Accountable Bulletin Boards: Definition and Provably Secure Implementation

Mike Graf* , Ralf Küsters* , Daniel Rausch* , Simon Egger†, Marvin Bechtold‡ , and Marcel Flinspach§
University of Stuttgart
Stuttgart, Germany
Email: *{mike.graf,ralf.kuesters,daniel.rausch}@sec.uni-stuttgart.de, †egger.simon@pm.me,
‡marvin.bechtold@iaas.uni-stuttgart.de, §marcelflinspach.uni@gmail.com

Abstract

Bulletin boards (BB) are important cryptographic building blocks that, at their core, provide a broadcast channel with memory. BBs are widely used within many security protocols, including secure multi-party computation protocols, e-voting systems, and electronic auctions. Even though the security of protocols crucially depends on the underlying BB, as also highlighted by recent works, the literature on constructing secure BBs is sparse. The so-far only provably secure BBs requiretrusted components and sometimes also networks without message loss, which makes them unsuitable for applications with particularly high security needs where these assumptions might not always be met.

In this work, we fill this gap by leveraging the concepts of accountability and universal composability (UC). More specifically, we propose the first ideal functionality for accountable BBs that formalizes the security requirements of such BBs in UC. We then propose $\text{Fabric}^*_{\text{BB}}$ as a slight extension designed on top of Fabric^*, which is a variant of the prominent Hyperledger Fabric distributed ledger protocol, and show that $\text{Fabric}^*_{\text{BB}}$ UC-realizes our ideal BB functionality. This result makes $\text{Fabric}^*_{\text{BB}}$ the first provably accountable BB, an often desired, but so far not formally proven property for BBs, and also the first BB that has been proven to be secure based only on standard cryptographic assumptions and without requiring trusted BB components or network assumptions. Through an implementation and performance evaluation we show that $\text{Fabric}^*_{\text{BB}}$ is practical for many applications of BBs.

I. Introduction

Electronic bulletin boards (BBs) are crucial components that are used as central building blocks by many security-sensitive protocols including, e. g., e-voting protocols [1, 10, 27, 41, 49, 56] electronic auctions [53], or multi-party computation (MPC) [7, 8, 55, 63] protocols. At their core, BBs are essentially *broadcast channels with memory* [36] that allow clients to submit messages/items such that all other clients can then read (a prefix of) the same sequence of all messages/items submitted so far. This security property is called *(delayed) consistent view*, *persistence*, or *consistency* and implies, among others, that a BB stores an *unalterable* or *append-only history*. Depending on the specific context of a higher-level protocol in which a BB is to be used, additional security properties are often required, e. g., *(i) liveness* or *non-discrimination* states that honestly transmitted messages will appear eventually on the BB, *(ii) no data injection*, sometimes called *correctness* or *authorized access*, ensures that only items posted by authorized users appear on the BB, *(iii) receipt consistency* ensures that, if the BB returns a receipt acknowledging that a message has been received, then the message will eventually appear on the BB, *(iv)* some systems require that BB items are *non-clashing* [24] while other systems require that clashing items can appear on the BB [38], and *(v) message validity* requires that all messages on the BB adhere to a specific message format. From a functional point of view, it is desirable for BBs to be *fail-safe* resp. *crash-fault tolerant* [36], i. e., the BB should still stay functional if parts of the BB components fail.

Many works have already proposed constructions of BB protocols including [9, 20, 22, 24, 36, 37, 46]. However, only [24, 39, 46] come with a formal security proof of their protocols. The BB protocols in these works assume trust in some parties running the BB, such as a trusted core component [24, 46], an honest majority [46], or a threshold of honest BB parties [39]. For applications of BBs with particularly high security requirements such trust assumptions can be undesirable and sometimes even unrealistic. For example, in high-stakes applications such as electronic elections or (MPC-based) auctions the parties running a BB might have a vested interest in modifying the result to their benefit. The same is true for the closely related field of distributed ledgers and blockchains, which are often used as a drop-in replacement for BBs in practice: so far, all existing provably secure distributed ledgers (e. g., [5, 25, 30, 35, 47]) require

an honest (super-)majority or an equivalent assumption. In many cases, they also require strong network assumptions, such as networks without message loss which, again, might not be met in practice. Altogether, a BB protocol that is provably secure *without requiring honesty of any BB party, compatible with fully asynchronous real world networks, and only based on standard cryptographic assumptions* is still missing.

In the literature, including works on e-voting and MPC [1, 7, 8, 41, 49, 52, 55], it is very common to construct higher-level protocols/applications and prove their security by simply assuming the existence of a perfect, never failing, incorruptible BB with instantaneous message delivery. Of course, such a perfect BB does not exist in reality and it remains unclear in how far security proofs still apply in practice when protocols are deployed with an actual BB implementation. Indeed, recent works show that this oversimplification leads to severe real-world attacks [21, 39].

Altogether, this raises the following open research questions: *Can we construct a provably secure BB protocol without requiring trust in any of the parties running the BB and without restricting the network? Can we further make this result re-usable such that higher-level protocols can be constructed and shown to be secure based on this BB?* In this work, we answer these questions affirmatively by leveraging accountability and universal composability as our main tools.

Tool 1: Accountability. Previous works that construct BB protocols aim to achieve so-called *preventive* security [28]. That is, it should be impossible to break a security property of the BB, such as consistency, even if parties running the BB actively misbehave. Achieving preventive security properties generally requires introducing (sometimes strong) assumptions which might not always be met in practice. For example, all previous provably secure BBs [24, 39, 46] require trust in at least some parties running the BB.

In this work, we take a different route and propose using the concept of *(individual) accountability* [28, 33, 48, 50] to obtain a secure BB. Individual accountability intuitively states that, if some intended security property of a protocol, e. g., consistency in a BB, is violated, then one can obtain undeniably cryptographic evidence that identifies at least one misbehaving protocol participant that has deviated from the protocol.[1,2] In addition to this *completeness* property, identification based on evidence must also be *fair* in that parties honestly following the correct protocol are never mistakenly blamed. Given such evidence, it is then possible to hold parties accountable for misbehavior, e. g., via financial or contractual penalties. This in turn serves as a strong incentive for malicious parties to honestly follow the protocol such that security properties will not break in the first place. In the case of BBs and the applications discussed in this paper, accountability should be *public*. That is, everyone including all clients and even external observers of the BB should be able to detect and obtain evidence of misbehavior. While accountability is often cited as desirable and sometimes even claimed as a feature of BBs [24, 37, 46], so far there is no BB that has been proven to achieve accountability with respect to at least some of its security properties.

As discussed in detail in [33], preventive security and accountability are *orthogonal* concepts which take different viewpoints on how a protocol can be protected, each with its own advantages and tradeoffs: Preventive security guarantees that security properties cannot be broken at all no matter what malicious parties do but require certain (sometimes very strong) assumptions for this to hold true. Accountability rather accepts that malicious parties can in principle choose to break a security property but uses *detection and deterrence* to discourage them from making use of this option. As a result, accountability-based security can often already be achieved under weaker and possibly more realistic assumptions, which is why we follow this approach in this work. For example, Fabric* [31] provides accountability w.r.t. consistency without assuming eventual message delivery or honest protocol participants. To achieve preventive consistency in a Byzantine-fault tolerant (BFT) algorithm, e. g., PBFT [19], one typically requires an honest supermajority among the protocol participants. On the flip side, accountability-based security might fail to protect a protocol if penalties for detected misbehavior are chosen to be too small to act as a deterrence.

While both preventive and accountability-based security can be used each on their own to protect a protocol, they can also be combined to create a layered defense (cf. [33]). In such a case, a property is shown to be preventively secure as long as certain assumptions hold true, with accountability serving as a backup for cases when one or more of those assumptions are no longer met.

[1] In this work, we always mean "individual accountability" when we say "accountability". Other works sometimes also consider weaker accountability forms where it might not be possible to identify a single misbehaving party.

[2] Ideally, one might want to identify *all* misbehaving parties. This is generally not possible since some types of misbehavior cannot be observed [50].

Tool 2: Universal Composability. The universal composability (UC) paradigm (e. g., [12, 13, 40, 51]) is an approach for designing, modeling, and analyzing security protocols. Compared to game-based analyses, UC provides strong security guarantees and supports the modular design and analysis of protocols. In a UC analysis one first defines an *ideal functionality/protocol* $\mathcal{F}$ that specifies the intended security properties of a target protocol, i. e., $\mathcal{F}$ is secure by definition but typically cannot be run in reality. For a concrete realization, the *real protocol* $\mathcal{P}$, one then proves that $\mathcal{P}$ is at least as secure as $\mathcal{F}$ for a suitable simulator that has full control over the network traffic of $\mathcal{F}$, i. e., no environment $\mathcal{E}$ can distinguish $\mathcal{P}$ from $\mathcal{F}$ where $\mathcal{F}$ runs with the simulator. One can then build higher-level protocols $\mathcal{P}'$ on top of $\mathcal{F}$ and analyze their security. A so-called composition theorem provided by the underlying UC model immediately implies that $\mathcal{P}'$ remains secure even after the ideal subroutine $\mathcal{F}$ is replaced/implemented by the concrete realization $\mathcal{P}$.

Ideal BB Functionalities. As a first contribution of this work and for the first time in the literature, we formalize the notion of an ideal functionality for individually accountable BBs in a universal composability model.

As explained above, while typical applications require that a BB at least provides append-only and consistency, the exact properties expected from a secure BB strongly depend on and vary wildly between different applications that are built on top of the BB. The BB properties needed by one application can even be mutually exclusive with the BB properties required by another application, e. g., [24] requires a BB with non-clashing items while [38] requires a BB that supports clashing items. Altogether, there does not exist the "one-size fits all" BB with a single fixed set of security properties. This is reflected in our work. We first design the ideal BB functionality $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ that captures the typically expected BB security properties, namely (accountability w.r.t.) consistency, which also ensures that data can only be appended. For many applications, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ therefore is already sufficient. We further explain how $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ may be extended to capture all standard BB security properties from the literature. We merge these extensions into the highly customizable BB functionality $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ which can be instantiated to capture arbitrary combinations of additional security properties on top of consistency, including all of the aforementioned ones.

We note that, while the focus of this work lies on accountability, $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ is actually able to capture both preventive and accountable security properties, even in combination. Also, to the best of our knowledge, these BB functionalities are not only the first ones with accountability, they are, more generally, also the first ideal BB functionalities that are not just modeling a perfect setup assumption but that can actually be realized by a concrete implementation (cf. Section VI).

An Individually Accountable BB Fabric$_{\mathrm{BB}}^{*}$. Next, we propose the first provably secure BB that does not require trust in any party running the BB and is compatible with a fully asynchronous network. Our starting point is Hyperledger Fabric, one of the most prominent open-source distributed ledgers. According to Forbes, many important companies in different economic sectors, such as Amazon, IBM, ING, Intel, Microsoft, VISA, Walmart, and NASDAQ are using Fabric [26]. In [31], Graf et al. proposed a minor modification to Fabric, called Fabric*, which improves accountability. They show in a game-based analysis that the core components of Fabric* are accountable w.r.t. consistency.

We design our BB Fabric$_{\mathrm{BB}}^{*}$ by slightly extending and instantiating Fabric*. Among others, this extension lifts accountability guarantees from the core components to the full protocol including clients. We then formally prove that Fabric$_{\mathrm{BB}}^{*}$ realizes an instantiation of $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$, i. e., is a secure BB that achieves accountability w.r.t. consistency. Among others, this extends the previous accountability result of Graf et al. to the full protocol and lifts it from the game-based to the stronger and modular UC setting.

