Fabric compared to other blockchains [2], [21], [27]. Note that one of the reasons for the improved throughput is that peers in Fabric simply trust the writesets contained in a valid transaction commitment, instead of recomputing the correct output on their own. Hence, the endorsement policy should be specified in such a way that it is reasonable to assume that the set of peers endorsing a transaction proposal contains at least one honest peer. In this case, malicious peers cannot manipulate the outcome of a transaction.

Secondly, in contrast to, e.g., Bitcoin or Ethereum, there is no longest chain rule or a similar construct that allows for changing the current chain. Instead, chains of peers are final in the sense that they only extend their chains, but never remove/change previously accepted blocks.

This completes the description of the basic Fabric framework. Next, we first briefly describe Apache Kafka and then how Kafka is used in Fabric.

2.3. Apache Kafka

Apache Kafka [3] is a so-called *streaming platform*. It is a crash fault-tolerant distributed system optimized for durability, high throughput, and reliable message distribution. In essence, Kafka provides an (atomic) broadcast channel: Kafka clients can send messages to the channel, which are then arranged into a total order. The resulting message stream can be accessed by the clients, allowing all clients to obtain the same sequence of messages. We give only a high-level overview of Kafka that is sufficient to follow our paper.

Kafka consists of two main components: *Kafka clients* and a *Kafka cluster*. Clients can publish messages in the Kafka message stream by sending those messages to the Kafka cluster, and clients can also retrieve the complete ordered message stream from the Kafka cluster. The Kafka cluster consists of several machines called *Kafka brokers*. One of the brokers is the so-called *Kafka leader*. The leader handles all requests from clients. That is, upon receiving a message from a client, the leader adds an incremental message ID (a counter), called *offset*, to the message and appends the message to the message stream. Clients can send a request to the leader to obtain the full message stream. All non-leading brokers provide redundancy by replicating the state of the leader. If the leader crashes, then the remaining brokers elect a new leader (via Apache ZooKeeper [4], [29]) who then takes over the duties of the former leader. In practice, the whole
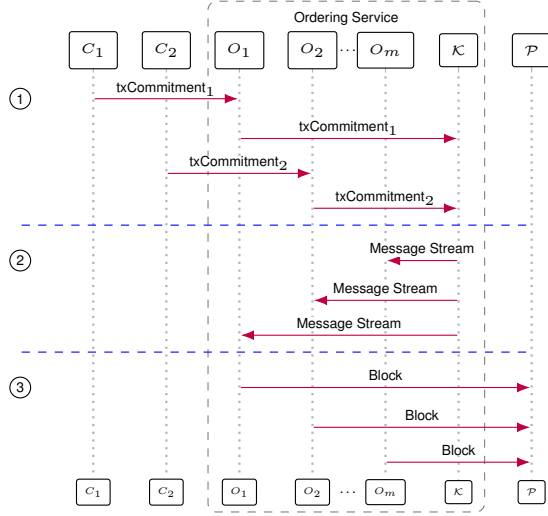
Figure 2. Ordering and Block Generation with clients C_1, C_2, orderers $O_1, \ldots, O_m$, an Apache Kafka Cluster $\mathcal{K}$, and (endorsing) peers $\mathcal{P}$

Kafka cluster, including all broker machines, is usually run as a service in a data center operated by a single entity [55], [59].

2.4. Fabric with Kafka-based Ordering Service

In Fabric, a Kafka-based ordering service consists of two components: multiple *orderers* (which act as Kafka clients) and a *Kafka cluster*. Orderers are typically run by the various organizations that take part in a Fabric channel, while the Kafka cluster is, as mentioned, run by a single organization in a data center. Orderers provide the external interface of the ordering service to the clients and peers of the Fabric channel. They receive transaction commitments, use the Kafka cluster to establish a total order over the incoming commitments, cut blocks from this ordered sequence, and finally deliver the blocks to the peers. We describe each step in more detail in what follows; an example message flow of two clients submitting commitments is given in Figure 2.

1. Forward Transaction Commitments to Kafka Cluster. Upon receiving transaction commitments from clients (cf. ① in Figure 2), orderers forward the messages to the Kafka cluster. Note that orderers do not perform any checks, such as checking whether a commitment meets the endorsement policy.

2. Distribute Ordered Sequence of Transactions. The Kafka cluster orders the incoming transaction commitments and attaches a message ID to every transaction (cf. ② in Figure 2). All orderers then receive an ordered sequence of transaction commitments from the Kafka cluster.

3. Block Generation or Cutting. All orderers "cut" the sequence of transaction commitments into blocks via a deterministic block cutting algorithm CUTBLOCK. This algorithm creates new blocks depending on *(i)* the maximum number of transactions in a block, *(ii)* the maximum size of the resulting block, and *(iii)* time passed since the last block. On a technical level, to ensure consistent

block cuts for all orderers in the cases *(ii)* and *(iii)*, an orderer running into one of these cases first broadcasts a special cutting message via the Kafka cluster. This message contains the ID of the block that the orderer wants to create via a cut. The Kafka cluster puts this message into the message stream, just like other messages (in the context of Fabric those are typically transaction commitments). This cutting message then tells an orderer to cut a new block up to this cutting message if the ID in that cutting message matches their next block ID (otherwise the cutting message is ignored). The new block contains all transaction commitments as ordered by Kafka since the last block. The cutting messages themselves are not part of a block. In the improved version of Fabric presented in Section 3, this is different: blocks will contain more information.

Orderers then add their name to the new block and sign the whole block. Thus, blocks from different orderers are supposed to be equal content-wise but differ in some of the meta-data, i.e., block creator and signature. All orderers push the resulting blocks to peers (not necessary all peers, see next).

In a typical Fabric setup using a Kafka-based consensus algorithm, *(i)* every organization runs its own peers to keep a copy of the chain, *(ii)* all organizations participate in the distributed ordering service, i.e., all organizations run at least one orderer as part of this service, *(iii)* orderers usually push blocks to all peers in their own organization, and *(iv)* if clients need to retrieve data from the chain, they usually query peers within their own organization; as explained, a state change (via a transaction) might involve sending transaction proposals to several peers also outside of the own organization, depending on the endorsement policy.

3. Accountability of Fabric with Kafka

To analyze accountability of Fabric with Kafka, we first have to state the security goals that we want to ensure via accountability. For this purpose, we first recall security goals of (public) blockchains from the literature, including consistency, the most important goal. It is quite obvious that, by construction, Fabric using Kafka does not satisfy consistency as a strict security goal (in the sense of the introduction). But, as we will observe in Section 3.2, Fabric also does not guarantee individual accountability w.r.t. consistency, i.e., while consistency might be broken, it might not be possible to blame individual parties for their misbehavior (only a set of parties, which is insufficient). We thus propose some slight modifications to Fabric with Kafka to improve accountability; we denote this modified version by Fabric*. In Section 6, we then show that Fabric* indeed provides individual accountability w.r.t. consistency as well as w.r.t. several other common security goals for blockchains, including those discussed next.

3.1. Security Goals of Blockchains

Garay et al. [24] and Pass et al. [53] established the following security notions for public blockchains: *consistency*, *chain-quality*, *chain-growth*, and *liveness*. In the

following, we briefly recap all security goals and discuss their applicability to Fabric.

Consistency (also called *common-prefix*) states that the chains of honest participants share a common prefix. Consistency is the most essential security goal for blockchains as it is a necessary requirement for obtaining a shared global (ledger) state.

Chain-quality states that the percentage of blocks of a chain that were created by malicious miners is upper bounded. This notion assumes that each block of a chain has exactly one unique "creator" and can thus be considered as either honestly or maliciously created. This, however, is generally not true for Fabric, e.g., when using Kafka: all of the orderers create and sign a block for every block ID. Hence, the goal of chain-quality is not directly applicable to the setting of Fabric. Conceptually, chain-quality is mainly used in public blockchains to argue that there is no censorship of transactions and thus to prove the security goal of liveness (see below).

Chain-growth states that, over time, a blockchain continues to grow. This is a necessary goal for blockchains which use the longest chain rule: If the blockchain stopped growing, it would allow for, e.g., 51%-attacks or long range attacks [25], where an attacker tries to overwrite some history of the chain by creating a malicious fork that is longer than the current chain. However, since in Fabric peers only extend their chains without changing previous blocks (which implies so-called *finality*), Fabric is not vulnerable to this class of attacks.

Liveness states that transactions cast by honest clients will eventually be included in the blockchain. Liveness clearly is a desirable goal of blockchains.

Other security goal. In addition to the above standard security goals, the designers of Fabric want to achieve the following goals [2]: *(i) Hash chain integrity* states, roughly speaking, that all honest peers share the same hash chain (according to Fabric's block definition), *(ii) No skipping* states that honest peers do not have gaps in their chain (i.e., all blocks in the chain of a peer have sequential IDs), and *(iii) No creation* states that chains of honest peers always consist of transactions cast (and signed) by Fabric clients.

Since consistency is the most important and fundamental security goal of blockchains, in our presentation, we focus on this goal. Except for chain-quality, which is not directly applicable to Fabric using Kafka, we, however, also cover all other mentioned goals in our work (see Appendix C).

3.2. Security Goals of Fabric

As already mentioned earlier, Fabric using Kafka trivially does not provide consistency in the presence of misbehaving participants. For example, a corrupted Kafka cluster can distribute messages in different orders to different orderers. Hence, the blocks that orderers build from those transactions will also be different and in particular define different ledger states. Even if one were to assume that Kafka operates honestly, there would still be other issues. For example, corrupted orderers can deliver different blocks to different peers, again violating consistency.

While this would be devastating for any anonymous public blockchain, Fabric is a permissioned blockchain

where the real identities of participants are known. So while consistency cannot be guaranteed in a strict sense, we put forward the idea of *(individual) accountability*, which so far has not been considered for blockchains. That is, if consistency is violated it should be possible to hold (individual) parties accountable for their misbehavior. More specifically, there should be cryptographic evidence that rightfully convinces all parties – parties who are part of the channel, but also external observers or an auditor – that one or more specific parties misbehaved. Such parties would then face punishment and image loss, which should deter them from misbehaving in the first place.

Therefore, our goal is to show *(individual) accountability w.r.t. consistency* for Fabric. This would then formally guarantee that consistency either holds true or we can blame at least one individual participant for breaking consistency. Unfortunately, we find that Fabric with Kafka does not achieve this goal either: there are cases where consistency is broken but one cannot uniquely identify a misbehaving party. For example, if there are two peers with different chains where each chain individually is valid (signatures verify etc.) but the two chains are inconsistent, say both contain a block with the same block number but the blocks contain different transactions, then it is not clear who misbehaved. It could, as mentioned above, be that the Kafka cluster has maliciously sent different transaction sequences to different orderers or it could be that a malicious orderer has not formed blocks correctly, given the transaction sequence received from the Kafka cluster. It is not clear who to blame, the data center running the Kafka cluster or some of the orderers. The orderers and the data center typically all belong to different organizations and companies, and hence could and would put the blame on one another. So, no individual party could be held accountable, which is insufficient in practice.

3.3. An Individually Accountable Fabric Protocol

In order to obtain individual accountability for Fabric with Kafka, we have to modify the protocol. In principle, one could add the PeerReview [28] system on top of Fabric with Kafka: PeerReview is a generic protocol that can be added to an arbitrary (deterministic) protocol to make said protocol (individually) accountable. While this would improve accountability of Fabric with Kafka to the desired level, this approach comes with a few downsides: *(i)* PeerReview assumes eventual message delivery and loosely synchronized clocks, *(ii)* PeerReview runs in an additional layer on top of the actual protocol, requires complete message logs from all parties, and produces a communication overhead by requiring parties to broadcast information on their current state, and *(iii)* to identify misbehaving parties, PeerReview requires "auditors" to recalculate the full internal state of suspects. Hence, a simpler solution tailored specifically towards Fabric with Kafka and with less overhead is preferable. In particular, it should be possible to identify misbehaving parties solely based on blockchain data instead of requiring separate message logs.