We further translate, adapt, and specialize the concept of smart contracts from DLTs to the setting of BBs; we call the resulting concept *smart read*. Intuitively speaking, a BB with smart read not only provides its state to clients but also allows clients to obtain the (correct) output of functions evaluated on that state. By this, clients can outsource computational tasks to the BB. For example, electronic elections often require running verification procedures on the contents of the BB which can be outsourced to the BB itself via smart reads.

We formalize accountability w.r.t. smart read in our instantiation of $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ and show that Fabric$_{\mathrm{BB}}^{*}$ also achieves this property. As we discuss in Section IV-C, this implies as a simple corollary that Fabric$_{\mathrm{BB}}^{*}$ offers (accountability w.r.t.) several other of the aforementioned BB properties as well since many of these properties directly follow from the smart read property.

This result answers both of our initial research questions. We are able to show that Fabric$_{\mathrm{BB}}^{*}$ realizes $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ based

on standard cryptographic assumptions such as EUF-CMA-secure signatures; no trust in any of the parties running the BB or network assumption are needed thanks to accountability-based security (cf. Section IV-F for an overview of assumptions). Since this is a UC security result, higher-level protocols $\mathcal{P}'$ can be designed and analyzed based on the ideal BB $\mathcal{F}^{\mathrm{acc}}_{\mathsf{c}\mathrm{BB}}$. The UC composition theorem then implies that all security results for $\mathcal{P}'$ are retained even if $\mathcal{P}'$ is later implemented using Fabric$^*_{\mathrm{BB}}$.

Performance Evaluation of Fabric$^*_{\mathrm{BB}}$. As we detail in Section V, the overall performance of Fabric$^*_{\mathrm{BB}}$ is essentially the same as for the underlying Fabric* which, however, has not been implemented and benchmarked so far. As a contribution of independent interest, we therefore provide the so-far missing implementation and evaluation of Fabric* [34]. Our results show that Fabric$^*_{\mathrm{BB}}$/Fabric* can write up to 500 items per second to the BB/ledger which is sufficient for many BB applications. For example, in the context of e-voting 200,000 ballots can be added to the BB in less than 10 minutes.

Summary of Our Contributions.

- We provide the first ideal accountable BB functionalities, namely $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$ and $\mathcal{F}^{\mathrm{acc}}_{\mathsf{c}\mathrm{BB}}$. They are also the first ideal (including non-accountable) BB functionalities that are not just perfect setup assumptions but can be realized.
- We propose the novel property of *smart read* for BBs.
- We propose Fabric$^*_{\mathrm{BB}}$ by instantiating and slightly extending Fabric* and prove that Fabric$^*_{\mathrm{BB}}$ UC-realizes an instantiation of $\mathcal{F}^{\mathrm{acc}}_{\mathsf{c}\mathrm{BB}}$, i. e., provides accountability w.r.t. consistency and smart read. This is the first provably secure BB that does not require trust in any BB party or assumptions on the network. This is also the first BB that is provably UC secure and hence the first BB security result that can directly be re-used by higher-level protocols.
- We implement and benchmark the underlying Fabric*. Our results demonstrate that Fabric$^*_{\mathrm{BB}}$ is practical.

Structure of this paper. After recalling preliminaries in Section II, we propose $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$ and $\mathcal{F}^{\mathrm{acc}}_{\mathsf{c}\mathrm{BB}}$ in Section III. Section IV recalls the Fabric* protocol, proposes Fabric$^*_{\mathrm{BB}}$ based on Fabric*, and shows that Fabric$^*_{\mathrm{BB}}$ realizes an instantiation of $\mathcal{F}^{\mathrm{acc}}_{\mathsf{c}\mathrm{BB}}$. Section V presents our implementation and benchmarks of Fabric* (available for download at [34]) and discusses the implications for Fabric$^*_{\mathrm{BB}}$. We discuss related work in Section VI. Full details and proofs are given in the appendix.

II. Preliminaries

Here we briefly recall relevant details of our main tools.

A. Computational Model

There are many different models following the universal composability paradigm, e. g., [12, 13, 40, 51]. Formally, here we use the iUC model, a highly general model by Camenisch et al. [12]. However, all of our definitions and results can also be translated to other models for universal composability such as the aforementioned ones. We will keep the presentation on a level such that readers familiar with any of these UC models can understand the paper.

In all UC models, the security experiment compares a *real protocol* $\mathcal{P}$ with the so-called *ideal protocol* or *functionality* $\mathcal{F}$ which is typically an ideal specification of some task. The idea is that if one cannot distinguish $\mathcal{P}$ from $\mathcal{F}$, then $\mathcal{P}$ must be "as good as" $\mathcal{F}$, written $\mathcal{P}\leq\mathcal{F}$. More specifically, we have $\mathcal{P}\leq\mathcal{F}$ if there exists a *simulator/ideal adversary* $\mathcal{S}$ that controls the network of $\mathcal{F}$ such that $\mathcal{P}$ and $\mathcal{F}$ (alongside $\mathcal{S}$) are indistinguishable, i.e., no environment $\mathcal{E}$ can tell whether it interacts with $\mathcal{P}$ (on both $\mathcal{P}$'s I/O and network interface) or with $\mathcal{F}$ along with $\mathcal{S}$ on $\mathcal{F}$'s I/O interface and the network interface of $\mathcal{S}$ it exposes to $\mathcal{E}$; equivalently one can consider a *real adversary* that runs alongside $\mathcal{P}$, analogously to $\mathcal{F}$ running with $\mathcal{S}$.

A protocol in a UC model is typically modeled as a set of interacting Turing machines. An instance of a machine manages or represents one or more so-called *entities*. An entity is identified by a tuple $(pid, sid, role)$. It describes a specific party with party ID (PID) pid running in a session with session ID (SID) sid and executes some code defined by the role $role$. Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances. In what follows, we use the terms entity and party interchangeably.

We call a party in a protocol *main* if it can directly receive inputs from and send outputs to the distinguisher/environment (who subsumes arbitrary higher-level protocols). We call a party *internal* otherwise, i. e., if it is part of an

internal subroutine. Whether a party is internal or main can be determined from its role. As in all UC models, an ideal functionality and a realization share the same sets of main parties/roles. A realization might have additional internal parties/roles that are not present in the ideal protocol and vice versa.

The adversary is allowed to corrupt a party by sending a special corrupt command on the party's network interface. If an entity is corrupted, the adversary generally gets full control over the entity. The environment can obtain the current corruption status of main parties in a protocol, which allows for checking whether corruption of main parties is simulated correctly.

We provide an extended overview of iUC and a formal definition of our pseudo code notation in Appendix A, resp. in Appendix B.

B. Accountability in UC

Graf et al. [32] recently proposed the AUC framework, which provides a general blueprint for modeling and formally proving accountability of arbitrary security properties of protocols within any model for universal composability. Therefore, AUC provides a template for how accountability is incorporated into ideal and real protocols. To formally prove that a real protocol provides accountability w.r.t. a security property, one then shows – as common in UC – that the real protocol is indistinguishable from the ideal protocol. Here we briefly recall those aspects of AUC that we use in this work. For interested readers, additional details are available in Appendix C and [32].

Formalizing Accountability in Ideal Functionalities. Start with an ideal functionality $\mathcal{F}$ that formalizes (preventive security of) a certain set of security properties $\mathcal{S}ec$, such as consistency, non-clashing, or liveness. Intuitively, AUC modifies $\mathcal{F}$ using the following main ideas to capture accountability of a subset $\mathsf{Sec}^{\mathsf{acc}} \subseteq \mathcal{S}ec$ of those properties. At any point in time, the adversary/simulator on the network is allowed to send a special message requesting that the ideal functionality $\mathcal{F}$ should from now on consider a property $p \in \mathsf{Sec}^{\mathsf{acc}}$ to be broken. This request must contain a so-called *verdict*, which identifies at least one unique party that has been misbehaving and hence has led to this breach of security. Verdicts in AUC are positive boolean formulas consisting of terms of the from $\mathsf{dis}((pid, sid, role))$ where $(pid, sid, role)$ is a protocol participant.

The ideal functionality $\mathcal{F}$ verifies that the verdict is fair, i. e., does not blame any parties that are currently uncorrupted and hence honestly following the protocol. If this check succeeds, then the request is accepted and p is marked as broken property, stored in a new state variable brokenProps in $\mathcal{F}$, and the adversary gains additional power depending on the property p. AUC further introduces a new role called (ideal) *public judge*[3] in $\mathcal{F}$ which represents the party that is responsible for computing the verdict. This ideal judge in $\mathcal{F}$ allows higher-level protocols/the environment to obtain the current verdict as previously provided by the attacker. Since a public judge computes verdicts based on *public data*, such a judge can be executed by anyone in reality including arbitrary honest parties. Thus, it makes sense to model (public) judges in AUC as incorruptible.

AUC provides ready-to-use pieces of code that can be included in an ideal functionality $\mathcal{F}$ to establish the above infrastructure, including the judge role, and which we also use here. The protocol designer then still has to define the set $\mathsf{Sec}^{\mathsf{acc}}$ and manually define new logic for $\mathcal{F}$ which specifies the exact implications of breaking a security property p. For example, if p represents consistency in a BB, then breaking p should allow the attacker to send contradicting outputs to clients of the BB.

Formalizing Accountability in Real Protocols. A real protocol $\mathcal{P}$ that is supposed to realize an ideal accountable functionality $\mathcal{F}$ additionally includes a dedicated *real* public judge J that implements the ideal public judge from $\mathcal{F}$. The real judge J describes the exact inputs/evidence and the algorithm/logic that is used to compute verdicts from that evidence in the actual protocol. Hence, the specification of J is one of the key tasks in designing and proving the security of $\mathcal{P}$.

One then shows that $\mathcal{P}$ UC-realizes $\mathcal{F}$. This implies that $\mathcal{P}$ provides accountability w.r.t. all $p \in \mathsf{Sec}^{\mathsf{acc}}$: As long as the real judge J has not output a verdict, by indistinguishability to $\mathcal{F}$ property p has not been broken. Conversely, if the adversary manages to use the logic of $\mathcal{P}$ and its control over corrupted parties in $\mathcal{P}$ to break p, then J must have already computed a verdict and this verdict has to be fair, again, by indistinguishability to the ideal judge in $\mathcal{F}$.

[3]AUC also supports other types of judges. In this work, we focus on public accountability and hence only use public judges, sometimes just called judge in what follows.

III. An Ideal Individually Accountable BB

In this section, we propose the ideal BB functionality $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ and its customizable variant $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$.

A. *The Ideal Accountable BB Functionality* $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$

As explained in the introduction, the most essential property that virtually every BB should provide is *consistency*. That is, clients reading from the BB should obtain (prefixes of) the same global state as stored in the BB. Commonly, consistency also includes the expectation that this global state should be *append-only*, i. e., information that was already read by some client will not change in the future. Here we construct an ideal $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ functionality that formalizes accountability w.r.t. consistency and hence also append-only. Thus, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ provides already sufficient security for constructing many higher-level protocols which only require a broadcast channel with memory, including most of the existing e-voting protocols.

Existing Ideal BB Functionalities in the Literature. As a starting point, we revisit previously existing ideal BB functionalities from UC literature (e.g., [4, 7, 14, 16–18, 61, 62]). These functionalities have only been used to model setup assumptions within higher-level protocols and, at their core, all work similarly to $\mathcal{F}_{\mathrm{BB}}$ reproduced in Figure 1. That is, clients can write a message to $\mathcal{F}_{\mathrm{BB}}$ which is then internally appended to a global list of messages $\mathsf{msglist}$. Clients can also read from $\mathcal{F}_{\mathrm{BB}}$ to obtain the current $\mathsf{msglist}$. Thus $\mathcal{F}_{\mathrm{BB}}$ formalizes (preventive security of) consistency and an append-only state.

Description of $\mathcal{F}_{\mathrm{BB}}$:

Implemented role(s): {`client`}
Corruption model: *incorruptible*
CheckID($pid, sid, role$)**:**
Accept all messages with the same sid. — {*Each instance of this functionality is responsible for an entire session sid*
Main:

recv (`Write`, msg) **from** `I/O`: — {*Write request from a honest identity*
id ← id + 1; msglist.add(id, msg) — {*Record message at position* id

recv `Read` **from** `I/O`: — {*Read request from an honest identity*
reply (`Read`, msglist) — {*Provide full item list to requestor*

Fig. 1: Sketch of a common ideal BB setup assumption $\mathcal{F}_{\mathrm{BB}}$.

All ideal functionalities in the spirit of $\mathcal{F}_{\mathrm{BB}}$ [4, 7, 14, 16–18, 61, 62] model a perfectly secure BB that does not exist in reality. For example, $\mathcal{F}_{\mathrm{BB}}$ implies network connections between parties running the BB and BB client without latency. Hence, even under very strong assumptions such as trusted third parties running the BB, we have for any realistic implementation $\mathcal{P}_{\mathrm{BB}}$ of a BB that $\mathcal{P}_{\mathrm{BB}}$ does not realize $\mathcal{F}_{\mathrm{BB}}$. As a result, $\mathcal{F}_{\mathrm{BB}}$ cannot be used for the security analysis of realistic BBs. Even more, security results obtained for higher-level protocols based on $\mathcal{F}_{\mathrm{BB}}$, including all of the aforementioned works [1, 7, 8, 41, 49, 52, 55], might not hold true when those higher-level protocols are based on an actual implementation $\mathcal{P}_{\mathrm{BB}}$.

In what follows, we therefore lift $\mathcal{F}_{\mathrm{BB}}$ from a setup assumption to $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ which can be realized by realistic BB protocols. We also incorporate accountability-based security into $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ which, as previously mentioned, allows us to prove security statements based on mild security assumptions compared to preventive security and is a desirable property by itself.

Constructing the first realizable and accountable ideal BB. By the previous observations, constructing $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ not only requires adding accountability to $\mathcal{F}_{\mathrm{BB}}$, we also have to add infrastructure to support (possibly fully asynchronous) real-world networks as well as corrupted parties which are necessary for being able to UC-realize such an ideal BB functionality with an actual real-world BB protocol $\mathcal{P}_{\mathrm{BB}}$, irrespective of accountability. We present the core logic of our proposed $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ in Figure 2, with lines capturing accountability following the AUC approach being highlighted in blue.[4] In what follows, we motivate and explain the definition of $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$. Full details are available in Appendix III-A.

[4]By removing those blue lines, one can easily obtain an alternative version that is still realizable but captures preventive security instead of accountability. While not the focus of this work, this might be of independent interest.

Description of $\mathcal{F}_{\text{BB}}^{\text{acc}}$:

```
Main:
  recv (Write, msg) from I/O:                                  {Write request from a honest identity
    writeCtr ← writeCtr + 1
    writeQueue.add(writeCtr, msg)                                {Store msg for update
    send (Write, writeCtr, msg) to NET                           {Leak full msg.
  recv Read from I/O:                                          {Read request from an honest identity
    readCtr ← readCtr + 1
    readQueue.add((pid_call, sid_call, role_call), readCtr, false, 0)
                {Store for later processing, where (pid_call, sid_call, role_call) denotes the calling entity
    send (Read, readCtr) to NET                                  {Leak full details
  recv (DeliverRead, readCtr, (sugPtr, sugOutput)) from NET
      s.t. ((pid, sid, role), readCtr, false, 0) ∈ readQueue:    {A triggers message delivery
    if brokenProps[consistency] = false:      {F_BB^acc provides consistency in the absence of a verdict
      if ∃((pid, sid, role), _, true, prt) ∈ readQueue :
          s.t. prt > sugPtr
        send nack to NET                                         {Delivery request of A was denied
      else:
        readQueue.remove((pid, sid, role), readCtr, false, 0)    {Clean up
        readQueue.add((pid, sid, role), readCtr, true, sugPtr)
        send (Read, msglist(sugPtr)) to (pid, sid, role)
    else:                                        {A defines the output if consistency is broken
      readQueue.remove((pid, sid, role), readCtr, msg, false, 0)
      readQueue.add((pid, sid, role), readCtr, msg, true, 0)
      send (Read, sugOutput) to (pid, sid, role)
  recv (Update, appendToMsglist, updRequestQueue) from NET
      s.t. updRequestQueue ⊂ writeQueue∧
          all entries from updRequestQueue appear in appendToMsglist:   {Update or maintain request trig-
    msglist.append(appendToMsglist)^a                                     gered by the adversary.
    for all item ∈ updRequestQueue do:           {Remove "consumed" elements from writeQueue
      writeQueue.remove(item)
    reply (Update, msglist)                                      {Leak data
  Include static code provided by AUC(cf. Appendix C and [33]) here. This code adds a public judge judge. A can send
  a verdict v to judge indicating that consistency should be broken. judge checks whether v is fair. If so, judge marks
  consistency as broken and sets brokenProps[consistency] to true; otherwise it remains false. If the environment
  E asks judge regarding verdicts, judge provides current verdicts (if any) to E.

a append adds each entry from appendToMsglist to msglist including consecutive IDs
```

Fig. 2: Excerpt of the accountable BB functionality $\mathcal{F}_{\text{BB}}^{\text{acc}}$.

The ideal functionality $\mathcal{F}_{\text{BB}}^{\text{acc}}$ captures accountability w.r.t. consistency, i. e., we consider $\text{Sec}^{\text{acc}} = \{\texttt{consistency}\}$ and no other (preventive) security properties. Clients in $\mathcal{F}_{\text{BB}}^{\text{acc}}$ can issue Write and Read requests. However, unlike $\mathcal{F}_{\text{BB}}$, these requests are not executed instantly. If a client calls Write with some input msg, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ stores the request with an index/ID writeCtr in writeQueue for later processing. Similarly, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ stores Read requests in readQueue, including an index/ID, a flag that indicates that the request has not been processed yet, and a pointer indicating what the entity has read so far. For both read and write, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ leaks the full requests to the network adversary $\mathcal{A}$, resp. the simulator. The adversary is then responsible for deciding whether and when requests are processed, which captures a real-world asynchronous network with latency and potential message loss.

To finish processing a read request and to generate an output, the adversary $\mathcal{A}$ can issue a DeliverRead command

to $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$. For this purpose, $\mathcal{A}$ specifies the unique ID of the pending read request as well as the output that shall be provided. The exact behavior depends on whether the property of consistency can still be guaranteed. By default and as long as the public judge did not detect any misbehavior yet (see below), we have that consistency must still hold true (i. e., brokenProps[consistency] = false) and hence only the black code is executed. In this case, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ enforces consistency for all clients. Therefore, $\mathcal{A}$ provides $sugPtr$ to $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ which determines the exact prefix of the msglist to be output. $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ accepts $sugPtr$ only if it includes at least all messages that the same client has previously already read, if any.[5] Therefore, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ checks that all stored output pointers for the requesting party in requestQueue are smaller or equal to $sugPtr$. Afterwards, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ then stores that the request was processed and that $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ delivered the msglist prefix up to $sugPtr$ to the requestor.

If consistency is broken, i. e., brokenProps[consistency] is set to true, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ does not ensure consistency any longer and it executes the blue code in DeliverRead. In this case, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ returns the string $sugOutput$ as provided by $\mathcal{A}$ to the requestor. This allows $\mathcal{A}$ to freely choose $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$'s output including the option to send contradicting outputs to different clients. Again, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ stores that the request was processed but does not store which data was delivered.

To add pending write requests from writeQueue to the $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$'s state, $\mathcal{A}$ can at any point in time issue a Update command to $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$. This new Update command is an abstraction of a concrete consensus mechanism that is used in a real-world BB to process and sort incoming messages. In an Update, $\mathcal{A}$ specifies *(i)* an ordered list of messages $appendToMsglist$ that shall be appended to the global state msglist and *(ii)* the set of pending write requests that are processed by this update and hence will be removed from writeQueue. As long as these inputs are well-formed, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ will perform such an Update request, i. e., change both msglist and writeQueue accordingly. We note that $\mathcal{A}$ may not only append messages from honest clients that are contained in writeQueue. He can also add arbitrary other messages which captures malicious parties that might add items to the bulletin board without first being triggered via a write command issued by a higher-level protocol.

If the adversary $\mathcal{A}$ provides a fair verdict v to the public judge that identifies at least one unique corrupted party, then $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ marks consistency as broken, i. e., it sets brokenProps[consistency] = true (this operation is part of the static code that we include from AUC as indicated at the bottom of Figure 2). Such a verdict indicates that the public judge has detected that a malicious party deviated from the protocol and hence consistency can no longer be guaranteed.

Discussion. Observe that $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ indeed formalizes accountability w.r.t. consistency: As long as the public judge has not received a fair verdict from $\mathcal{A}$, who by this also indicates that consistency should be considered broken, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ enforces consistency by requiring that all outputs are (non-decreasing) prefixes of the same globally unique msglist. Conversely, as soon as the judge's verdict is non-empty, $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ does not guarantee consistency any longer. However, the party from the verdict can then be held accountable for this security breach.

Hence, if we can show that an actual implementation of a BB $\mathcal{P}_{\mathrm{BB}}$ realizes $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$, then $\mathcal{P}_{\mathrm{BB}}$ must enjoy the same security properties. Since $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ and $\mathcal{P}_{\mathrm{BB}}$ have to have the same behavior towards the environment $\mathcal{E}$, $\mathcal{P}_{\mathrm{BB}}$ has also to provide consistency as long as the judge (in the real protocol) does not render a verdict. As soon $\mathcal{P}_{\mathrm{BB}}$'s judge renders a verdict, $\mathcal{P}_{\mathrm{BB}}$ does not guarantee consistency any longer.

One can also build higher-level protocols $\mathcal{P}'$ on top of $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$, which then use that $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ provided consistency guarantees as long as the public judge does not output a verdict (and if there is a verdict, then $\mathcal{P}'$ can hold the same person accountable also for any failure in the higher-level protocol). By UC composition, all security results shown for $\mathcal{Q}$ carry over when $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ is later implemented via $\mathcal{P}_{\mathrm{BB}}$.

B. Capturing Additional Properties by Customizing $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$

As explained in the introduction, some applications require BBs that achieve additional properties beyond just (accountability w.r.t.) consistency. Our ideal functionality $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ can be customized to also capture arbitrary combinations of other properties, both in an accountable but also in a preventively secure fashion. Such customization mainly entails introducing additional checks while processing Write, Update, and/or DeliverRead commands to enforce further security properties. In what follows, we describe how $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ is modified to obtain the customization

[5] We require only a prefix, instead of the full msglist, to allow for realizations $\mathcal{P}_{\mathrm{BB}}$ without any network assumptions, i. e., that can be deployed in a fully asynchronous real-world network where messages might be lost or delayed.

framework $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ which can be instantiated in various ways to capture arbitrary combinations of properties. The main task of protocol designers is then to define suitable instantiations for the desired properties. We describe how all security properties mentioned in the introduction can be captured via such instantiations of $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$. In Section IV-C, we further establish a full instantiation of $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ that formalizes the precise security properties provided by Fabric*.