Therefore, instead of using the generic PeerReview system, we rather propose some minor modifications to

Fabric with Kafka; we denote this modified version by *Fabric**.

Jumping ahead, we first note that Fabric* indeed achieves individual accountability w.r.t. consistency solely based on blockchain data with small overhead. This result holds true for dynamically corruptible participants in fully asynchronous networks. In particular, we do not need additional assumptions such as honest majorities or networks with eventual message delivery, which are common assumptions in the analyses of other blockchains.

Modifications to Fabric. We obtain Fabric* from Fabric with Kafka by two small changes.

Firstly, we essentially have the Kafka cluster sign each message it includes in its (ordered) message stream. Recall that the stream contains mostly transaction commitments, but also other messages such as cutting messages from orderers. We use Merkle trees [50] to make the signing process more efficient. More specifically, the resulting messages distributed by Kafka are of the following form: $(id, pid, msg, mr, mp, s)$ where id is the message ID (i.e., the counter of the enumeration), pid is the identity of the Kafka leader that signed this message, msg is the message payload (usually a transaction commitment), mr is the root of the Merkle tree, mp is a Merkle proof that proves that (id, pid, msg) is a leaf of the Merkle tree with root mr, and $s = \text{SIGN}_{sk(pid)}(mr)$ is a signature of the root mr from pid. Importantly, the signature on a message is also over the message's ID, which is a consecutive number that uniquely defines the order of all messages. In Appendix A, we provide a detailed specification of this change.

Secondly, we require that orderers cut blocks by including a complete consecutive section of the Kafka message stream, including all message IDs and valid signatures, without dropping any messages. In particular, there may be no gaps (according to the message IDs, which must be consecutive) within one block and across consecutive blocks. Note that this means that also messages which cannot be parsed as transactions, such as cutting messages, are included in blocks. This removes any ambiguity that might otherwise be present at different orderers, e.g., due to dropped messages on the network, and effectively changes the computation of the blocks to be a deterministic function on the same content across all orderers. Thus, all honest orderers generate the same (prefix of the) "correct" chain, allowing us to identify misbehaving orderers by comparing their outputs to this "correct" chain. Peers are changed accordingly, i.e., they now also check whether a new block meets the above criteria and reject those blocks that do not pass the check.

4. Security Model of Fabric*

We now describe the formal model that we use to analyze and show accountability of Fabric* (formal definitions of all machines used in our model are available in the full version [41]). Our basic model is similar in spirit and inspired by those of Garay et al. [24], Pass et al. [53], and the recent work on the Ouroboros family [8], [19], [35], [36]. However, as already mentioned and further explained below, it differs in important aspects.

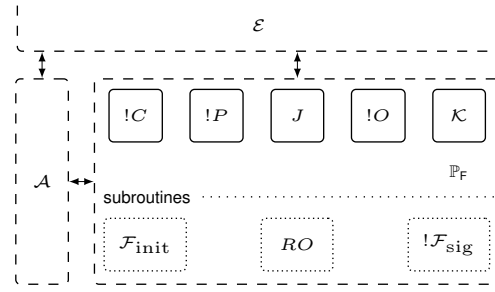We first explain a few high-level points of our modeling: We model Fabric* in the *IITM model* [37], [47],

Figure 3. IITM Model of Fabric*: the upper layer of machines in $\mathbb{P}_F$ are connected via tapes to the environment $\mathcal{E}$ and via tapes to the lower layer/subroutines $\mathcal{F}_{\text{init}}$, *RO*, and $\mathcal{F}_{\text{sig}}$. Furthermore, all machines in $\mathbb{P}_F$ are connected via tapes to $\mathcal{A}$.

a model for universal composability similar to the UC model [15]. We model a fully asynchronous setting, i.e., with message loss and without assuming rounds or a global clock. This is in contrast to most analyses of other blockchains, including the above mentioned works, which require networks with bounded message delay to show their security results and use rounds and/or clocks to model, e.g., limited access to computational resources. We also model full dynamic corruption of parties without assuming any limitations on, e.g., the number of corrupted participants. This is again in contrast to most analyses of other blockchains, which generally need assumptions on the number of honest parties such as, e.g., an honest majority. Furthermore, we provide a wide range of options to the adversary who is, for example, able to choose/change the Kafka leader arbitrarily and initiate block cutting messages at any point in the run. For simplicity of presentation, we model a static, but arbitrarily big number of participants that does not change throughout a run of a system.

Modeling the Fabric* Protocol. In the following, we describe our model in detail. We start by giving a brief overview of the IITM model that is sufficient to understand our Fabric* model and analysis.

IITMs are probabilistic Turing machines which can communicate via connected *input* and *output tapes*. A set of IITMs is called a *system*; in such a system, IITMs can be marked via a special so-called bang operator "!". For example, $M_1 \,|\, !M_2$ is a system consisting of two IITMs M_1 and M_2. In runs of a system $\mathbb{Q}$, written $\mathbb{Q}(1^\eta)$ where $\eta \in \mathbb{N}$ is the security parameter, there can be multiple instance of machines marked by "!", but only a single instance of each of the other machines. Only one instance is active at a time, which can send messages to other instances via a connected tape; that instance then becomes active. Different instances of one machine can be addressed by unique IDs. In our Fabric* model, these IDs are party IDs of the participants.

We model our Fabric* protocol via the following system of IITMs

$$\mathbb{P}_F := \,!C \,|\, !P \,|\, !O \,|\, \mathcal{K} \,|\, J \,|\, \mathcal{F}_{\text{init}} \,|\, !\mathcal{F}_{\text{sig}} \,|\, RO$$

(cf. Figure 3). Here the machine C is a client, P is an (endorsing) peer, and O is an orderer. There can be multiple instance of these machines/roles (indicated by "!"), where each instance models one participant with

a unique participant ID (PID) running in that role. The machine $\mathcal{K}$ models the Kafka cluster, i.e., a single instance subsumes all Kafka brokers (see below). Finally, J is an IITM called judge, used for accountability, i.e., it has to render verdicts on misbehaving participants (cf. Section 5 and Section 6.2). All of these machines have three subroutines that they directly connect to, namely an ideal signature functionality $\mathcal{F}_{\text{sig}}$ (where each instance models one key pair of a participant), a random oracle RO, and an ideal initialization functionality $\mathcal{F}_{\text{init}}$ (which models system parameters including the genesis block). The protocol $\mathbb{P}_F$ runs with a network attacker $\mathcal{A}$, modeling (fully asynchronous) network communication, and an environment $\mathcal{E}$ that orchestrates the run. As usual, the runtimes of the different entities are defined in such a way that the combined system $\mathcal{E} \,|\, \mathcal{A} \,|\, \mathbb{P}_F$ runs in polynomial time in the security parameter (cf. [47]).

The initialization machine $\mathcal{F}_{\text{init}}$ is parameterized by chaincodes, endorsement policies, and the sets of PIDs of the participants for each role of the Fabric* channel.[2] At the start of the run, $\mathcal{E}$ triggers $\mathcal{F}_{\text{init}}$ which then retrieves public signature keys of all participants from $\mathcal{F}_{\text{sig}}$; these keys are provided to other participants, modeling a (certificate based) public key infrastructure. We note that, while $\mathcal{E}$ is an arbitrary environment and can thus technically also trigger other machines first, $\mathbb{P}_F$ is defined in such a way that nothing much would then happen. Afterwards, roles follow their specification according to our Fabric* protocol (cf. Section 2.2 and Section 3.3) while using the subroutines $\mathcal{F}_{\text{init}}$ to obtain parameters, participant lists, and public keys, using $\mathcal{F}_{\text{sig}}$ to sign and verify messages, and using RO to compute hashes. More specifically, transaction proposals from clients are initiated by $\mathcal{E}$, who provides the external input that is used in the transaction. We do not explicitly model a (global) clock, which is used in reality to decide when a block cutting message should be sent by an orderer; we instead allow the adversary $\mathcal{A}$ to issue such messages at arbitrary points (via an arbitrary orderer). All communication between the roles is done via an unprotected network, i.e., messages are sent to the network attacker $\mathcal{A}$ who is then free to modify, forward, or drop messages arbitrarily. Note that we do not include secure channels; while they can be desirable to protect privacy of the data in the blockchain, they are not needed for our results.

The Kafka cluster requires some additional explanation. We model Kafka via an IITM $\mathcal{K}$ which internally emulates all Kafka brokers. In particular, $\mathcal{K}$ signs incoming messages with the key of the current Kafka leader and then outputs a globally unique message stream. We allow the adversary $\mathcal{A}$ to choose and change the current leader of the Kafka cluster (who is the one that signs new incoming messages) arbitrarily at any point in the run. This safely over-approximates real behavior, where the leader might change, for example, due to crashes.

In addition to their regular protocol tasks, peers also forward all accepted blocks to the judge J via a directly connected tape. These blocks are used as evidence by the judge J to detect and blame misbehaving participants. In reality, this models that peers with, e.g., inconsistent chains would provide these chains to other parties (e.g., other peers or external observers) who would then figure out which participant caused the problem (cf. Section 6.2 for a more detailed explanation).

Modeling Corruption. The adversary $\mathcal{A}$ can dynamically corrupt any instance of C, P, O, and $\mathcal{K}$ during the execution of the protocol. Upon corruption, the adversary obtains full control over the corrupted instance, i.e., the instance acts as a message forwarder for the adversary. This, for example, allows for signing messages via $\mathcal{F}_{\text{sig}}$ in the name of the corrupted instance or reporting arbitrary data to the judge in the name of a corrupted peer.

In the case of the roles C, P, and O, a corrupted instance models a single corrupted/malicious participant. In the case of $\mathcal{K}$, corruption models that the entity that runs the Kafka cluster is corrupted and hence no longer ensures correct operation. As mentioned before, the whole cluster is typically run by a single entity. This entity should be accountable for any errors produced by the Kafka cluster.

The adversary may also dynamically corrupt arbitrary instances of $\mathcal{F}_{\text{sig}}$. If an instance of $\mathcal{F}_{\text{sig}}$ is corrupted, this allows the adversary to forge signatures in the name of the corrupted instance, modeling that the adversary gained access to the signing key. Because this allows the adversary to impersonate the participant that owns this key, we also consider an instance of C, P, or O to be (implicitly) corrupted if the instance of $\mathcal{F}_{\text{sig}}$ that is used for signing is corrupted. Similarly, we consider $\mathcal{K}$, which uses multiple instances of $\mathcal{F}_{\text{sig}}$ for multiple brokers, to be corrupted if at least one of these instances/signature keys is corrupted, again following the intuition that in this case the entity operating the Kafka cluster can no longer ensure correct operation.

We do not allow corruption of $\mathcal{F}_{\text{init}}$, RO, or the judge J. $\mathcal{F}_{\text{init}}$ models trusted system parameters that all participants have externally agreed upon, as well as a trusted membership service provider; thus, $\mathcal{F}_{\text{init}}$ is a trusted component of the system. Similarly, RO models a secure hash function. The judge J is an abstract judging algorithm that identifies misbehaving participants. Such an algorithm must be run honestly to obtain meaningful results.

5. Accountability

Our formal definition of accountability w.r.t. consistency (and the other security goals mentioned in Section 3.1), presented in Section 6, is based on the generic accountability framework by Küsters et al. [43]. We now recall and slightly adapt this framework, starting with some intuition.

5.1. Intuition

In general, it is not necessary (or even possible) to penalize *every* type of misbehavior of a party. For example, a corrupted party might run different internal computations, which, however, yield outputs that are "as good" as outputs computed by an honest party.

Instead, one wants to capture that if certain (crucial) *security goals* are violated in a protocol run, then one can hold some parties accountable for the violation. Such

---

2. Thus, $\mathcal{F}_{\text{init}}$ not only models externally agreed upon parameters but also models the membership service provider of Fabric*.

goals can be, e.g., correctness of the output in runs of an MPC protocol [10] or correctness of election results [43]. In this work, among others, one main goal is *consistency of blockchains*, which, as mentioned, we formalize in Section 6. Holding a party accountable is formalized via a so-called *judge*, as already mentioned in the previous section. A judge, also called a *judging procedure*, is a special algorithm that, throughout the protocol run, gathers data (evidence) from some or all of the participants and renders a verdict on a group of participants once it detects misbehavior. Intuitively, one requires that the judge renders a verdict once a goal is violated, and never blames a group where everyone was honestly following the protocol (these two properties are called *completeness* and *fairness*). In reality, such a judging algorithm can be used by protocol participants, external auditors, or real judges to determine misbehaving (groups of) parties.

In the general version of accountability, a judge is allowed to blame a group of parties where it is not clear which exact party/parties misbehaved (fairness of the judge only guarantees that at least one person in that group was misbehaving). As already discussed in the introduction, this is too weak in practice, which is why we aim for *individual accountability*, as mentioned before.

5.2. Formal Definition of Accountability

We now formalize the intuition given in the previous section. While we follow the accountability framework from [43], we slightly adapt and extend the framework to (i) allow the judge to render a verdict at any point of a run (instead of only at the end of a run) and to (ii) work for dynamic corruption (instead of only static corruption).

Let $\mathbb{Q}$ be a system of IITMs that includes a judge IITM J and let A be a set of participants (sometimes called *agents*). Typically, participants $p \in \mathsf{A}$ are interpreted as PIDs, and p usually correspond to exactly one instance of a machine in a run of $\mathbb{Q}$.

A *verdict* of J is a boolean formula ψ built from propositions of the form $\mathsf{dis}(p)$ where $p \in \mathsf{A}$. A proposition $\mathsf{dis}(p)$ of a judge expresses that p (supposedly) violated the protocol w.r.t. a goal. To illustrate verdicts, consider the following examples for $p_1, p_2 \in \mathsf{A}$: If a judge renders the verdict $\mathsf{dis}(p_1) \vee \mathsf{dis}(p_2)$, then he claims at least one of the parties p_1, p_2 has misbehaved; analogously, a verdict $\mathsf{dis}(p_1) \wedge \mathsf{dis}(p_2)$ expresses the claim that both p_1 and p_2 misbehaved. We denote the set of all verdicts by $\mathcal{V}$.

The correctness of a verdict at some point in a run is defined as follows: Let ω be a run of $\mathbb{Q}$ and let $\lfloor \omega \rfloor$ be some point in the the run ω. We set $\mathsf{dis}(p)$ to true at point $\lfloor \omega \rfloor$ iff the party p exists and is corrupted at point $\lfloor \omega \rfloor$. This is extended to verdicts (boolean formulas) in the obvious way. We write $\lfloor \omega \rfloor \models \psi$ to say that the verdict is true at $\lfloor \omega \rfloor$; we write $\lfloor \omega \rfloor \not\models \psi$ otherwise.

The "violation of a goal" of a protocol is formalized via *properties* of runs. Formally, a property α of a protocol $\mathbb{Q}$ is a subset of all runs of $\mathbb{Q}$. Intuitively, α contains all runs of $\mathbb{Q}$ where the desired property is violated; for example, we later consider a property α that contains all runs where consistency of Fabric* is violated. Every property α is accompanied by a so-called *accountability constraint*, which formalizes minimal verdicts that are

expected from the judge for runs in α. Such constraints can be used to specify, e.g., that a judge should render a verdict on at least one specific party of the protocol. Formally, an accountability constraint $\mathcal{C}$ of $\mathbb{Q}$ is of the form $\mathcal{C} := (\alpha \Rightarrow \psi_1 \mid \cdots \mid \psi_k)$ where α is a property of $\mathbb{Q}$ and $\psi_1, \ldots, \psi_k \in \mathcal{V}$ are one or more verdicts. Such a constraint says that J, in a run from α, should output a verdict ψ that (logically) implies at least one of the verdicts $\psi_1, ..., \psi_k$. Note that the judge is allowed to render stronger verdicts or verdicts that imply multiple verdicts. We say that J *ensures* $\mathcal{C}$ in a run ω of $\mathbb{Q}$ if either $\omega \notin \alpha$ or at some point of the run ω the judge J states a verdict ψ that implies one of $\psi_1, \ldots, \psi_k$.

For example, for Fabric* we later consider an accountability constraint of the form

$$\mathcal{C} := (\alpha \Rightarrow \mathsf{dis}(p_1) \mid \ldots \mid \mathsf{dis}(p_m) \mid \mathsf{dis}(o_1) \mid \ldots \mid \mathsf{dis}(o_l) \mid \mathsf{dis}(\mathfrak{K}))$$

for peers p_i, orderers o_i, and the Kafka cluster $\mathfrak{K}$ where α essentially contains all runs where consistency is violated. So $\mathcal{C}$ requires that if consistency is violated in a run, the judge has to blame at least one of the mentioned parties individually in that run; by the fairness property of accountability (see below) it is guaranteed that this party indeed misbehaved. More technically, for example, a verdict $\mathsf{dis}(o_1) \wedge \mathsf{dis}(\mathfrak{K})$ from the judge meets the requirement of $\mathcal{C}$ as it implies $\mathsf{dis}(o_1)$ as well as $\mathsf{dis}(\mathfrak{K})$, i.e., there are two individual parties that are blamed. In contrast, a verdict $\mathsf{dis}(o_1) \vee \mathsf{dis}(\mathfrak{K})$ would not be sufficient.

An accountability constraint $\mathcal{C} = (\alpha \Rightarrow \psi_1 \mid \cdots \mid \psi_k)$ is said to achieve *individual accountability* if for every $i \in \{1, \ldots, k\}$ there exists an identity p such that ψ_i implies $\mathsf{dis}(p)$. In other words, each $\psi_i, i \in \{1, \ldots, k\}$ determines at least one misbehaving party. Clearly, the example constraint above achieves individual accountability.

A set Φ of accountability constraints for a system $\mathbb{Q}$ is called an *accountability property* of $\mathbb{Q}$. We write $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi)]$ to denote the probability that the judge J does not ensure $\mathcal{C}$ for some $\mathcal{C} \in \Phi$ in a run of $\mathbb{Q}(1^\eta)$, where the probability is taken over the random coins of the run and 1^η is the security parameter given to the IITMs. That is, intuitively, we would like to ensure all accountability constraints in Φ. Furthermore, we denote by $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \not\models \psi\}]$ the probability that the judge J states a verdict ψ at some point $\lfloor \omega \rfloor$ of a run of $\mathbb{Q}(1^\eta)$ such that $\lfloor \omega \rfloor \not\models \psi$, i.e., the judge renders a false verdict.

Definition 1 (Accountability). *Let $\mathbb{Q}$ be a system of IITMs that includes a judge J and let A be a set of agents. Let Φ be an accountability property of $\mathbb{Q}$.*

We say that J ensures Φ-accountability for $\mathbb{Q}$ (or $\mathbb{Q}$ is Φ-accountable w.r.t. J) iff

1. *(fairness)* $\Pr[\mathbb{Q}(1^\eta) \mapsto \{(J : \psi) \mid \not\models \psi\}]$ *is negligible as a function in η, and*
2. *(completeness)* $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi)]$ *is negligible as a function in η.*

6. Security and Accountability of Fabric*

In this section, we analyze security goals of Fabric* using the concept of accountability. We start by formally defining *consistency*. We then propose the novel notion of *(individual) accountability w.r.t. consistency* and show that Fabric* meets this notion. The remaining security goals

from Section 3.1 are analyzed in Appendix C. We believe that all of our definitions and concepts should be easy to adapt to other (permissioned) blockchains as well.

6.1. Defining Accountability w.r.t. Consistency

Standard definitions of consistency/common prefix of blockchains have been introduced in [24], [53]. These definitions essentially state that two *honest* participants that hold a blockchain always share the same chains, except for potentially the last T blocks. We will show that Fabric* is accountable with respect to an even stronger definition of consistency: intuitively, we define consistency to hold true if, for every two participants, the chain of one participant is a prefix of the chain of the other participant. This definition is stronger in two aspects: Firstly, we do not have to allow for a potential divergence in the last T blocks. Secondly, our definition also includes dishonest participants.

In the following, we refer to what we call *reported blocks* to denote the blocks $\mathcal{B} = (j, prevHash, B, pid, s)$ that peers send to the judge $\mathcal{J}$ where j is a block number, B is a block body, *pid* is the ID of the orderer that generated the block, *prevHash* is the hash over its predecessor block, and s is the orderer's signature over the block.

In order to define consistency, it is convenient to require that a chain at least follows the basic parsing and verification rules of the blockchain protocol. We call this goal *basic correctness* (see also the remark before Definition 3), which is somewhat similar to *chain-level validity* as introduced by [9]. With this, we can now define consistency of Fabric*. We later enforce both consistency and basic correctness via accountability.

Definition 2 (Consistency of Fabric*). *Consider a run ω of the system $\mathcal{E}|\mathcal{A}|\mathbb{P}_F$, for some environment $\mathcal{E}$ and some adversary $\mathcal{A}$ that satisfies basic correctness. Let P be the set of peer identities (as specified as a parameter of $\mathcal{F}_{init}$). Let $\lfloor\omega\rfloor$ denote one point somewhere in the run ω. Let $p_1, p_2 \in P$ and let*

$$\mathcal{B}_1^i = (1, prevHash_1^i, B_1^i, pid_1^i, s_1^i), \ldots,$$
$$\mathcal{B}_{id_i}^i = (id_i, prevHash_{id_i}^i, B_{id_i}^i, pid_{id_i}^i, s_{id_i}^i)$$

be all blocks reported by p_i up to point $\lfloor\omega\rfloor$ for $i \in \{1, 2\}$ and $id_1, id_2 \in \mathbb{N}$. We say that p_1 and p_2 share a prefix at point $\lfloor\omega\rfloor$ if $B_j^1 = B_j^2$ for all $j = 1, \ldots, \min(id_1, id_2)$.

The IITM system $\mathcal{E}|\mathcal{A}|\mathbb{P}_F$ satisfies consistency in ω iff all pairs of peers $p_1, p_2 \in P$ share a prefix at every point during the run ω.

In contrast to typical definitions of consistency, our consistency definition merely requires that peers agree on the order and contents of block bodies. There are three reasons for this difference: Firstly, agreement on block bodies is already sufficient for all peers to obtain an identical ledger state, so there is no reason to require agreement on other parts. Secondly, the construction of Fabric, resp. Fabric*, allows for the same block (ID) to be created and signed by different orderers, so it is not possible to show agreement on the full blocks. Thirdly, agreement of hashes (which do not include orderer IDs and signatures and hence should still be identical for different peers) is already covered separately by the security goal hash chain integrity; if desired, one could add agreement on hashes to be part of the consistency definition – this would not change our results.

For Fabric*, we would like to show that every run provides either both basic correctness and consistency or we can hold someone accountable for misbehaving. This is captured in the notion of *(individual) accountability w.r.t. consistency*, which we define via Φ-accountability. Note that in the following definition we specify three properties, α_1, α_2, and α_3, that each cover a violation of one of the requirements of basic correctness as well as a fourth property α_4 that captures a violation of consistency. In fact, formally we define basic correctness to be satisfied for a run if it does not belong to $\alpha_1 \cup \alpha_2 \cup \alpha_3$. Also, α_4 defines exactly those runs which satisfy basic correctness but violate consistency (see Lemma 1). We remark that all properties α_1, α_2, α_3, and α_4 are defined to be disjoint sets of runs. This simplifies the security proof.