We provide the full formal definition of the customization framework $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ inAppendix D. In summary, it involves the following changes compared to $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$:

Customizations of operations: The customization framework $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ is derived from $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ mainly by introducing several additional subroutines, namely $\mathcal{F}_{\mathrm{write}}$, $\mathcal{F}_{\mathrm{update}}$, and $\mathcal{F}_{\mathrm{read}}$, which are called during the `Write`, `Update`, and `DeliverRead` operations.[6] Protocol designers have to specify these subroutines to capture the exact conditions imposed on each operation and thereby formalize modifications to $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$. When called by $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$, a subroutine receives the full internal state allowing it make decisions given the full view. The subroutines $\mathcal{F}_{\mathrm{write}}$, $\mathcal{F}_{\mathrm{update}}$, and $\mathcal{F}_{\mathrm{read}}$ can *(i)* impose additional requirements on, *(ii)* abort, or *(iii)* influence the output/result of `Write`, `Update`, and `DeliverRead` operations. Below we give examples of how this can be used to capture a wide range of security properties.

Supporting multiple types of read requests: The `Read` command is extended to also take an auxiliary input msg, which is an arbitrary bit string. This auxiliary input can be used to distinguish multiple types of reads with possibly differing security properties. We use this feature to formalize the novel security property of *smart read* in Section IV-C.

Addition of an internal clock: To be able to formalize time-based properties such as liveness, the $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ framework additionally contains an internal clock. Formally, this is just an internal counter round which models arbitrary discrete time steps such as seconds or network rounds. The environment/higher-level protocols can query $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ to obtain the current time, i.e., the value of round. The adversary can send a new `UpdateRound` command to request increasing the current value of round by one time unit. The ideal $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ uses a new subroutine $\mathcal{F}_{\mathrm{updRnd}}$ to decide whether this request is granted, where $\mathcal{F}_{\mathrm{updRnd}}$ just as for the other subroutines is a parameter that formalizes the precise conditions and hence security properties that a protocol designer wants to consider.

As a mostly straightforward sanity check, in Appendix **D.4** we formally verify that the $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ customization framework captures $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ as a special case:

Lemma 1 (informal)**.** *There exists an instantiation of (the subroutines of) $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ such that $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ UC-realizes $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ and $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ UC-realizes $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$.*

Capturing standard security properties via $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$. Standard security properties of BBs, including the ones mentioned in the introduction, can be captured via instantiations of $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$. Here we illustrate several examples, starting with preventive security. The remaining properties are discussed in Appendix **D.3**.

Liveness states that write requests will become part of the state of the BB within a bounded time frame, say δ. Furthermore, once stored in the BB, the message will be part of outputs read by clients after another bounded time frame, typically also δ. The first aspect can be formalized by instantiating $\mathcal{F}_{\mathrm{updRnd}}$ to prevent the adversary from advancing time as long as writeQueue still contains pending requests that have been submitted δ time units ago. The second aspect can be formalized by instantiating $\mathcal{F}_{\mathrm{read}}$ to only allow outputs that contain at least all messages that have been added to the global state more than δ time units ago.

Authorized (Write) Access states that only a certain set of clients is allowed to write messages on the BB. This can be formalized by instantiating $\mathcal{F}_{\mathrm{write}}$ and $\mathcal{F}_{\mathrm{update}}$ to drop write requests and state updates, respectively, containing messages from clients that are not part of the authorized set.

Allowing or preventing clashing items in the global state of the BB can be captured by instantiating $\mathcal{F}_{\mathrm{update}}$ to allow or prevent such updates.

Starting with the above preventive formalizations, it is easy to switch to accountability, if desired, using the same method as for accountability w.r.t. consistency in $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$. For example, to consider accountability w.r.t. liveness one starts with the above instantiation of preventive security and first adds the property `liveness` to the set $\mathsf{Sec}^{\mathsf{acc}}$. By the static code of AUC, this has the effect that the adversary can now set brokenProps[`liveness`] $=$ `true` if and

[6]Only $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ can call the subroutines. To allow the subroutines to make decisions based on $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$'s state, $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ always includes its internal state and transcript to calls to the defined subroutines.

only if he provides a verdict v identifying a misbehaving party to the public judge. The protocol designer then only has to modify $\mathcal{F}_{\text{updRnd}}$ and $\mathcal{F}_{\text{read}}$ to first check whether $\mathsf{brokenProps}[\mathtt{liveness}] = \mathtt{true}$ and, if so, skip all security checks that enforce liveness. We finally emphasize that using the above techniques one can easily formalize preventive security for some properties while others are protected by accountability in $\mathcal{F}^{\text{acc}}_{\text{cBB}}$.[7]

IV. Fabric*

We show that the Fabric* distributed ledger proposed by Graf et al. in [31] can be slightly extended and instantiated to obtain a provably secure, composable, and accountable BB which we call $\text{Fabric}^*_{\text{BB}}$. Notably, we are able to prove this result based on standard cryptographic assumptions and without requiring trust in any of the parties running $\text{Fabric}^*_{\text{BB}}$ or network assumptions.

We structure this section as follows: In Section IV-A, we recall the Fabric* protocol. In Section IV-B, we recall previous results regarding Fabric* and relate this to our work. We then, in Section IV-C, formalize, via an instantiation $\mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}$ of $\mathcal{F}^{\text{acc}}_{\text{cBB}}$, the security guarantees that we want our BB to achieve. In Section IV-D we present our BB protocol $\text{Fabric}^*_{\text{BB}}$ based on Fabric*. Finally, in Section IV-E we formally prove the security of $\text{Fabric}^*_{\text{BB}}$, with full details given in Appendix E and F.

A. Recap: The Fabric* Protocol

The Fabric* protocol is derived from the prominent Hyperledger Fabric ledger [2]. Graf et al. [31] construct Fabric* by slightly adapting Fabric to improve accountability of the system. In what follows, we recall Fabric* closely following the terminology and presentation of Graf et al. [31]. Along the way, we recall how Fabric* differs from the original Hyperledger Fabric.

Fabric* is a *permissioned* blockchain protocol typically executed by a set of organizations that do not fully trust each other. These organizations set up a new Fabric* instance – a so-called *channel* – by agreeing externally on a Genesis block. The Genesis block defines the channel's configuration including *(i)* the channel's organizations/participants, *(ii)* the components/parties that run the Fabric* protocol, and *(iii)* the set of (deterministic) smart contracts available in this channel.

Roles in Fabric*. In Fabric*, all protocol participants are identified via certificates which include their role, the organization they belong to, and their public key.

Clients initiate transactions to read from or to write to the blockchain. They typically obtain inputs for transactions from end users. All client interactions in Fabric* are calls to smart contracts that are executed by so-called peers. The smart contracts compute, among others, the outputs that clients obtain and whether any new data is written to the ledger state (see below). Clients do not keep a copy of the chain.

(Endorsing) Peers are essentially the "miners" and "full-nodes" of Fabric*. They execute transactions from clients. They also replicate the chain, i. e., they keep a copy of the full blockchain and allow clients to query data from the chain. Peers also convert the blockchain to a current *ledger state*. They are then responsible for executing smart contracts based on the current ledger state. Peers differ from traditional miners in that they are not directly involved in the block generation process. This process is outsourced to a so-called ordering service.

An Ordering Service is an abstract concept that provides a "consensus service". Transactions from clients are first executed by peers. Afterwards, transactions and execution results are forwarded to the ordering service who forms blocks from the incoming transactions. The service then distributes the blocks to all peers and peers mark transactions in the chain as invalid if necessary.

Transaction Flow. In what follows, we explain how Fabric* runs by following the steps used by clients for reading from or submitting a new transaction to the ledger (cf. Figure 3):

[7] Using techniques established by AUC, it is even possible to formalize security statements in $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ of the form "as long as certain assumptions hold true, then a property is guaranteed to hold true, i.e., preventively secure. If assumptions are broken at some point, then the property is still accountable, i.e., still holds true as long as the judge has not yet obtained a verdict." Since this is not the main focus of this work, we refer to [33] for more details.

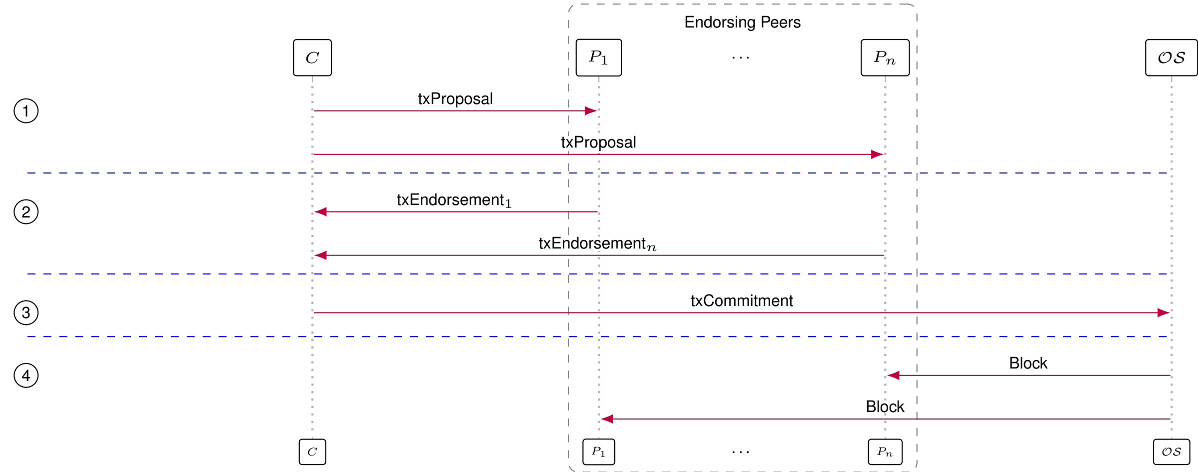


Fig. 3: Example Flow in Fabric* with Client C, (endorsing) peers $P_1, \ldots, P_n$, and ordering service $\mathcal{OS}$ (cf. [31]).

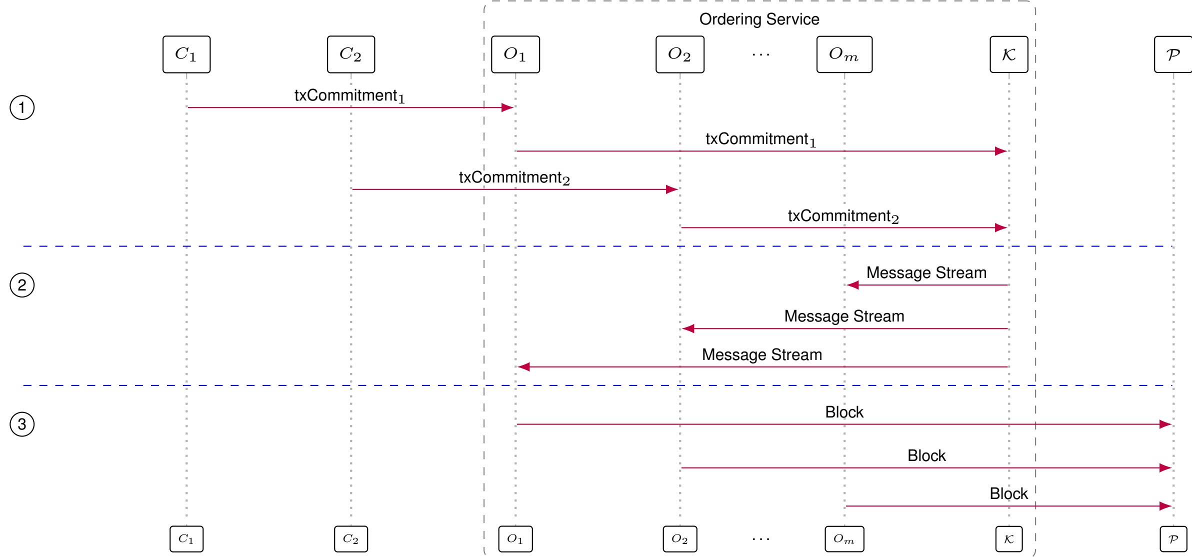


Fig. 4: Ordering and block generation with clients C_1, C_2, orderers $O_1, \ldots, O_m$, a Apache Kafka $\mathcal{K}$, and (endorsing) peers $\mathcal{P}$ (cf. [31]).