Definition 3 (Accountability w.r.t. Consistency). *We consider runs of the IITM system $\mathcal{E}|\mathcal{A}|\mathbb{P}_F$, for some environment $\mathcal{E}$ and some adversary $\mathcal{A}$. Let $P = \{p_1, \ldots, p_m\}$ be the set of peers, let $O = \{o_1, \ldots, o_l\}$ be the set of orderers, and let K be the set of Kafka brokers as specified in the parameters of $\mathcal{F}_{init}$. Let $A = P \cup O \cup \{\mathfrak{K}\}$ be the set of identities that should be held accountable, where $\mathfrak{K}$ is a special symbol that denotes the whole Kafka cluster. In the following, let $msg_{\mathfrak{K}} := (id_{\mathfrak{K}}, pid_{\mathfrak{K}}, msg, mr, mp, s_{\mathfrak{K}})$ be a Kafka message and $\mathcal{B} = (id_{\mathcal{B}}, prevHash, B, pid_{\mathcal{B}}, s_{\mathcal{B}})$ be a block. We now define the following four properties (predicates on runs):*

$\boldsymbol{\alpha_1}$ **(well-formed data):** α_1 *includes all runs where peers report data to the judge that violates any of the following four requirements:* (i) *Reported data can be parsed as a block $\mathcal{B}$ and components of the block body $\mathcal{B}$ have correct data types.* (ii) *The block body B of a reported block $\mathcal{B}$ can be parsed as a sequence of Kafka messages $msg_{\mathfrak{K}}$.* (iii) *The sequence of blocks that a single peer reports is in consecutive order without gaps (i.e., if a block with ID $id_{\mathcal{B}}$ is reported, then the next reported block has ID $id_{\mathcal{B}} + 1$) and the first block has ID 1.* (iv) *The sequence of blocks that a single peer reports contains, in the block bodies, a consecutive sequence of Kafka messages $msg_{\mathfrak{K}}$ without gaps such that the first message has ID 1.[3]*

$\boldsymbol{\alpha_2}$ **(role compliance):** α_2 *includes all runs that are not in α_1 and where a peer reports a block $\mathcal{B}$ such that* (i) $pid_{\mathcal{B}} \notin O$ *or* (ii) *the block body B includes a Kafka message $msg_{\mathfrak{K}}$ where $pid_{\mathfrak{K}} \notin K$.*

$\boldsymbol{\alpha_3}$ **(signature validity):** α_3 *includes all runs that are not in $\alpha_1 \cup \alpha_2$ and where a peer reports a block $\mathcal{B}$ such that* (i) $\text{VERIFYSIG}_{pk(pid_{\mathcal{B}})}((id_{\mathcal{B}}, prevHash, B), s_{\mathcal{B}}) = false$ *or* (ii) *the block body B includes a Kafka message $msg_{\mathfrak{K}}$ such that $\text{VERIFYMERKLEPROOF}[4]((id_{\mathfrak{K}}, pid, msg), mr, mp) = false$ or $\text{VERIFYSIG}_{pk(pid_{\mathfrak{K}})}(mr, s_{\mathfrak{K}}) = false$.*

$\boldsymbol{\alpha_4}$ **(consistency):** α_4 *contains all runs that are not in $\bigcup_{i=1}^{3} \alpha_i$ and where two peers report blocks*

$$\mathcal{B} = (id_{\mathcal{B}}, prevHash, B, pid_{\mathcal{B}}, s_{\mathcal{B}}) \text{ and}$$
$$\overline{\mathcal{B}} = (id_{\mathcal{B}}, \overline{prevHash}, \overline{B}, \overline{pid_{\mathcal{B}}}, \overline{s_{\mathcal{B}}})$$

such that $B \neq \overline{B}$.

---

3. We note that well-formed data includes the security goal of *no skipping* as defined by Androulaki et al. [2].

Overall, we set $\alpha := \bigcup_{i=1}^{4} \alpha_i$. We define an accountability constraint $\mathcal{C}$ for α as follows:

$\mathcal{C} := (\alpha \Rightarrow \mathsf{dis}(p_1)|\ldots|\mathsf{dis}(p_m)|\mathsf{dis}(o_1)|\ldots|\mathsf{dis}(o_l)|\mathsf{dis}(\mathfrak{K}))$. *That is, if basic correctness or consistency is violated, then the judge must blame at least one individual from the set A of participants. We define the accountability property $\Phi := \{\mathcal{C}\}$.*

We call $\mathbb{P}_\mathsf{F}$ (individually) accountable w.r.t. to consistency if for all environments $\mathcal{E}$ and adversaries $\mathcal{A}$ we have that $\mathcal{E}|\mathcal{A}|\mathbb{P}_\mathsf{F}$ achieves Φ-accountability.

In Section 6.3, we prove that $\mathbb{P}_\mathsf{F}$ provides (individual) accountability w.r.t. to consistency.

Trivially, the above definition of accountability w.r.t. consistency indeed covers the expected security goals of basic correctness and consistency, i.e., runs where one of the properties is violated belong to α.

Lemma 1. *Let $\mathcal{E}$ be an environment and $\mathcal{A}$ adversary. Let Ω be the set of runs of $\mathcal{E}|\mathcal{A}|\mathbb{P}_\mathsf{F}$ that satisfy basic correctness and consistency. Let α be as defined above.*

Then it holds true that α is the complement of Ω.

6.2. The Judging Procedure

Before we can prove that Fabric* achieves individual accountability w.r.t. consistency, we have to specify the judging procedure, i.e., by what evidence the judge blames which parties. Before we specify the judging procedure, we discuss what the judge resembles in reality.

What the judge models in reality. Recall that, in our model, (honest) peers report blocks that they accept to the judge, who can then use this information to render verdicts. In reality, this corresponds to the situation where an error in the chain(s) of one or more peers was found, and those chains are then given to other parties (e.g., other peers or independent third parties) to determine who caused this error.

For example, if a client suspects that consistency is violated, say, because the data received upon a request from one peer is suspiciously inconsistent with the data received from another peer, the client would notify the organizations that take part in the Fabric* channel about the inconsistency. The organizations would ==inspect== the chains of both peers following the judging procedure to determine whether a party misbehaved. If one of the peers refuses to provide the requested data he is trivially guilty of disrupting the protocol. Note that this process indeed requires basic correctness: a peer should not be able to escape a verdict by providing malformed evidence. In addition to clients suspecting problems, peers might (be required to) exchange data from time to time to see whether there are inconsistencies.

Once an error has been detected and the judge has identified a misbehaving participant, organizations then agree out of band on the "correct" chain (e.g., by dropping all blocks after an inconsistency) and continue from there. Note that individual accountability acts as a strong deterrence and hence errors such as inconsistent chains should generally not occur in practice.

Specification of the judging procedure. At the start of a run, the judge J obtains basic information about the Fabric* channel from $\mathcal{F}_{\mathrm{init}}$, including the sets of identities of peers, orderers, and Kafka brokers. During the run, (honest) peers immediately report all blocks that they accept to J. The judge J first stores those reported blocks as evidence for verdicts. If no verdict has been issued so far, the judge then continues to check whether he has to issue a verdict due to the newly received block:

1. Basic Correctness. The judge starts by checking whether basic correctness has been violated, i.e., whether the current run is in $\bigcup_{i=1}^{3} \alpha_i$. Note that this is indeed possible as all of those properties are defined over the evidence/blocks submitted to the judge. Further note that, according to their specification, honest peers also check those properties of blocks accordingly and will neither accept nor report a block that violates basic correctness (for their own chain). Thus, if a block causes a violation, then the judge blames the peer that reported that block.

2. Consistency. If basic correctness still holds true in the current run, then the judge checks whether consistency has been violated, i.e., whether the current run is in α_4. As defined, this is the case iff there are two reported blocks $\mathcal{B}_1, \mathcal{B}_2$ with the same ID but differing bodies (generated by two, not necessarily different orderers). The verdict is then computed as follows: The judge first checks whether there are two different Kafka messages with the same ID (and valid signatures) in $\mathcal{B}_1$ and $\mathcal{B}_2$. If so, then this inconsistency was caused by the Kafka cluster who is supposed to generate unique messages per ID, and hence the Kafka cluster is blamed. Otherwise, (at least) one of the orderers has cut the chain at a wrong point, i.e., the block ends too early or too late. To find the culprit, the judge checks whether $\mathcal{B}_1$ follows the rules of the CUTBLOCK algorithm: That is, $\mathcal{B}_1$ must *(i)* contain the maximum number of Kafka messages such that none of those messages is a valid cut message, or *(ii)* contain at most the maximum number of Kafka messages as well as a single valid cut message that is the last of the Kafka messages. If this check fails, the orderer who generated $\mathcal{B}_1$ is blamed. Otherwise, the same checks are performed for $\mathcal{B}_2$, potentially resulting in a verdict on the orderer that generated $\mathcal{B}_2$.

6.3. Proving Accountability

In this section, we present our main theorem: Fabric* achieves (individual) accountability w.r.t. consistency, i.e., Φ-accountability. Together with Lemma 1 this theorem says that Fabric* achieves basic correctness and consistency in the absence of a verdict of the judge J. Should basic correctness or consistency be violated at some point, the judge would rightly and individually blame at least one party.

Due to lack of space, we provide the formal analysis of the remaining security goals from Section 3.1 for Fabric*, namely, *chain-growth*, *liveness*, *hash chain integrity*, and *no creation* in Appendix C.

Theorem 2 (Fabric* achieves accountability w.r.t. consistency)**.** *Let $\mathbb{P}_F$ be the Fabric* protocol with arbitrary parameters for $\mathcal{F}_{\mathrm{init}}$ and the judge J defined as in Section 6.2. Then, it holds true that $\mathbb{P}_F$ achieves individual accountability w.r.t. consistency.*

We briefly discuss the core ideas of the proof, with the full proof given in Appendix B.

Proof Idea. We have to show two properties, fairness and completeness.

Fairness. Firstly, we prove that the judge J does not render a wrong verdict in a run of $\mathcal{E} \,|\, \mathcal{A} \,|\, \mathbb{P}_{\mathrm{F}}$, i.e., verdicts must evaluate to `true` (except for a negligible set of runs). This mostly follows from the reasoning given in Section 6.2 and the fact that, by the definition of $\mathcal{F}_{\mathrm{sig}}$, an adversary cannot forge signatures for honest parties. Note that here we benefit from the construction of the block cutting algorithm CUTBLOCK, which allows for deterministically recomputing a block from the Kafka message stream contained in such a block. Thus, all blocks from an honest orderer will be the same as the re-computation of the judge. Hence, honest peers and orderers will not be blamed by accident.

There is, however, one special case that requires further attention: the Kafka cluster does not directly sign messages, but rather builds a Merkle tree and then signs its root. Thus, if an adversary finds a different message body and Merkle proof with the same Merkle root or if he can create a different Merkle proof for the same message body and Merkle root, he can re-use a signature of an honest Kafka cluster to create inconsistent Kafka stream messages and thus cause an incorrect verdict. However, one can reduce both cases to a collision in the random oracle (see, e.g., [17]). As collisions have only negligible probability for polynomial time systems, the judge issues a wrong verdict with at most negligible probability.

Completeness. We have to prove that if a run of $\mathcal{E} \,|\, \mathcal{A} \,|\, \mathbb{P}_{\mathrm{F}}$ belongs to $\alpha = \bigcup_{i=1}^{4} \alpha_i$, then the judge J renders a verdict that blames at least one of the peers, orderers, or the Kafka cluster. Observe that, according to the judging procedure, all verdicts of the judge blame an individual party or the Kafka cluster directly.