Proposal: To interact with a channel, i. e., to read from or to write to it, *clients* call smart contracts at peers by sending a signed *(transaction) proposal* to the (endorsing) peers (cf. ①). After having distributed the proposal, clients wait for the results of their request, the so-called *endorsements*.

Endorsement: Peers execute proposals by running the smart contract with the input parameters specified in the proposal. Smart contracts in Fabric* can be implemented in several common programming languages, such as Go, Java, and node.js. Peers execute this code natively but isolated in Docker containers [43]. At the end of a successful smart contract execution, peers generate an *endorsement*. An endorsement contains the original transaction proposal, the data read from the peer's ledger state during the execution (called *readset*), possibly some changes in the ledger state caused by the execution of the proposal (called *writeset*), and/or potential *output* for the client. This endorsement is a confirmation of the peer that the transaction and its results are allowed to become part of the blockchain. Peers send the signed endorsement to the client initiating the proposal (cf. ②) but do not apply the writesets to their state yet. If the proposal is a read request, the client extracts the output from the endorsements and stops the protocol at this

point. Read requests are thus "off-chain" in that they are not added and confirmed to the blockchain but also do not change the ledger state. Please note that, since the ledger state of different peers may differ, readsets, writesets, and output computed by different peers for the same proposal may differ as well.

Commitment: For a write request that the client wants to add to the ledger state, she keeps collecting endorsements for her proposal until she collects sufficiently many endorsements, e. g., from all peers she queried. The exact number of endorsements that are required is specified as part of the channel setup. Then, the client forwards the proposal and all endorsements as a so-called *commitment* to the ordering service (cf. ③).

Block Generation and Distribution: The ordering service is responsible for block generation (cf. ④). After generating the blocks, the ordering service distributes them to peers. We explain the specific ordering service used by Fabric* below.

Block Validation and Ledger State Update Upon receiving a new block, peers validate whether they accept the block including checking that blocks are correctly signed by the ordering service. After accepting a new block, peers update their current view on the ledger state by iterating over the commitments contained in the block. A commitment is invalid and hence ignored if *(i)* endorsements in the commitment do not agree on the readset and writeset or *(ii)* the readset does not match the current state of the ledger. If a commitment is valid, then the peer applies the writeset to update its current view of the ledger state before checking the next commitment.

Fabric*'s Ordering Service. The original Hyperledger Fabric supports various ordering services, including an Apache Kafka-based ordering service [3, 42]. Fabric* uses this ordering service but with one major modification.

As depicted in Figure 4, the Kafka-based ordering service consists of two components: *(i)* so-called *orderers* provide the interface for clients and peers to the ordering service. They collect commitments from clients that should enter the blockchain, forward them to Apache Kafka [3] (in short Kafka), and receive a totally ordered sequence of messages back. Thereupon, orderers follow a deterministic algorithm to split this message sequence into blocks, sign them, and then distribute those blocks to peers. *(ii)* Kafka is a crash fault-tolerant distributed consensus service optimized for durability, high throughput, and reliable message distribution. A Kafka cluster consists of several machines called *Kafka brokers*. One of the brokers is the so-called *Kafka leader*. The leader receives all incoming messages from orderers, establishes a total order, and returns the resulting message sequence to the orderers. All non-leading brokers replicate the state of the leader to provide redundancy. If the leader crashes, then the remaining brokers elect a new leader who takes over the duties of the former leader. In practice, the whole Kafka cluster, including all broker machines, is supposed to be run as a service in a data center operated by a single entity [54, 60]. This entity is then responsible for providing a correct execution of the entire Kafka protocol.

By default, the Kafka component as used by Hyperledger Fabric does not provide any accountability. Intuitively, this is improved in Fabric* by letting Kafka leaders additionally sign their ordered sequences of messages and letting orderers include this information in the generated blocks.

B. Results for Fabric*

Graf et al. [31] prove in a game-based security analysis that Fabric* provides public and individual accountability w.r.t. consistency for peers. For our purposes, however, this result is not sufficient: *(i)* Security was shown for Fabric* as a ledger, not a BB. Notably, providing and proving security guarantees for clients was out of scope. *(ii)* Since the security analysis is game-based, it is not directly composable with higher-level protocols that might want to use Fabric*. *(iii)* While smart contracts were formally modeled, no security results were shown for (the execution of) those smart contracts.

We build on these results for Fabric* to design and prove the security of Fabric$^*_{\mathrm{BB}}$, thus showing that this and similar distributed ledgers can be used to build accountable BBs (cf. Section IV-F). Along the way, we also extend prior results for Fabric* in various ways: *(i)* As part of Fabric$^*_{\mathrm{BB}}$, we propose a slight extension of the clients of Fabric* which is needed to lift consistency from peers to clients. *(ii)* The security proof of Fabric$^*_{\mathrm{BB}}$ extends the previous consistency result for Fabric* from peers to clients (using the aforementioned extension of clients). We also show, for the first time, that smart contract execution of Fabric* is accountable, which is in turn needed for Fabric$^*_{\mathrm{BB}}$. *(iii)* Our proofs are for a UC security notion and thus immediately imply composability with higher-level protocols. *(iv)* As part of evaluating the practical performance of our Fabric$^*_{\mathrm{BB}}$, we have implemented and provide the first benchmarks for the underlying Fabric*.

C. Intended Security Properties of Fabric$^*_{\mathrm{BB}}$

We want to show accountability w.r.t. two security properties for Fabric$^*_{\mathrm{BB}}$: Firstly, we consider the standard notion of consistency, i. e., clients can read a (growing) prefix of the same globally unique state of the BB. Secondly, we consider a novel BB property called *smart read*.

Smart Read. Intuitively speaking, a BB with smart read offers two types of read operation: The standard read operation to retrieve the full state of the BB (we call this *full read* in what follows) and a second operation, called *smart read*. A smart read is closely related to and can be implemented by smart contracts offered by distributed ledgers: it allows clients to instruct the BB to run a (potentially arbitrary) algorithm on the BB state and then return the output. If the BB is secure, then the output of such a smart read must be *correct* and *fresh*, i. e., *(i)* it was obtained by running the algorithm requested by the client and *(ii)* the input to the algorithm was a prefix of the current global state that is at least as long as any prefix previously read by the client, either as part of a full read or during a smart read.

Thus, similar to smart contracts for distributed ledgers, smart reads allow for outsourcing tasks to the BB that would usually be performed in higher-level protocols. For example, in e-voting protocols [1, 41, 49] it is often necessary to perform verification checks on the state of the BB. It is of course possible for, e. g., a voter to just query the entire state of the BB via a full read and then perform those checks locally, which is what systems have been doing so far. By using a BB with smart read, it is possible to outsource these verification checks to the BB and only obtain the (correct) output.

Smart Read Implies Many Standard BB Properties. If a BB provides (secure) smart read, then it also provides many further standard BB security properties as special cases. This is because often clients can choose a suitable algorithm that computes a view on the BB's state which has the desired security properties, even if the state itself might not have these properties. Security then immediately follows from the correctness property of smart reads. For example:

No Data Injection: Given a list of eligible parties, define the algorithm used during a smart read to return all entries of the BB that are signed by a party in that list, i. e., all entries that were not injected.

Non-clashing: Just as for "no data injection", a smart read algorithm that filters out and removes any clashing items, e. g., by always returning the item that was submitted earlier, can be used to ensure non-clashing.

Message Validity: Given a message validation procedure, a smart read that returns all BB items which are valid according to that procedure can be used to implement message validity.

Receipt-consistency can also be implemented via smart reads but requires a more involved construction. We provide more details regarding receipt consistency in Appendix **D.3**.

Formalization of these security properties. We formalize accountability w.r.t. consistency and smart read by providing an instantiation of our customization framework $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB}}$. We refer to this instantiation as $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$ in what follows.

The functionality $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$ sets $\mathsf{Sec}^{\mathsf{acc}} = \{\mathtt{consistency}, \mathtt{smartRead}\}$. It is parameterized with an arbitrary but fixed set of algorithms that are identified via IDs $1, \ldots, n$. This set specifies all algorithms that can be executed via a smart read, where each algorithm takes as input a prefix of the current global state of the BB as well as, optionally, additional input from the client. To distinguish different types of read requests, we use the auxiliary input msg added to read requests in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB}}$: $msg = 0$ indicates a full read whereas $msg = (id, clientInput), id \in \{1, \ldots, n\}$ indicates a smart read using the algorithm with ID id and additional input $clientInput$. Security properties are then formalized via the following instantiations of subroutines:

- $\mathcal{F}_{\mathrm{write},\mathsf{FAB}^*_{\mathrm{BB}}}$ discards inputs that do not follow the input format of Fabric$^*_{\mathrm{BB}}$.
- $\mathcal{F}_{\mathrm{read},\mathsf{FAB}^*_{\mathrm{BB}}}$ works differently depending on the type of read request, i. e., the auxiliary input msg. For a full read with $msg = 0$, $\mathcal{F}_{\mathrm{read},\mathsf{FAB}^*_{\mathrm{BB}}}$ captures accountability w.r.t. consistency by using the same logic as $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$ in Figure 2. For smart reads with $msg = (id, clientInput)$, $\mathcal{F}_{\mathrm{read},\mathsf{FAB}^*_{\mathrm{BB}}}$ first checks whether the property has already been broken, i. e., $\mathsf{brokenProps}[\mathtt{smartRead}] = \mathtt{true}$, due to a verdict obtained by the public judge. If so, then the adversary $\mathcal{A}$ is allowed to choose the output freely. Otherwise, if the property still holds true, the adversary $\mathcal{A}$ is expected to provide a prefix p of the current global state $\mathsf{msglist}$ that is used as input for the smart read. $\mathcal{F}_{\mathrm{read},\mathsf{FAB}^*_{\mathrm{BB}}}$ verifies that p is strictly growing, i. e., larger than all prefixes previously used to respond to any read of the same client. If so, then $\mathcal{F}_{\mathrm{read},\mathsf{FAB}^*_{\mathrm{BB}}}$

runs the algorithm with ID id on BB state p and additional input $clientInput$ and instructs $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$ to return the result to the client.

- $\mathcal{F}_{\mathrm{update},\mathsf{FAB}^*_{\mathrm{BB}}}$ uses the same logic as the `Update` procedure in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$ (cf. Figure 2) to ensure that updates to the global msglist are append-only. It further checks that any message added to msglist follows the message format of Fabric*.
- $\mathcal{F}_{\mathrm{updRnd},\mathsf{FAB}^*_{\mathrm{BB}}}$ always allows time updates since we do not formalize any time-based security properties.