As explained in Section 6.2, if a run is in $\bigcup_{i=1}^{3} \alpha_i$, then the judge will notice this and blame the peer that caused the violation. If the run is in α_4, i.e., consistency is violated while basic correctness holds true, then either the judge blames the Kafka cluster for distributing different messages, or we know the Kafka message stream in one conflicting block is a (real) prefix of the Kafka message stream of the other block. This essentially follows from basic correctness (in particular, this is where we need that there are no gaps in the Kafka message streams contained in the chains) and the fact that this is the first inconsistent block ID. Hence, the blocks are inconsistent only because they end at different points of the Kafka message stream. By definition of CUTBLOCK, both orderers should have cut the stream at the same endpoint (as they agree on the same prefix of the Kafka message stream). This implies that the final check of the judge fails for at least one of the orderers, who is then blamed. $\qquad\square$

All of our security results are shown for an ideal signature functionality $\mathcal{F}_{\mathrm{sig}}$. In Appendix D, we show that all results carry over if $\mathcal{F}_{\mathrm{sig}}$ is replaced by its realization $\mathcal{P}_{\mathrm{sig}}$ (cf. [46]) where signing is done using an EUF-CMA secure signature scheme (*existential unforgeability under chosen message attacks*). While this result uses that $\mathcal{P}_{\mathrm{sig}}$ realizes $\mathcal{F}_{\mathrm{sig}}$ in the usual UC sense (see e.g., [46]), the result does not follow immediately from UC composition theorems.

7. Related Work

Most of the established literature on blockchain security (see, e.g., [8], [9], [11], [19], [24], [35], [36], [53]) does not use accountability or the closely related concept of covert adversaries [5], [7]. While there are several works that mention accountability as a concept for blockchains [12], [14], [18], [22], [23], [33], [34], [54], none of them formalizes let alone proves accountability. Perhaps the closest to our work is the work by Karame et al. [34], who suggest countermeasures against double-spending in Bitcoin by identifying misbehaving parties. While this work uses the underlying intuition of accountability — namely identifying a misbehaving participant — to show a formal statement about double-spending, they do not use the concept of accountability itself in a formal way.

As discussed in detail in Section 3.3, Haeberlen et al. [28] introduce PeerReview to add accountability on top of an arbitrary (deterministic) protocol. PeerReview has several drawbacks that typically make solutions tailored to the protocol at hand preferable (cf. Section 3.3). Furthermore, Haeberlen et al. do not provide a formal security proof for their system.

8. Conclusion

In this paper, we put forward the formal treatment of accountability for (permissioned) blockchains and distributed ledgers. As an orthogonal property to strict security notions, on the one hand, accountability can serve as an additional security layer for blockchains and distributed ledgers. On the other hand, as a stand-alone property in its own right, accountability opens up alternative and interesting ways of analyzing and constructing secure blockchains and distributed ledgers with other, partly weaker assumptions (e.g., no eventual message delivery and no honest majorities) and security goals that in some aspects are stronger.

We successfully illustrated this approach by a rigorous analysis of an (improved) instance of Hyperledger Fabric, a very successful and widely used blockchain in the industry. In addition to the formal treatment of accountability, given the wide spread use of this blockchain, the results obtained in our case study are of independent interest. We expect this approach to also prove useful for the analysis of a wide range of existing (permissioned) blockchains and distributed ledgers, such as Tendermint-based blockchains [1] and Corda [13].

Altogether, the formal treatment of accountability is a powerful tool that opens up new options for blockchain (security) research. Our approach also provides a formal foundation for formalizing and proving the informal accountability claims that are commonly found in many blockchain specifications.

Acknowledgements

References

[1] All In Bits, Inc., "Blockchain Consensus - Tendermint," https://tendermint.com/, 2019, (Accessed on 06/05/2019).

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 2018, pp. 30:1–30:15.

[3] Apache Software Foundation, "Apache Kafka," https://kafka.apache.org/, 2017, (Accessed on 04/01/2019).

[4] ——, "Apache ZooKeeper," https://zookeeper.apache.org/, 2018, (Accessed on 04/01/2019).

[5] G. Asharov and C. Orlandi, "Calling Out Cheaters: Covert Security with Public Verifiability," in *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7658. Springer, 2012, pp. 681–698.

[6] N. Asokan, V. Shoup, and M. Waidner, "Asynchronous protocols for optimistic fair exchange," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*. IEEE Computer Society, 1998, pp. 86–99.

[7] Y. Aumann and Y. Lindell, "Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries," in *Proceedings of the 4th Theory of Cryptography Conference,(TCC 2007)*, ser. Lecture Notes in Computer Science, S. P. Vadhan, Ed., vol. 4392. Springer, 2007, pp. 137–156.

[8] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas, "Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 2018, pp. 913–930.

[9] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a Transaction Ledger: A Composable Treatment," in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part 1*, ser. Lecture Notes in Computer Science, vol. 10401. Springer, 2017, pp. 324–356.

[10] C. Baum, I. Damgård, and C. Orlandi, "Publicly Auditable Secure Multi-Party Computation," in *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, ser. Lecture Notes in Computer Science, vol. 8642. Springer, 2014, pp. 175–196.

[11] I. Bentov, R. Pass, and E. Shi, "Snow White: Provably Secure Proofs of Stake," *IACR Cryptology ePrint Archive*, vol. 2016, p. 919, 2016.

[12] A. Boudguiga, N. Bouzerna, L. Granboulan, A. Olivereau, F. Quesnel, A. Roger, and R. Sirdey, "Towards Better Availability and Accountability for IoT Updates by Means of a Blockchain," in *2017 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2017, Paris, France, April 26-28, 2017*. IEEE, 2017, pp. 50–58.

[13] M. Brown and R. G. Brown, "Corda: A distributed ledger," https://www.r3.com/reports/corda-technical-whitepaper/, 2019, (Accessed on 11/11/2019).

[14] V. Buterin and V. Griffith, "Casper the Friendly Finality Gadget," *CoRR*, vol. abs/1710.09437, 2017.

[15] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*. IEEE Computer Society, 2001, pp. 136–145.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[17] C. Coronado, "On the security and the efficiency of the Merkle signature scheme," *IACR Cryptology ePrint Archive*, vol. 2005, p. 192, 2005.

[18] G. D'Angelo, S. Ferretti, and M. Marzolla, "A Blockchain-based Flight Data Recorder for Cloud Accountability," in *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems, CRYBLOCK@MobiSys 2018, Munich, Germany, June 15, 2018*. ACM, 2018, pp. 93–98.

[19] B. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain," in *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 10821. Springer, 2018, pp. 66–98.

[20] M. del Castillo, "Blockchain 50: Billion dollar babies," https://www.forbes.com/sites/michaeldelcastillo/2019/04/16/blockchain-50-billion-dollar-babies/, 2019, (Accessed on 05/01/2019).

[21] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 2017, pp. 1085–1100.

[22] Ethereum Foundation, "Ethereum enterprise," https://www.ethereum.org/enterprise/, 2019, (Accessed on 11/13/2019).

[23] E. Funk, J. Riddell, F. Ankel, and D. Cabrera, "Blockchain technology: a data framework to improve validity, trust, and accountability of information exchange in health professions education," *Academic Medicine*, vol. 93, no. 12, pp. 1791–1794, 2018.

[24] J. A. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin Backbone Protocol: Analysis and Applications," in *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9057. Springer, 2015, pp. 281–310.

[25] P. Gazi, A. Kiayias, and A. Russell, "Stake-Bleeding Attacks on Proof-of-Stake Blockchains," *IACR Cryptology ePrint Archive*, vol. 2018, p. 248, 2018.

[26] S. Goldwasser, S. Micali, and R. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 281–308, 1988.

[27] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "FastFabric: Scaling Hyperledger Fabric to 20, 000 Transactions per Second," in *IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019*. IEEE, 2019, pp. 455–463.

[28] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: practical accountability for distributed systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 175–188.

[29] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*. USENIX Association, 2010.

[30] Hyperledger Project, "fabricdocs: The ordering service," https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html, 2018, (Accessed on 07/24/2019).

[31] ——, "Hyperledger Fabric," https://www.hyperledger.org/projects/fabric, 2018, (Accessed on 03/02/2019).

[32] ——, "fabricdocs: Architecture origins," https://hyperledger-fabric.readthedocs.io/en/latest/arch-deep-dive.html, 2019, (Accessed on 01/29/2019).

[33] IBM, "Hyperledger Fabric: What you need to know about the framework that powers IBM Blockchain," https://www.ibm.com/blogs/blockchain/2018/08/hyperledger-fabric-what-you-need-to-know-about-the-framework-that-powers-ibm-blockchain/, 8 2018, (Accessed on 11/11/2019).

[34] G. O. Karame, E. Androulaki, M. Roeschlin, A. Gervais, and S. Capkun, "Misbehavior in Bitcoin: A Study of Double-Spending and Accountability," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 2:1–2:32, 2015.

[35] T. Kerber, M. Kohlweiss, A. Kiayias, and V. Zikas, "Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake," *IACR Cryptology ePrint Archive*, vol. 2018, p. 1132, 2018.

[36] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol," in *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 10401.   Springer, 2017, pp. 357–388.

[37] R. Küsters, "Simulation-Based Security with Inexhaustible Interactive Turing Machines," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*.   IEEE Computer Society, 2006, pp. 309–320, see http://eprint.iacr.org/2013/025/ for a full and revised version.

[38] R. Küsters and M. Tuengerthal, "Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation," Cryptology ePrint Archive, Tech. Rep. 2008/006, 2008, available at http://eprint.iacr.org/2008/006 with an updated version from 2013.

[39] R. Küsters and J. Müller, "Cryptographic Security Analysis of E-voting Systems: Achievements, Misconceptions, and Limitations," in *Electronic Voting - Second International Joint Conference, E-Vote-ID 2017, Bregenz, Austria, October 24-27, 2017, Proceedings*, ser. Lecture Notes in Computer Science, vol. 10615.   Springer, 2017, pp. 21–41.

[40] R. Küsters, J. Müller, E. Scapin, and T. Truderung, "sElect: A Lightweight Verifiable Remote Voting System," in *IEEE 29th Computer Security Foundations Symposium (CSF 2016)*.   IEEE Computer Society, 2016, pp. 341–354.

[41] R. Küsters, D. Rausch, and M. Simon, "Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric," Cryptology ePrint Archive, Tech. Rep. 386, 2020.

[42] R. Küsters and T. Truderung, "Security Analysis of Re-Encryption RPC Mix Nets," in *IEEE 1st European Symposium on Security and Privacy (EuroS&P 2016)*.   IEEE Computer Society, 2016, pp. 227–242.

[43] R. Küsters, T. Truderung, and A. Vogt, "Accountability: Definition and Relationship to Verifiability," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*.   ACM, 2010, pp. 526–535, the full version is available at http://eprint.iacr.org/2010/236.

[44] R. Küsters, T. Truderung, and A. Vogt, "Clash Attacks on the Verifiability of E-Voting Systems," in *33rd IEEE Symposium on Security and Privacy (S&P 2012)*.   IEEE Computer Society, 2012, pp. 395–409.

[45] ——, "Formal Analysis of Chaumian Mix Nets with Randomized Partial Checking," in *35th IEEE Symposium on Security and Privacy (S&P 2014)*.   IEEE Computer Society, 2014, pp. 343–358.

[46] R. Küsters and M. Tuengerthal, "Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*.   IEEE Computer Society, 2008, pp. 270–284, the full version is available at https://eprint.iacr.org/2008/006.

[47] R. Küsters, M. Tuengerthal, and D. Rausch, "The IITM Model: a Simple and Expressive Model for Universal Composability," Cryptology ePrint Archive, Tech. Rep. 2013/025, 2013, available at http://eprint.iacr.org/2013/025.