D. Deriving the Accountable BB Fabric$^*_{\mathrm{BB}}$ from Fabric**

In what follows, we define our Fabric$^*_{\mathrm{BB}}$ based on Fabric*. To obtain Fabric$^*_{\mathrm{BB}}$, we consider one Fabric* channel with the following smart contracts: *(i)* one write contract that appends a message to the ledger state without modifying or deleting any prior messages, *(ii)* one read-only contract with ID 0 to implement full reads by returning the current ledger state of the peer to the client, and *(iii)* n read-only contracts to implement smart reads. That is, for each algorithm alg supported by $\mathcal{F}^{\mathrm{acc}}_{\mathsf{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$, we use a corresponding smart contract sc in Fabric* which runs alg to obtain the result out. The smart contract then returns the tuple (ctr, out), where ctr is the length of the ledger state that was used to run alg.

Fabric$^*_{\mathrm{BB}}$ is then built on top of this Fabric* instance by adding some additional client logic. That is, the client keeps an indicator $recentState \in \mathbb{N}$ for storing the length of the most recent state used for full or smart read. Now, to write a message msg to the BB, clients use the write contract with input msg and then follow the standard Fabric* logic such that msg is added to the ledger state of the underlying Fabric* channel.

To run a full read in Fabric$^*_{\mathrm{BB}}$, clients start a transaction proposal for the full read smart contract in the underlying Fabric* channel. After obtaining and verifying the signature of the resulting endorsement[8] according to the standard Fabric* logic, clients additionally check for the output out, which is the current ledger state as seen by the peer, whether $|out| \geq recentState$. If so, then the client accepts out and it sets $recentState := |out|$. Otherwise, the client discards the response. Intuitively, this is needed to ensure that full reads in Fabric$^*_{\mathrm{BB}}$ return strictly growing prefixes of the ledger state and also that data is fresh compared to former smart reads.

To run a smart read in Fabric$^*_{\mathrm{BB}}$, clients start a transaction proposal for the corresponding smart read contract in the Fabric* channel. After obtaining and verifying the signature of the resulting endorsement according to the standard Fabric* logic, clients additionally verify for the output (ctr, out) that $ctr \geq recentState$. If so, the client returns out (and updates $recentState$ to ctr). Intuitively, this is necessary to ensure in Fabric$^*_{\mathrm{BB}}$ that the smart read was performed on a growing prefix of the state even if different peers are used.

Note that smart reads in Fabric$^*_{\mathrm{BB}}$ are implemented via *read-only* smart contracts in Fabric* which are evaluated off-chain. Hence, a smart read essentially consists of a client sending a signed request to a peer, the peer running the requested algorithm in native code, and the peer sending the signed output to the client. The runtime of a smart read in Fabric$^*_{\mathrm{BB}}$ is thus essentially the same as if the client were to run the same algorithm locally; there is only a negligible additional overhead due to sending two network messages and computing/verifying two signatures. This is a feature of Fabric$^*_{\mathrm{BB}}$ which in turn leverages a feature of Fabric*.

Modeling Fabric$^*_{\mathrm{BB}}$ as a (real) UC protocol. We model Fabric$^*_{\mathrm{BB}}$ as a protocol $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ in the iUC model [12], with the structure of the resulting protocol depicted in Figure 5. We provide a formal specification of all machines in Appendix F. Here, we provide a high-level overview of $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$.

One session of $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ models one Fabric$^*_{\mathrm{BB}}$ instance. There can be several sessions of $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ running in parallel. $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ mainly consists of the machines `client`, `peer`, and `orderer` – one machine instance per protocol participant. The sets of client, peer, and orderer identities within each session are arbitrary but fixed and the corresponding machines follow the specification of the Fabric$^*_{\mathrm{BB}}$ protocol. In each $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ session, there is a single `kafka` instance. The `kafka` instance models an ideal but accountable consensus service that is under the control of one party. This models the realistic setting that the entire Kafka cluster is carried out by one service provider in one data center. In particular, `kafka` signs the sequence of messages it distributes to orderers. $\mathcal{F}_{\mathrm{init}}$ provides the channel setup to the

[8]As common for read requests in blockchains, we expect that clients query only one peer for a read request. However, one could also model, e. g., that clients query several peers and only output the response to the environment $\mathcal{E}$ if all peers provide the same readset and output.

participants, we use a random oracle $\mathcal{F}_{\text{ro}}$ to model hash functions, and the ideal certificate functionality $\mathcal{F}_{\text{cert}}$ (see, e. g., [32]) captures secure digital signatures with a PKI.

$\mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}}$ allows for dynamic corruption of all participants. There are no assumptions on the network and communication is unprotected, i. e., there are no authenticated or secure channels.

E. Security Analysis of Fabric$^*_{\text{BB}}$

Showing security of Fabric$^*_{\text{BB}}$, as for any other accountability-based protocol, involves two major steps. Firstly, it is necessary to establish the exact evidence required by and the procedure that a public judge can use to detect misbehavior and blame parties in Fabric$^*_{\text{BB}}$. Secondly, we then have to show that Fabric$^*_{\text{BB}}$ together with this specific judge is a secure realization of $\mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}$ and hence achieves all desired accountability-based properties. In this sense, the judge can also be seen as part of the specification of Fabric$^*_{\text{BB}}$ that we complete in the first step.

Specifying the public judge. Our judge collects evidence from clients and peers. More specifically, (honest) clients forward accepted responses to full and smart read requests, i. e., the signed endorsements containing the desired output generated by a peer. (Honest) peers forward all blocks that they have accepted to the judge. This captures the following situation in reality: at any point in time one or more clients suspecting misbehavior can come together to share or publish their knowledge. They can then run the judging procedure on this information. If the procedure detects any issues such as inconsistent full reads or smart reads that were not executed on the data obtained from full reads, then peers that have been involved with these requests will be able to prove their innocence by showing their current copy of the blockchain.

We define the judging procedure as follows. In what follows, we group logical blocks according to the property that is checked and the corresponding party that is affected:

Validity (Peers): The judge checks whether blocks provided by peers are valid, i. e., they have the correct format as required by Fabric$^*_{\text{BB}}$ and all signatures contained therein are valid. If a peer provides an invalid block as evidence, then he violates the Fabric$^*_{\text{BB}}$ protocol (honest peers only provide accepted and therefore valid blocks as evidence) and the judge therefore renders a verdict blaming those peers. Note that rendering a verdict also stops the judging procedure.

This initial check is necessary to be able to detect other types of misbehavior that might break, e. g., consistency.

Consistency (Peers): If correctness still holds in the current run, the judge checks whether *consistency* has been broken already for peers, i. e., purely based on the blocks that peers provided. This is the case iff there are two blocks $\mathcal{B}_1, \mathcal{B}_2$ reported by peers with the same ID but differing bodies (that are signed and hence were generated by two not necessarily distinct orderers according to the previous check). In this case, the judge computes the verdict as follows: the judge first checks whether there are two different Kafka messages with the same ID (and, by the previous check, valid signatures of the Kafka cluster) in $\mathcal{B}_1$ and $\mathcal{B}_2$. If so, then the service provider running the Kafka cluster has misbehaved and thus the judge blames him. Otherwise, all blocks are derived from the same Kafka message stream, i. e., the two blocks have to differ due to a different number of messages. Since the block-cutting algorithm of Fabric* and thus of Fabric$^*_{\text{BB}}$ is deterministic and always cuts blocks at the same position(s), irrespective of the length of the message stream that is being processed, we have that at least one orderer has misbehaved. The judging procedure thus re-runs the block-cutting algorithm on the Kafka message stream and blames all orderers that have signed blocks that differ from the result.

Note that if there has not been a verdict so far, then the judge has successfully computed a globally unique sequence of messages that is consistent with the view of all peers (who have provided evidence).

Consistency (Clients): If a client reports a response to a full read request which contains an ordered sequence of messages seq and is signed with a valid signature from a peer p, then the judge checks whether seq is a prefix of the previously reconstructed globally unique sequence of messages. If seq is not a prefix, then the peer p did not or was unable to provide evidence that seq is a copy of the state of the blockchain. Since honest peers always can and will provide this evidence to show their innocence, the judge blames this peer in a verdict.

Smart Read (Clients): Finally, if a client reports a response to a smart read which is validly signed by a peer p, then the judge proceeds as follows. Recall that such a response is an endorsement on the smart contract output (ctr, out), where the (signed) endorsement also contains the client input $clientInput$ of the client as well as the ID id of the smart contract. The judge takes the prefix pre of the globally unique message sequence computed above up to position

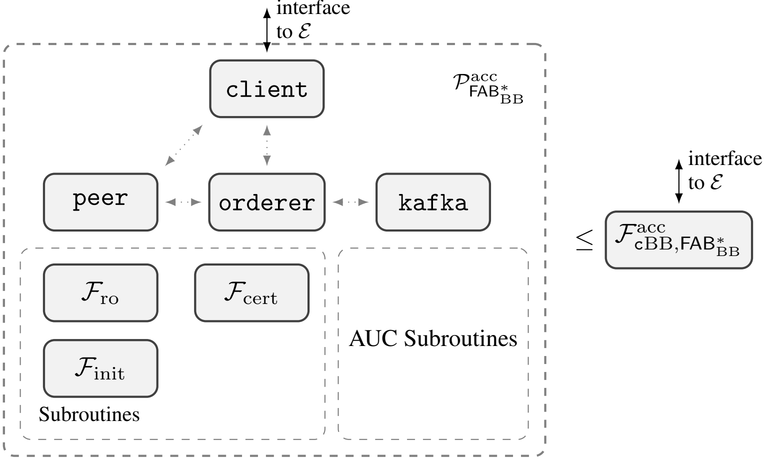


Fig. 5: Figure of Theorem 2. All machines have access to all (AUC) subroutines. $\mathcal{A}$ is connected to all machines. Dotted arrows indicate intended message flows.

ctr and then simulates the deterministic smart contract with ID id on inputs pre and $clientInput$. If the simulated output is different from (ctr, out), then the peer p misbehaved by providing an incorrect result to the client (or by not providing his entire copy of the chain to show his innocence) and hence is blamed in a verdict.

Altogether, by the above reasoning, it already follows that verdicts of this judge are fair, i. e., never blame an honest party. Intuitively, we also have that the properties of consistency and smart read still hold true for all (honest) clients as long as the judge does not output a verdict: In that case, the judge was able to compute a globally unique message sequence such that both full reads and smart reads of all clients (that have provided evidence) were computed correctly from prefixes of that message sequence. While the judge does not check whether those results were computed based on growing prefixes, this is done by the clients themselves.

Security proof. We can show the following security result as also depicted in Figure 5:

Theorem 2 (Informal)**.** *Let $\mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}$ be as defined Section IV-C with an arbitrary set of deterministic smart read algorithms. Let $\mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}}$ with the public judge as defined above. Then,*

$$\mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}} \text{ UC-realizes } \mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}.$$

The description and intuition of the public judge above serve as a proof sketch. We provide the formal proof of Theorem 2 and the description of the judge as an iUC machine in Appendix F. As part of Theorem 2, we show that FabricBB provides smart read. As detailed in Section IV-C, it hence directly follows that $\text{Fabric}^*_{\text{BB}}$ also provides several additional security properties.