[48] B. W. Lampson, "Computer Security in the Real World," *IEEE Computer*, vol. 37, no. 6, pp. 37–46, 2004.

[49] J. Liedtke, R. Küsters, J. Müller, D. Rausch, and A. Vogt, "Ordinos: A Verifiable Tally-Hiding Electronic Voting Protocol," in *IEEE 5th European Symposium on Security and Privacy (EuroS&P 2020)*. IEEE Computer Society, 2020, to appear.

[50] R. C. Merkle, "A Certified Digital Signature," in *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, ser. Lecture Notes in Computer Science, vol. 435. Springer, 1989, pp. 218–238.

[51] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, jan. 18, 2019 https://bitcoin.org/bitcoin.pdf.

[52] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. USENIX Association, 2014, pp. 305–319.

[53] R. Pass, L. Seeman, and A. Shelat, "Analysis of the Blockchain Protocol in Asynchronous Networks," in *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 10211, 2017, pp. 643–673.

[54] R3, "R3 corda master documentation," https://docs.corda.net/head/corda-network/governance-structure.html, 2019, (Accessed on 11/13/2019).

[55] J. Rao, "Intra-cluster replication in apache kafka — linkedin engineering," https://engineering.linkedin.com/kafka/intra-cluster-replication-apache-kafka, 2 2013, (Accessed on 11/28/2019).

[56] H. Sukhwani, J. M. Martínez, X. Chang, K. S. Trivedi, and A. Rindos, "Performance Modeling of PBFT Consensus Process for Permissioned Blockchain Network (Hyperledger Fabric)," in *36th IEEE Symposium on Reliable Distributed Systems, SRDS 2017, Hong Kong, Hong Kong, September 26-29, 2017*.   IEEE Computer Society, 2017, pp. 253–255.

[57] M. Szydlo, "Merkle Tree Traversal in Log Space and Time," in *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3027.   Springer, 2004, pp. 541–554.

[58] The Linux Foundation, "The Linux Foundation," https://www.linuxfoundation.org/, 2019, (Accessed on 04/01/2019).

[59] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a Replicated Logging System with Apache Kafka," *PVLDB*, vol. 8, no. 12, pp. 1654–1655, 2015.

[60] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," https://gavwood.com/paper.pdf, 2014, (Accessed on 01/18/2019).

[61] J. Zhou and D. Gollmann, "A fair non-repudiation protocol," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*.   IEEE Computer Society, 1996, pp. 55–61.

Appendix A.
Adding Signatures to Kafka

In this section, we detail our first change to Kafka for Fabric* that was briefly discussed in Section 3.3.

Recall that, on the input side, a Kafka leader gets requests to add messages to the stream. On the output side, orderers and other Kafka brokers pull new messages from the Kafka leader. Now, for efficiency reasons, instead of having a Kafka leader sign each message it puts into the message stream, we use Merkle trees [50] as follows. The Kafka leader collects incoming messages until the first request for delivering new messages arrives. For all newly collected messages, the Kafka leader then orders and enumerates the messages, forms a Merkle tree with theses messages, and signs the root of the tree. More specifically, the resulting messages are of the following form: $(id, pid, msg, mr, mp, s)$ where id is the message

ID (i.e., the counter of the enumeration), *msg* is the message payload (usually a transaction commitment), *pid* is the identity of the Kafka leader, *mr* is the root of the Merkle tree, *mp* is a Merkle proof that proves that (id, pid, msg) is a leaf of the Merkle tree with root *mr*, and $s = \text{SIGN}_{sk(pid)}(mr)$ is a signature of the root *mr* from *pid*. By this, only one signature is necessary for all messages in one Merkle tree. These messages are put into the message stream in the appropriate order and given to the entity who requested new messages. For messages that have already been ordered and delivered once before (and hence, for which a Merkle tree has already been formed), the leader simply reuses the messages that were sent before (including their Merkle root, Merkle proof, and signature).

We note that we include the Merkle proof in every message since messages from the same Merkle tree might be part of multiple different blocks.

Appendix B.
Accountability of Fabric* w.r.t. the Judge - Proof

In this section, we present the full proof of accountability w.r.t. consistency for the Fabric* (cf. Theorem 2).

Proof. Let $\mathbb{P}_F$ be the Fabric* protocol as required by the theorem, and let $\mathcal{E}$ be an arbitrary environment and $\mathcal{A}$ be an arbitrary adversary.[6] We have to show that $\mathcal{E} \,|\, \mathcal{A} \,|\, \mathbb{P}_F$ provides Φ-accountability, where Φ is the one from Definition 3.

Let us start with some general observations. We know that signatures of honest participants cannot be forged by the definition of $\mathcal{F}_{\text{sig}}$. Furthermore, if we look at the common case occurring in runs where first a signature in a block is checked by an (honest) peer, then this block is reported, and finally the signatures in that block are checked again by the judge, then we have that the signature check outputs the same for all three points in time. This is because the validity check of $\mathcal{F}_{\text{sig}}$ is a deterministic function over the input (which is the same message) and the internal state of $\mathcal{F}_{\text{sig}}$ (which does not change between those three points; in particular, the adversary is not activated in-between and cannot change the corruption status).

We now show that $\mathcal{E} \,|\, \mathcal{A} \,|\, \mathbb{P}_F$ provides Φ-accountability by proving fairness and completeness.

Fairness. We have to show that $\Pr\left[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi)\right]$ is a negligible function in η, i.e., the judge's verdicts evaluate to false (when they are issued) in at most a negligible amount of cases.

Observe that, by the definition of J, all verdicts are of the form $\text{dis}(p)$ for a participant p of the protocol. Hence, it suffices to argue that, for every role r, the probability that J renders the verdict $\text{dis}(p)$ for some honest party p of role r is negligible (we denote this property by (*) in what follows); this then immediately implies fairness.

Clients: There is never a verdict that contains a client. Hence, (*) trivially holds true for clients.

6. As mentioned in Section 4, runtimes are defined in such a way that $\mathcal{E} \,|\, \mathcal{A} \,|\, \mathbb{P}_F$ runs in polynomial time (see [47]).

Peers: Let p be an honest peer (in some run). The judge J outputs $\text{dis}(p)$ as verdict if one of the basic correctness properties $\alpha_1, \dots, \alpha_3$ is violated by a block $\mathcal{B}_1$ sent by p. Observe that peers have a direct (authenticated) connection to the judge which guarantees that no other party can report data in the name of p. Hence, if the judge blames p due to a block $\mathcal{B}_1$ that violates basic correctness, then this block was submitted by p itself. Since p is honest, he will only submit blocks that pass all of the following checks: *(i)* The block is well-formed (α_1), *(ii)* The block including the Kafka messages satisfies role compliance (α_2), *(iii)* and contains valid orderer signatures, Kafka broker signatures, and Merkle proofs (α_3). By this, we know that p never reports a block that violates either of the properties $\alpha_1, \dots, \alpha_3$ and thus $\text{dis}(p)$ is never part of a verdict. This implies (*) for peers.

Orderers: Let o be an honest orderer (in some run). This implies that signatures of o are unforgeable. The judge can potentially issue a verdict $\text{dis}(o)$ only for the property α_4. That is, $\text{dis}(o)$ can only be part of a verdict if there are two inconsistent blocks $\mathcal{B}_1, \mathcal{B}_2$ reported by peers such that at least one of the blocks has a valid signature from o. If there are such two conflicting blocks, then exactly one of them was created by o (as signatures are unforgeable and honest orderers create unique blocks for each block ID). W.l.o.g., let $\mathcal{B}_1$ be the block created by o. Now, J outputs the verdict $\text{dis}(o)$ only if $\mathcal{B}_1$ was not created by using CUTBLOCK, i.e., $\mathcal{B}_1$ should contain the maximum number of Kafka messages that are allowed per block and no valid cut messages before the end, or it should end with the first (valid) cutting message contained in that block while not exceeding the maximum number of messages. Since o is honest, the block $\mathcal{B}_1$ exactly follows those rules and hence o will not be blamed. This implies (*) for orderers.

Kafka Cluster: Let $\mathfrak{K}$ be an honest Kafka cluster (in some run). In particular, this implies that the signature keys of all Kafka Brokers in $\mathcal{F}_{\text{sig}}$ are still uncorrupted and hence signatures in the name of any broker/the whole cluster cannot be forged. The judge only includes $\text{dis}(\mathfrak{K})$ in a verdict if he finds two Kafka messages with the same ID, different message bodies, valid Merkle proofs, and valid signatures on the Merkle roots. Since signatures are unforgeable and the honest Kafka cluster never outputs different messages with the same ID, the only way for this to occur is if the adversary manages to re-use a (correctly) signed Merkle root (either for a different message, or for a different proof on the same message). As, e.g., Coronado [17] showed, such a successful forgery of a Merkle proof can be reduced to the probability of a collision in the random oracle. Since we are using a random oracle that outputs hashes with length η (where η is the security parameter) and the whole system runs in polynomial time, the probability for collisions is negligible. Hence, $\text{dis}(\mathfrak{K})$ is part of a verdict only in a negligible number of cases, which gives (*).

Completeness. For completeness, again observe that all verdicts of a judge are of the form $\text{dis}(p)$ (where p is a peer, an orderer, or the Kafka cluster). Note that those verdicts trivially imply one of the verdicts of the accountability constraint $\mathcal{C}$ (which essentially states that

there is at least one individual participant/the cluster that was at fault). Hence, to show that $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \Phi)]$ is negligible, we only have to show that the judge indeed outputs some verdict for all runs in $\bigcup_{i \in \{1,\dots,4\}} \alpha_i$ (except for potentially a negligible probability). We do so by showing that for all $i \in \{1,\dots,4\}$ the probability that a run is in α_i but the judge does not output a verdict (written $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \alpha_i)]$ in what follows) is negligible. Completeness then directly follows.

On $\alpha, i = 1, \dots, 3$: Each of the properties is defined in a deterministic way over the evidence that the judge obtains. Note in particular that, as discussed above, the validity of signatures is identical at the point of a run where a block is reported and at the point where the judge checks the block. By its definition, the judge checks precisely these three properties and, if one of them is met, he always outputs a verdict that blames the peer that submitted the faulty block. Thus, $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \alpha_i)] = 0$, $i \in \{1, \dots, 3\}$.

On α_4: In a run of α_4, there are two conflicting blocks $\mathcal{B}_1, \mathcal{B}_2$ with the same ID but differing block bodies (and arbitrary hashes, orderer IDs, and signatures) reported by some peers. Just as for the other properties, the judge can (and does) indeed check this property as it is defined deterministically over the evidence reported to the judge. However, unlike for the other properties, in this case there is no "default" output that always blames some party. Instead, the judge performs a series of checks and might potentially blame the Kafka cluster or an orderer. Thus, we have to show in what follows that at least one of the checks results in a verdict.

Let o_1 and o_2 be the orderers that signed blocks $\mathcal{B}_1$, resp. $\mathcal{B}_2$, reported by peer p_1, resp. p_2. The judge first checks whether there are two different Kafka messages with the same ID in the block bodies of $\mathcal{B}_1$ and $\mathcal{B}_2$; if so, then the Kafka cluster is blamed. Otherwise, we can conclude that the Kafka message stream included in one block is a prefix of the Kafka message stream of the other: Since α_4 is disjoint from $\alpha_1, \dots, \alpha_3$, we know that the inconsistent chains of p_1 and p_2 contain a sequential (i.e., without gaps) message stream starting from ID 1. Since the chains have been consistent up to the block $\mathcal{B}_1$ and $\mathcal{B}_2$, we hence know that both blocks start with the same Kafka message ID. As the blocks $\mathcal{B}_1$ and $\mathcal{B}_2$ also contain no gaps and agree on the contents of all Kafka messages with the same ID, their inconsistency can only be due to differing lengths. That is, w.l.o.g. $\mathcal{B}_1$ ends at an earlier point of the Kafka message stream than $\mathcal{B}_2$.

The judge now checks whether $\mathcal{B}_1$ was correctly cut from the message stream contained within. If not, then o_1 is blamed. Otherwise, the cut of o_1 was correct and we have to argue that o_2 will be blamed for an incorrect cut. There are two cases:

- The block $\mathcal{B}_1$ of o_1 was correct because the block ends with the first (valid) cut message and does not exceed the maximum number of messages. Since the block $\mathcal{B}_2$ also contains all Kafka messages from $\mathcal{B}_1$ but ends at a later point, there is a cut message in the middle of $\mathcal{B}_2$ that was ignored. Hence, the block $\mathcal{B}_2$ was cut incorrectly and thus o_2 is blamed.
- The block $\mathcal{B}_1$ of o_1 was correct because it does not contain a (valid) cut message and it contains the maximum number of transactions. Since the block $\mathcal{B}_2$ is

longer than $\mathcal{B}_1$, it contains more than the maximum number of transactions. Hence, the block $\mathcal{B}_2$ was thus cut incorrectly and thus o_2 is blamed.

We conclude that the judge always outputs a verdict for runs in α_4, i.e., $\Pr[\mathbb{Q}(1^\eta) \mapsto \neg(J : \alpha_4)] = 0$. This concludes the proof.

$\square$

Appendix C.
Definition and Analysis of Further Goals of Fabric*

We have already formally shown in Section 6 that Fabric* is *accountable with respect to consistency* (which includes *basic correctness* and in particular *no skipping*). In this section, we first summarize and briefly discuss our results for the remaining security goals from Section 3.1 for the Fabric* protocol, namely, *chain-growth*, *liveness*, *hash chain integrity*, and *no creation*. Then, we provide full details on the analysis of these goals.

1. Chain-growth and Liveness. In general, one cannot prove chain-growth and liveness in a network with message loss for any blockchain. Hence, we adapt our model of Fabric* to include a round-based network model with bounded message delay, as it is commonly used also for other blockchains. However, even when assuming such a network model, both goals can still be broken due to Fabric*'s centralized structure: malicious orderers and in particular a malicious Kafka cluster can simply drop messages instead of forwarding them. Furthermore, culprits also cannot be held individually accountable for breaking these goals (based on data in the blockchain). Thus, formally one can only show that both goals hold true if *(i)* all peers, *(ii)* all orderers, and *(iii)* the Kafka cluster are honest.

We note that this formal lack of chain-growth and liveness is not that devastating in practice, unlike in many other common blockchains: Firstly, for chain-growth this has already been discussed in Section 3.1. Secondly, as shown above, neither liveness nor chain-growth are necessary to guarantee consistency, unlike in, e.g., blockchains relying on the longest chain rule. Thirdly, all participants of Fabric, resp. Fabric* know each other. So in a real world execution of Fabric*, if peers, orderers, or the Kafka cluster do not receive updates from the channel, operators will report this issue to the organizations of the channel. The organizations can then monitor individual connections to identify where messages are dropped and whether this was caused by one of the participants or an unreliable network.

2. Hash chain integrity. Intuitively, hash chain integrity states that for all pairs of peers p_1, p_2, if p_1 accepts a block $\mathcal{B}$ with ID id and p_1 has accepted a block $\mathcal{B}'$ with ID $id - 1$, then the predecessor hash contained in $\mathcal{B}$ is the hash of $\mathcal{B}'$. Analogously to consistency (which is a necessary requirement for hash chain integrity to hold true), we can enforce hash chain integrity for all peers, including dishonest ones, via accountability.

3. No creation. Fabric* provides a slight variant of *no creation* which we call *chain soundness*. This goal holds true for all honest peers and can be enforced for malicious peers via accountability.

C.1. Chain-growth and liveness

As already mentioned, it is well-known that one can obtain neither chain-growth nor liveness (as, e.g., defined by Garay et al. [24]) in a fully asynchronous network with message loss. Unlike for consistency, one also cannot use accountability to enforce both goals as, even if all participants are honest, the network is still able to drop messages.

Typically, chain-growth and liveness are shown for blockchains by assuming a network with bounded message delay where messages cannot be dropped. However, even using this assumption, formally Fabric* does not provide any of the two goals in general: A corrupted Kafka cluster can simply drop incoming messages, thus preventing chain-growth and violating liveness. Furthermore, formally it is still not possible to address this issue using (individual) accountability: If a transaction is not included in the Kafka message stream, this can be either due to a malicious Kafka cluster dropping the message or due to malicious orderers that did not forward the message to the cluster in the first place. As already briefly discussed in the main part, in reality, however, if messages keep being dropped, one can of course investigate the problem since all participants of Fabric* know each other.

One can only formally show that chain-growth and liveness hold true for Fabric* in a network with bounded message delay if orderers, peers, and the Kafka cluster are honest. On a technical level, to show this result one has to adapt our model from Section 4 by routing all network communication via a round based ideal broadcast or diffuse channel with Δ-delays $\mathcal{F}_\Delta$ (cf. [24]). In $\mathcal{F}_\Delta$, the adversary is allowed to delay a message for at most $\Delta \in \mathbb{N}$ rounds, but he cannot drop or change any messages.

Proposition 3 (Chain-growth and Liveness of Fabric* - informal). *Let $\mathbb{P}_F$ be the Fabric* protocol where network traffic is sent and received via $\mathcal{F}_\Delta$. Let $\mathcal{E}$ and $\mathcal{A}$ be an arbitrary environment and an arbitrary adversary for $\mathbb{P}_F$ such that they do not corrupt any peer, orderer, or the Kafka cluster, resp. their instances at $\mathcal{F}_{\text{sig}}$. Then $\mathbb{Q}$ achieves chain-growth and liveness.*

Proof. The proof for both goals quite easily follows from the construction of $\mathcal{F}_\Delta$ and $\mathbb{P}_F$ as messages are eventually delivered after Δ rounds. □

C.2. Hash Chain Integrity

Formally, we define hash chain integrity as follows:

Definition 4 (Hash chain integrity). *Consider a run ω of the system $\mathcal{E}|\mathcal{A}|\mathbb{P}_F$, for some environment $\mathcal{E}$ and some adversary $\mathcal{A}$, that satisfies basic correctness.*

Let $\mathcal{B}_1 = \langle id_1, prevHash_1, B_1, pid_1, s_1 \rangle$ be a reported block in ω. We say that $\mathcal{B}_1$ violates hash chain integrity if there is a reported block $\mathcal{B}_2 = \langle id_2, prevHash_2, B_2, pid_2, s_2 \rangle$ in ω such that $id_1 = id_2 + 1$ and $prevHash_1 \neq H(id_2, prevHash_2, B_2)$, where H is a hash function. In addition, we also say that a block $\langle 1, prevHash, B, pid, s \rangle$ violates hash chain integrity if $prevHash \neq H(S_{init})$ (where S_{init} is the initial state modeling the genesis block).

The IITM system $\mathcal{E}|\mathcal{A}|\mathbb{P}_F$ satisfies hash chain integrity in ω if there is no reported block that violates hash integrity.

We can show that Fabric* provides individual accountability w.r.t. hash chain integrity. Formally, this notion is defined by extending the definition of accountability w.r.t. consistency since consistency is a necessary requirement for hash chain integrity. That is, we add the following property to Definition 3:

α_5 *(Hash Chain Integrity):* α_5 contains all runs that are not in $\alpha_i, i = 1, \ldots, 4$ and where a peer reports a block that violates hash chain integrity.

Furthermore, the combined property α from Definition 3 is extended to be $\alpha := \bigcup_{i \in \{1, \ldots, 5\}} \alpha_i$. The accountability constraint for α stays the same. These changes give the formal definition of (individual) accountability w.r.t. hash chain integrity.

To show accountability w.r.t. hash chain integrity, the judge has to be modified to also render verdicts whenever a run is in α_5. Note that this is easy to check for the judge. If the property is violated, we blame the peer that reported the violating block. This judge is both fair and complete: Intuitively, all new blocks of an honest peer contain the correct hash of their respective previous block. Since the previous block is the same for all peers (due to consistency), an honest peer will never violate hash integrity and thus never be blamed. In contrast, once a violation is detected, then a single peer is blamed. We therefore obtain the following result:

Proposition 4 (Fabric* achieves accountability w.r.t. hash chain integrity). *Let $\mathbb{P}_F$ be the Fabric* protocol with arbitrary parameters where the judge J is as above. Then it holds true that $\mathbb{P}_F$ achieves individual accountability w.r.t. hash chain integrity.*

Proof. Follows analogously to the proof of Theorem 2 using the intuition given above for completeness and fairness in the case of α_5. □

C.3. Chain soundness of Fabric*

The authors of Fabric want to achieve the goal of *no creation*. This is essentially a form of access control which states that, if a peer accepts a block, then all transactions in that block were sent by a client before. Since orderers in Fabric* must include all messages from the Kafka stream in their blocks, including transactions with potentially invalid signatures, the goal of no creation technically does not hold true for Fabric*. However, we can show that Fabric* provides accountability w.r.t. *chain soundness*, which is a similar goal proposed in this work that is "as good" for the purpose of access control: If a peer considers a transaction from a block as valid, i.e., applies the transactions to his ledger state, then this transaction was send by a client before (or the clients signature key is corrupted).

Chain soundness requires additional proofs to be reported to the judge, namely, lists of transactions txValidity that are considered valid by a peer for the current reported block. Formally, the property is then defined as follows:

Definition 5 (Chain Soundness)**.** *Consider a run ω of the system $\mathcal{E}|\mathcal{A}|\mathbb{P}_F$, for an environment $\mathcal{E}$ and an adversary $\mathcal{A}$, that satisfies basic correctness.*[7]

Let $(\mathcal{B}, \mathsf{txValidity}_{\mathcal{B}})$ be a block and the corresponding list indicating the validity of each transaction in that block that are reported at a point $\lfloor\omega\rfloor$ of the run ω. Let tx be a transaction commitment from client c in $\mathcal{B}$ that is considered valid according to $\mathsf{txValidity}_{\mathcal{B}}$. We say that tx violates chain soundness if, at point $\lfloor\omega\rfloor$, the client c is honest but has not sent the transaction proposal contained in tx before.

We say $\mathbb{Q}$ achieves chain soundness in a run ω if there is no transaction commitment tx in ω that violates chain soundness.

Individual accountability w.r.t. chain soundness is defined by modifying the definition of accountability w.r.t. consistency (cf. Definition 3) in the following way. Firstly, we change α_1 to also check that reported evidence include a well-formed $\mathsf{txValidity}$ that marks only well-formed transactions as valid. Secondly, we change α_2 to also check correct client identities. Thirdly, we add a new run property:

α_6 *(Chain Soundness):* α_6 contains all runs that are not in $\alpha_i, i = 1, \ldots, 3$ and where a peer reports $(\mathcal{B}, \mathsf{txValidity} - \mathcal{B})$ such that $\mathsf{txValidity}_{\mathcal{B}}$ marks a transaction commitment tx of an honest client c as valid such that c has not send the transaction proposal contained in tx before.

Fourthly, the run property α is changed to be $\alpha := \bigcup_{i \in \{1,\ldots,3,6\}} \alpha_i$, i.e., it includes the basic correctness properties and the new property that covers chain soundness. The accountability constraint for α remains unchanged. These changes give the formal definition of (individual) accountability w.r.t. chain soundness.