As an interesting side note, our formal security proof is structured into two separate steps. First, in a rather simple initial step we show that an ideal accountable ledger $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ with certain properties realizes $\mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}$. By this, we not only establish the first formal definition of an accountable ideal ledger functionality $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ which is of independent interest. We also formally verify the corresponding folk wisdom: ledgers with specific properties can be used to build BBs. The bulk of the proof then shows that $\mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}}$ is such an accountable ledger, i. e., realizes $\mathcal{F}^{\text{acc}}_{\text{ledger}}$. By transitivity of UC security, this gives the overall result.

F. Discussion

In this section, we discuss important aspects and features of our $\text{Fabric}^*_{\text{BB}}$ construction and security result.

Summary of Assumptions and Abstractions From Reality. In our model of $\text{Fabric}^*_{\text{BB}}$ (cf. Figure 5), we use $\mathcal{F}_{\text{cert}}$ which can be UC-realized with an EUF-CMA secure signature scheme and a PKI for distributing public keys of the parties running the BB [17]. We use a random oracle $\mathcal{F}_{\text{ro}}$ as a setup assumption capturing ideal hash functions. Beyond these standard cryptographic assumptions, we also assume that all parties know and agree on the parameters of the underlying Fabric* channel. This is a standard assumption in the distributed ledger space [6, 30, 35] and formalized

via $\mathcal{F}_{\text{init}}$, which distributes the (unbounded but statically fixed) set of participants, genesis block, and smart contracts to all parties. Finally, we abstract from the internals of the Kafka cluster and instead model this as a single machine `kafka`. We note, however, that this machine still captures the capabilities of a malicious service provider running the Kafka cluster that might choose to deviate from the intended protocol, e. g., by providing inconsistent outputs.

While we model the public judge in $\text{Fabric}^*_{\text{BB}}$ as incorruptible, this is actually not a trust assumption but rather reflects the fact that the judging algorithm defined in Section IV-E can be executed by anyone, including external observers, given the needed evidence. Hence, if any party such as a client does not want to trust others or is afraid of faulty judges, then that party can collect the signed statements and compute the verdict herself by following the correct judging procedure. Altogether, we therefore indeed show Theorem 2 *without introducing any assumptions on the network and without assuming that any parties running* $\text{Fabric}^*_{\text{BB}}$ *are trusted/honest.*

Practical Security Guarantees of the Accountable $\text{Fabric}^*_{\text{BB}}$. As already explained in the introduction, accountability is a very different approach for obtaining security guarantees compared to preventive security, with both approaches complementing each other. Notably, a preventively secure BB provides the guarantee that, e. g., consistency always holds true no matter what malicious parties do as long as certain assumptions, including the existence of trusted parties are met (cf. Section VI).

In our accountable $\text{Fabric}^*_{\text{BB}}$ a security property such as consistency can in principle be broken by malicious parties. For example, a malicious Kafka cluster can decide to deviate from the protocol by providing inconsistent message sequences to orderers. This in turn breaks consistency for clients of the BB. However, accountability w.r.t. consistency guarantees that based on the inconsistent data provided to clients it is possible to identify a misbehaving party along with publicly verifiable cryptographic evidence for the misbehavior such that this party can be held accountable for the inconsistency. Relying on *deterrence*, as opposed to ensuring that properties cannot be broken at all as in preventive security, is the main reason why $\text{Fabric}^*_{\text{BB}}$ can still provide reasonable security guarantees even when *all* parties running the BB are untrusted and might misbehave.[9]

For a practical deployment of $\text{Fabric}^*_{\text{BB}}$ it is therefore necessary to determine suitable penalties which can indeed act as deterrence, where "suitability" depends on the application at hand and how much misbehaving parties stand to gain from breaking, e. g., consistency. This can be determined by a *deterrence analysis* (cf. [33]).

Using Other Distributed Ledgers. In principle, the ideas that we used to construct $\text{Fabric}^*_{\text{BB}}$ based on Fabric* in Section IV-D can also be applied more generally to other distributed ledgers as long as those ledgers meet certain requirements.

An obvious requirement for the underlying ledger is accountable w.r.t. to consistency. As part of this, the ledger should provide *finality* [2] which ensures that transactions in the state/blocks in the chain will not be revoked/changed at a later point in time. Without finality, the state of the chain and hence the output of the BB might be inconsistent between different points in time even when all parties are honest/trusted, i. e., no party can be held accountable. This rules out certain common ledgers such as Bitcoin which does not provide finality due to its longest chain rule (but can be shown, under certain assumptions, to achieve preventive security of consistency [6, 30]).

Another major requirement is the accountable correct execution of general smart contracts which is necessary to implement accountable smart reads. In most distributed ledgers, obtaining correctness for smart contract executions requires submitting the contract call to the chain, including establishing consensus over the outputs. This results in more overhead compared to the read-only smart contracts offered by Fabric*. However, it might be possible to implement the off-chain read-only smart contract system of Fabric* on top of other distributed ledgers to support smart reads with similar performance.

Given these considerations, the most promising other ledger, beside Fabric*, for applying our BB construction is the one proposed in [58]. This is the so far only other distributed ledger with proven accountability (but which requires some honest parties). We leave exploring this to future work.

Using $\text{Fabric}^*_{\text{BB}}$ as Building Block for Secure BB Applications. Since Theorem 2 is a UC security result which supports full composability, one can easily re-use this security result in the context of applications/higher-level

[9]This is also why Theorem 2 does not contradict the well-known FLP result [29], which roughly states that deterministic consensus algorithms cannot establish consensus in an asynchronous network: $\text{Fabric}^*_{\text{BB}}$ does *not* guarantee consensus; it only guarantees that misbehavior will be detected whenever clients obtain inconsistent states.

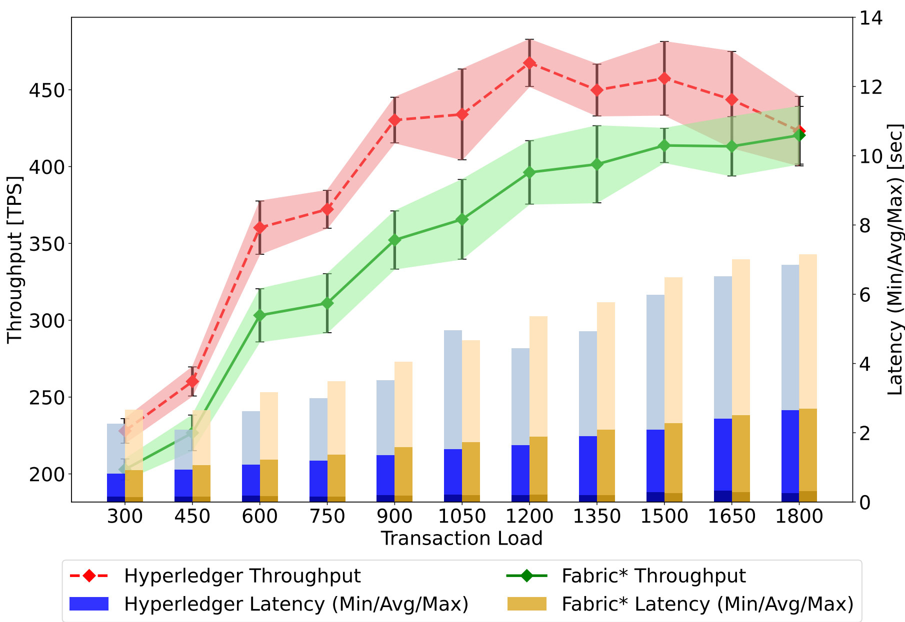


Fig. 6: Performance comparison of Hyperledger Fabric and Fabric* with an transaction size of 2000 bytes.

protocols $\mathcal{Q}$, say an MPC protocol, that want to use our BB. More specifically, protocol designers can model and prove the security of such an application $\mathcal{Q}$ based on the ideal functionality $\mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}$. As discussed in Section IV-C, $\mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}$ not just offers (accountability w.r.t.) consistency to $\mathcal{Q}$ but, due to smart read, can be instantiated via suitable algorithms to also offer several additional properties that might be required by $\mathcal{Q}$. Once the security of the combined protocol $(\mathcal{Q} \mid \mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}})$ has been proven based on the ideal BB, the UC composition theorem combined with Theorem 2 immediately implies as a corollary that $\mathcal{Q}$ using our $\text{Fabric}^*_{\text{BB}}$, i. e., the protocol $(\mathcal{Q} \mid \text{Fabric}^*_{\text{BB}})$, retains at least the same security guarantees.

This sets our work apart from existing provably secure BBs [24, 39, 46] which do not offer full and automatic composition of their security results with BB applications $\mathcal{Q}$. It also sets our work apart from applications $\mathcal{Q}$ that were shown secure based on ideal subroutines similar to $\mathcal{F}_{\text{BB}}$ (cf. Figure 1) which cannot be UC-realized by a real BB protocol $\mathcal{P}_{BB}$. Indeed, as observed in [21, 39] and unlike for our $\mathcal{F}^{\text{acc}}_{\text{cBB},\mathsf{FAB}^*_{\text{BB}}}$, security properties shown for applications $\mathcal{Q}$ based on an idealized "perfect" BB, such as $\mathcal{F}_{\text{BB}}$ (cf. Figure 1), might not apply once the ideal subroutine is replaced with any actual BB $\mathcal{P}_{BB}$.

V. Performance of Fabric* resp. $\text{Fabric}^*_{\text{BB}}$

$\text{Fabric}^*_{\text{BB}}$ is essentially an instantiation of Fabric* with a specific set of smart contracts as well as some added client logic that is negligible in terms of overhead. Thus performance of $\text{Fabric}^*_{\text{BB}}$ is directly determined by the performance of the underlying Fabric* channel. So far, Fabric* had neither been implemented nor evaluated for its practicality. In what follows, we therefore first fill this gap by providing the missing implementation, benchmarks, and also comparison to the original unmodified Hyperledger Fabric. We then discuss how those Fabric* results carry over to $\text{Fabric}^*_{\text{BB}}$ and their implications for practical deployments of our BB. We provide our Fabric* implementation and raw benchmark data at [34].

A. Test Setup

We conduct our benchmarks for a single Fabric/Fabric* channel that consists of four peer organizations, each of them running two peers and one CA. The underlying ordering service uses three orderers and a Kafka cluster with four Kafka brokers. The experiments utilize 17 Ubuntu 20.4 LTS VMs located in three different data centers.[10] of bwCloud: *(i)* each VM is equipped with a 60GB hard disk, *(ii)* instances belonging to the same organization are placed

[10]The data centers are located in Karlsruhe, Mannheim, and Ulm.

in the same data center, *(iii)* the Kafka-based ordering service is deployed across four 4-vCPU VMs with 8GB of RAM, and two 8-vCPU VMs with 16GB of RAM, *(iv)* peers are placed on different VMs equipped with 16-vCPUs and 32GB of RAM, *(v)* in each data center, there is one VM that generates workload, i.e., inputs for clients, via 20 Caliper Workers (see below).

B. Methodology

We measure throughput in terms of messages, resp. transactions, per second (TPS) and latency of our Fabric* implementation. We also measure both quantities for the unmodified Hyperledger Fabric v1.4.8 using Kafka-based ordering.

We performed the experiments with Hyperledger Caliper [44, 45], which is the standard tool for benchmarking Hyperledger Fabric instances. Caliper generates inputs for clients utilizing a range of smart contracts either at a fixed or dynamically varying rate and orchestrates the whole test execution. We set up Caliper such that it inputs a fixed rate of transactions per second, also called *transaction load*, into the tested Fabric* instances. We utilize the standard *create asset* benchmark/smart contract which writes transactions to the blockchain with a predefined transaction size. To cover a wide range of possible applications, we benchmark transaction sizes of 100, 1000, 2000, 4000, 8000, and 16000 bytes. For each transaction size, we further select a reasonable interval and sampling rate which yields good results. Each experiment is repeated ten times and results are then averaged.