To show accountability w.r.t. chain soundness, the judge J works as follows: It checks properties $\alpha_1, \ldots, \alpha_3$ analogous to the judge given in Section 6.2. In addition, for α_1 and α_2 it also checks the additional change from above. By similar arguments as given in the proof of Theorem 2, one can show that fairness and completeness still hold. Finally, observe that α_6 cannot directly be checked by the judge since he has no list of past transactions of clients. Instead, the judge checks, for each reported block, whether there is a transaction commitment that is marked as valid but contains a transaction proposal with an invalid signature. If the judge finds such a transaction commitment, then the peer that reported the block is blamed. Fairness for this case follows from the fact that honest peers will mark transactions as invalid if they have an invalid signature. Completeness follows from the fact that, if the client is honest, then his signatures are unforgeable. Thus, transactions that violate chain soundness always contain an invalid signature, which is detected by the judge. Using this judge, we obtain the following result:

Proposition 5 (Fabric* achieves accountability w.r.t. chain soundness)**.** *Let $\mathbb{P}_F$ be the Fabric* protocol with arbitrary*

---

7. Basic correctness is extended to require that reported blocks $\mathcal{B}$ come with a list of valid transactions $\mathsf{txValidity}_{\mathcal{B}}$ of the currently reported block, and all transactions that are well-formed, including a signature field, and an identity of a client $c \in \mathsf{C}$, where C is part of the parameters of $\mathcal{F}_{\mathrm{init}}$.

parameters where the judge J is as above. Then it holds true that $\mathbb{P}_F$ achieves individual accountability w.r.t. chain soundness.

Proof. Follows analogously to the proof of Theorem 2 using the intuition given above for completeness and fairness. ☐

Appendix D.
Transferring Ideal Results to Realizations

All of our security results are shown for an ideal signature functionality $\mathcal{F}_{\mathrm{sig}}$. Here we show that if $\mathcal{F}_{\mathrm{sig}}$ is replaced by its realization $\mathcal{P}_{\mathrm{sig}}$ (cf. [46]) where signing is done using an EUF-CMA secure secure signature scheme $\mathcal{P}_{\mathrm{sig}}$ (*existential unforgeability under chosen message attacks*), then all results still hold true. EUF-CMA is the standard security notion for signature schemes (see, e.g., [26]) and it is well-known that such schemes are (necessary and) sufficient to realize an ideal signature functionality (see, e.g., [38], [46]).

Let $\mathbb{P}_F$ be the Fabric* protocol and let $\mathbb{P}_F^r$ be the system obtained from $\mathbb{P}_F$ by replacing $\mathcal{F}_{\mathrm{sig}}$ by $\mathcal{P}_{\mathrm{sig}}$. Informally speaking, the proposition below states that all our results indeed carry over to $\mathbb{P}_F^r$. We provide a proof sketch for the proposition, giving the general intuition, and then formally show this result for the property "accountability w.r.t. consistency"; all other properties follow analogously. We emphasize that while this result uses that $\mathcal{P}_{\mathrm{sig}}$ realizes $\mathcal{F}_{\mathrm{sig}}$ in the usual UC sense (see, e.g., [46]), the result does not follow from the UC composition theorems.

Proposition 6 (Transferability of Security Properties - informal)**.** *Let $\mathfrak{P}$ be one of the previous security properties that holds true for $\mathbb{P}_F$. If the signature scheme used by $\mathcal{P}_{\mathrm{sig}}$ is EUF-CMA secure, then $\mathfrak{P}$ also holds true for $\mathbb{P}_F^r$.*

Proof Sketch. The proof uses the fact that $\mathcal{P}_{\mathrm{sig}}$ with a EUF-CMA secure signature scheme and $\mathcal{F}_{\mathrm{sig}}$ with an appropriate simulator $\mathcal{S}$ are indistinguishable for any polytime environment (cf. [46]). Suppose the security property $\mathfrak{P}$ is violated with non-negligible probability in $\mathcal{E}|\mathcal{A}|\mathbb{P}_F^r$ (for some environment $\mathcal{E}$ and adversary $\mathcal{A}$). Then, we can build a distinguishing environment $\mathcal{E}'$ for $\mathcal{P}_{\mathrm{sig}}/\mathcal{F}_{\mathrm{sig}}$ by internally simulating $\mathcal{E}|\mathcal{A}|\mathbb{P}_F^r$ except for signing requests, which are handled using the external $\mathcal{P}_{\mathrm{sig}}/\mathcal{F}_{\mathrm{sig}}$. Since all security properties can be observed and checked by $\mathcal{E}'$ in this (almost) full simulation of $\mathcal{E}|\mathcal{A}|\mathbb{P}_F^r$, $\mathcal{E}'$ has a non-negligible advantage of distinguishing between running with $\mathcal{P}_{\mathrm{sig}}$ and running with $\mathcal{S}$ and $\mathcal{F}_{\mathrm{sig}}$, in contradiction to the assumption that $\mathcal{P}_{\mathrm{sig}}$ realizes $\mathcal{F}_{\mathrm{sig}}$ with $\mathcal{S}$. ☐

We now formally show the above proposition for accountability w.r.t. consistency:

Theorem 7. *Let $\mathcal{E}$ and $\mathcal{A}$ be an environment and adversary such that $\mathcal{E}|\mathcal{A}|\mathbb{P}_F^r$ is ppt and $\mathbb{P}_F^r$ be the Fabric protocol using a real signature scheme as defined earlier.*

If the signature scheme used in $\mathcal{P}_{\mathrm{sig}}$ is EUF-CMA secure, then $\mathcal{E}|\mathcal{A}|\mathbb{P}_F^r$ is also accountable w.r.t. consistency.

Proof. Let the signature scheme used by $\mathcal{P}_{\mathrm{sig}}$ be EUF-CMA secure. Then, by the results from [38], [46], we know that there exists a simulator $\mathcal{S}$ such that $\mathcal{P}_{\mathrm{sig}}$ running with a dummy adversary $\mathcal{A}_D$ is indistinguishable

from $\mathcal{F}_{\text{sig}}$ running with the simulator $\mathcal{S}$ for every (ppt) environment $\mathcal{E}$. That is, for all environments $\mathcal{E}$ we have $\mathcal{E} \mid \mathcal{A}_D \mid \mathcal{P}_{\text{sig}} \equiv \mathcal{E} \mid \mathcal{S} \mid \mathcal{F}_{\text{sig}}$.

Assume by contradiction that $\mathbb{P}_{\text{F}}^r$ does not provide accountability w.r.t. consistency. Then, there exist $\mathcal{E}$ and $\mathcal{A}$ such that $\mathcal{E}|\mathcal{A}|\mathbb{P}_{\text{F}}^r$ runs in polynomial time and the judge violates fairness or completeness (of the accountability property for consistency) in a non-negligible set of runs. We define a new environment $\tilde{\mathcal{E}}$ that connects to $\mathcal{A}_D \mid \mathcal{P}_{\text{sig}}$ and internally simulates $\mathcal{E} \mid \mathcal{A} \mid \mathbb{P}_{\text{F}}^r$, except for using the external machine $\mathcal{P}_{\text{sig}}$ for signing requests. The simulation continues at most until the polynomial runtime bound of $\mathcal{E}|\mathcal{A}|\mathbb{P}_{\text{F}}^r$ has been reached. At the end of the simulation, $\tilde{\mathcal{E}}$ checks whether the judge violated fairness or completeness. Note that $\tilde{\mathcal{E}}$ can indeed observe a violation since this decision depends only on information that is available in the internal simulation and on the correctness of signatures, which can be checked by using $\mathcal{P}_{\text{sig}}$. If a violation is detected, then $\mathcal{E}$ outputs 1. Otherwise, the environment outputs 0. Note that $\tilde{\mathcal{E}}$ is a ppt environment as it performs at most a polynomial number of steps, no matter which system it connects to.

Observe that $\mathcal{E}|\mathcal{A}|\mathbb{P}_{\text{F}}^r$ and $\tilde{\mathcal{E}} \mid \mathcal{A}_D \mid \mathcal{P}_{\text{sig}}$ behave identical except for the computation of the overall output of the run. In particular, the runtime bound on the internal simulation of $\tilde{\mathcal{E}}$ is never exceeded, i.e., the simulation never aborts early. Since the probability of violation of fairness/completeness in $\mathcal{E}|\mathcal{A}|\mathbb{P}_{\text{F}}^r$ is non-negligible, we obtain that the system $\tilde{\mathcal{E}} \mid \mathcal{A}_D \mid \mathcal{P}_{\text{sig}}$ outputs 1 with non-negligible probability. Furthermore, because $\tilde{\mathcal{E}}$ is a ppt en-

Note that the system $\mathcal{E}^p \mid (\mathcal{A} \mid \mathcal{S})^p \mid \mathbb{P}_{\text{F}}$ runs in polynomial time (in particular, by definition $\mathbb{P}_{\text{F}}$ runs in polynomial time of its inputs received by the environment and adversary). Thus, by the Theorem 2, the system

vironment and due to the indistinguishability of $\mathcal{A}_D \mid \mathcal{P}_{\text{sig}}$ and $\mathcal{S} \mid \mathcal{F}_{\text{sig}}$, we also have that the system $\tilde{\mathcal{E}} \mid \mathcal{S} \mid \mathcal{F}_{\text{sig}}$ outputs 1 with non-negligible probability.

Observe that the system $\tilde{\mathcal{E}} \mid \mathcal{S} \mid \mathcal{F}_{\text{sig}}$ behaves just as $\mathcal{E} \mid (\mathcal{A} \mid \mathcal{S}) \mid \mathbb{P}_{\text{F}}$ (where $\mathcal{A} \mid \mathcal{S}$ is an attacker that internally runs both $\mathcal{A}$ and $\mathcal{S}$) except for the output, unless the internal runtime bound of the simulation in $\tilde{\mathcal{E}}$ is exceeded. The runtime bound is exceeded with at most negligible probability by Lemma 8 from [47] (otherwise, one could distinguish $\mathcal{A}_D \mid \mathcal{P}_{\text{sig}}$ and $\mathcal{S} \mid \mathcal{F}_{\text{sig}}$). Since $\tilde{\mathcal{E}} \mid \mathcal{S} \mid \mathcal{F}_{\text{sig}}$ outputs 1 with non-negligible probability, this implies that the system $\mathcal{E} \mid (\mathcal{A} \mid \mathcal{S}) \mid \mathbb{P}_{\text{F}}$ violates fairness or completeness with non-negligible probability. We will use (*) to refer to this observation in the following.

The runtime of $\mathcal{E} \mid (\mathcal{A} \mid \mathcal{S}) \mid \mathbb{P}_{\text{F}}$ is polynomial, except for a negligible set of runs. This also follows from Lemma 8 from [47] and the construction of $\mathcal{S} \mid \mathcal{F}_{\text{sig}}$ which runs in polynomial time when receiving overall inputs of at most polynomial length. Let p the polynomial that bounds the runtime of that system. We define $\mathcal{E}^p$ and $(\mathcal{A} \mid \mathcal{S})^p$ to be an environment and an adversary that simulate $\mathcal{E}$ and $\mathcal{A} \mid \mathcal{S}$, respectively, for at most $p(1^\eta)$ steps, where η is the security parameter. By construction, the systems $\mathcal{E} \mid (\mathcal{A} \mid \mathcal{S}) \mid \mathbb{P}_{\text{F}}$ and $\mathcal{E}^p \mid (\mathcal{A} \mid \mathcal{S})^p \mid \mathbb{P}_{\text{F}}$ behave identical except for a negligible set of runs where the runtime bound p is reached. Hence, (*) implies that $\mathcal{E}^p \mid (\mathcal{A} \mid \mathcal{S})^p \mid \mathbb{P}_{\text{F}}$ also violates fairness/completeness with non-negligible probability. We use (**) to refer to this result in what follows. provides accountability w.r.t. consistency, i.e., fairness and completeness are violated in at most a negligible set of runs. This contradicts (**) and hence proves the proposition. $\square$