In our experiments, we use the following configuration: *(i)* Client transactions solely require a single endorsement in their commitment to enter the blockchain. *(ii)* New blocks are created from 300 transactions or 2MB of data, whichever is reached first. *(iii)* Communication between entities is unencrypted.

C. Results for Fabric*

We present our results for a transaction size of 2000 bytes in Figure 6 with the other results being available in Appendix G, in particular Figure 46. Figure 6 shows the average throughput, including the standard deviation bounds, and the min/avg/max latencies (the time it takes for an issued transaction to be completed); min is in a darker blue/yellow and very close to the x-axis, avg is given in solid colors, and max is indicated via partially transparent colors. As becomes apparent from the figure, throughput gradually increases before becoming eventually more stationary.

As expected, the additional signature checks of Fabric* to achieve accountability result in a slight performance decrease compared to an unmodified Fabric instance: There is an average throughput loss of 10.7% with the worst case being 18.1%; this is irrespective of the transaction size. The change in latency depends on the transaction size, with average latency increasing at most by 16.7% and 24.6% for a transaction size of 2000 and 16000 bytes, respectively. For a transaction size of 2000 bytes, Fabric* reaches a maximum throughput of roughly 420 TPS. Using smaller transaction sizes of 100 and 1000 bytes, the maximum throughput in our setting can be further increased to roughly 640 and 500 TPS, respectively.

D. Implications for $\text{Fabric}^*_{\text{BB}}$

The *create asset* smart contract that we use for our benchmarks of Fabric* performs essentially the same operations as the write contract used by $\text{Fabric}^*_{\text{BB}}$. The main difference is that *create asset* uses a fixed transaction size. The transaction sizes that we benchmarked cover a common range of BB item sizes in particular, such as ballots from e-voting systems [27, 41, 49]. Hence, the TPS measured for Fabric* also show the throughput of $\text{Fabric}^*_{\text{BB}}$ in terms of how many write requests/items per second can be added to the BB.

This indicates that the performance of $\text{Fabric}^*_{\text{BB}}$ is sufficient for many practical applications of BBs. For example, in an electronic election with typical ballot sizes of about 2000 bytes, our results indicate that $\text{Fabric}^*_{\text{BB}}$ can add more than $220,000$ incoming ballots to its state in 10 minutes.

Regarding the performance of smart read operations in $\text{Fabric}^*_{\text{BB}}$, note that these are implemented using *off-chain read-only smart contract execution*. As already explained in Section IV-D, the runtime of a smart read for some algorithm is therefore basically the same as the runtime needed by a client for locally running the same algorithm. The only additional and generally negligible overhead is due to sending two network messages and computing/verifying two signatures. We therefore did not benchmark this operation separately as it would not provide any new insights.

VI. Related Work

In this section, we discuss and compare closely related works organized by area.

BB	Proven Properties	Security Type	Assumptions[a]	Proof Technique / Composability	Benchmarks
[24]	- no data injection - receipt-consistency - non-clashing	preventive	- trusted central component (WBB) - $> 2/3n$ honest peers - static corruption - synchronous network	simulation-based / ✗	✗
[46]	- liveness - persistence	confirmable	- $> 1/2m$ honest WBB parties - $> 2/3n$ honest peers - static corruption - partially synchronous network - global clock	game-based / ✗	✗
[39]	- final agreement	preventive	- ≥ 1 honest peers - dynamic corruption - asynchronous network - authenticated channels - phase-based	mechanized / ✗	✗
Fabric$^*_{\mathrm{BB}}$	- consistency - smart read, including: - no data injection - non-clashing	accountable	- *no* trusted BB component/party - dynamic corruption - asynchronous network - random oracle	UC / ✓	✓

[a] All works additionally assume secure (threshold) signature schemes as well as a trusted PKI or equivalently pre-distributed identities. All works also assume that all protocol algorithms and parameters, potentially including some initial state, are securely distributed.

TABLE I: Comparison of different provably secure BB solutions from the literature.

Accountability for BBs. Several works mention and discuss that accountability is a desirable mechanism that can or even should be used to protect the security of BBs, e. g., [22, 24, 36]. However, we are the first to formalize the notion of accountability for security properties of BBs and the first to prove accountability of a concrete BB.

Secure BBs in the UC literature. In UC literature, there exist many ideal BB functionalities, e. g., [4, 7, 14, 16–18, 61, 62]. Typically, these ideal functionalities ensure *(i)* consistency/consistent view, *(ii)* total ordering of BB items, *(iii)* append-only, and/or *(iv)* persistence preventively. As explained in detail in Section III-A, to the best of our knowledge all existing ideal BB functionalities in UC were only designed as setup assumptions and cannot be UC-realized by an actual implementation of a BB, which was not the goal of these works anyways.

With $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$ and $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB}}$ we provide the first ideal BB functionalities in UC that permit realizations and thus support composing security results for BBs with applications (cf. Section IV-F). We are also the first to formally prove UC-security for a concrete real-world BB.

Secure BBs in other literature. Outside of the UC literature, there exist several works on implementing BBs. Often, such implementations are directly integrated into and tailored towards the specific system at hand, e. g., [11, 20, 23, 24]. Security proofs are then given for the entire system, not for the integrated BB itself. The integrated BB cannot easily be used as a building block for other applications. There are also several papers such as [37, 57] that focus on constructing stand-alone BBs but do not formalize, let alone prove, their security properties.

There are only three works that formally analyze and prove the security of (stand-alone) BBs [24, 39, 46]. Table I summarizes and compares key properties of these BBs with our Fabric$^*_{\mathrm{BB}}$. Culnane et al. [24] construct a BB consisting of $n \in \mathbb{N}$ peers that receive and forward client inputs to a trusted component called the Web BB (WBB). They show, among others, that their construction achieves *receipt-consistency*.

Kiayias et al. [46] build on and improve the construction of [24] by, among others, distributing the central WBB among m parties half of which are assumed to be trusted. They then show that their BB provides *confirmable persistence* and *confirmable liveness*. Persistence intuitively states that, once an honest peer adds an item to its local view of the BB state, then all other honest peers either agree on the position of this item or do not have this item in their state (yet). This property is therefore closely related to our notion of consistency. Confirmability is essentially a strictly weaker version of accountability that does not require fairness of verdicts: if a security property is broken at some point, then one can identify a party where the property broke down. However, this party might not be malicious but might have been honestly following the protocol.

As a part of their work, Hirschi et al. [39] propose a conceptually very simple BB: n peers are responsible for signing incoming items and clients interpret items as part of the BB iff at least γ many BB peers have signed it. Hirschi et al. suggest selecting $\gamma = \lfloor n - \frac{n_h}{2} + 1 \rfloor$, where $n_h \geq 1$ is the number of peers that are assumed to be trusted. This construction therefore gives a tradeoff between trust assumption and performance/availability. Hirschi et al. show that their BB achieves *final agreement* which mainly states that, if a client receives a valid final state of the BB, then this is the same as for other clients who have received a valid final state. Note that this property is related to consistency and can be implemented on top of BBs providing consistency.[11]

The approach taken by Hirschi et al. is simple, and while not discussed in their paper, it might provide an alternative route for constructing an accountable BB without trusted parties. One of the main reasons why we have designed our BB based on the more complex Fabric* is due to performance: Hirschi et al. expect that their protocol needs further modifications and enhancements to achieve availability and scalability as required for practical deployments. Such extensions might then also require additional mechanisms to retain security guarantees. In contrast, Fabric/Fabric* is designed specifically for practical deployments with high throughput and therefore includes several scalability and availability mechanisms out of the box [2]. Another functional difference is that, unlike Fabric*, Hirschi et al.'s approach is not designed to and indeed does not establish a total order among the BB items.

Altogether, by leveraging accountability our work is the first to formally prove the security of a BB based only on standard cryptographic assumptions and without requiring any trusted BB component or network assumptions. $\text{Fabric}^*_{\text{BB}}$ is also the first BB that is shown to be UC secure and which can therefore be directly composed with higher-level protocols while retaining security results for the BB (cf. IV-F).

Alternative approaches to protect against malicious BBs. In recent works, researchers investigated how (e-voting) systems can be hardened to remain secure even if the underlying BB is malicious [21, 39]. In other words, while many works including ours focus on constructing secure BBs, these works (also) investigate how to deal with broken BBs at the next protocol level.

Distributed ledgers. There are numerous provably secure distributed ledgers, including provably UC-secure ones, (e. g., [5, 25, 30, 35, 47, 58]) which, according to folk wisdom, might be candidates for secure BBs. Before our work, this had not been formally verified for any ledger and might indeed require additional modifications as was the case for Fabric*. Also, all of the aforementioned works rely on strong honesty and/or network assumptions, such as honest (super-)majorities and networks without message loss. They are therefore not suitable for implementing BBs in many applications.

Furthermore, our work is the first to formalize and prove accountability of a DLT in a UC model as part of showing Theorem 2. More generally, so far the only DLTs with proven accountability properties (shown via non-UC and hence non-composable proof techniques) are Fabric* [31] and the relatively recent DLT construction presented by Shamis et al. [58] (cf. Section IV-F). Unlike Fabric*, the construction by Shamis et al. still requires some honest parties to achieve accountability.

VII. Acknowledgments

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program "Services Computing" as well as the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Project-ID 459731562. We thank Jonathan Gröber for supporting implementation and performance testing.

References

[1] B. Adida, "Helios: Web-based Open-Audit Voting," in *Proceedings of the 17th USENIX Security Symposium*, P. C. van Oorschot, Ed. USENIX Association, 2008, pp. 335–348.

[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).

[11] For example, some party publishes a special "end/final item" marker on the BB. Clients then accept an output as finalized only if it contains this marker and ignore anything after that marker. By consistency, any (finalized) output that a client accepts must therefore be the same as for all other clients.

[4] T. Attema, V. Dunning, M. H. Everts, and P. Langenkamp, "Efficient Compiler to Covert Security with Public Verifiability for Honest Majority MPC," in *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, ser. Lecture Notes in Computer Science, vol. 13269. Springer, 2022, pp. 663–683.

[5] 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.

[6] 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 I*, ser. Lecture Notes in Computer Science, vol. 10401. Springer, 2017, pp. 324–356.

[7] 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.

[8] C. Baum, E. Orsini, and P. Scholl, "Efficient Secure Multiparty Computation with Identifiable Abort," in *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, vol. 9985, 2016, pp. 461–490.

[9] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, E. V. Herreveghen, and M. Waidner., "Design, Implementation, and Deployment of the iKP Secure Electronic Payment System," *IEEE Journal on Selected Areas in Communications*, vol. 18, no. 4, pp. 611–627, 2000.

[10] X. Boyen, T. Haines, and J. Müller, "Epoque: Practical End-to-End Verifiable Post-Quantum-Secure E-Voting," in *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 272–291.

[11] C. Burton, C. Culnane, and S. A. Schneider, "vVote: Verifiable Electronic Voting in Practice," *IEEE Secur. Priv.*, vol. 14, no. 4, pp. 64–73, 2016.

[12] J. Camenisch, S. Krenn, R. Küsters, and D. Rausch, "iUC: Flexible Universal Composability Made Simple," in *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 11923. Springer, 2019, pp. 191–221, the full version is available at http://eprint.iacr.org/2019/1073.

[13] R. Canetti, "Universally Composable Security," *J. ACM*, vol. 67, no. 5, pp. 28:1–28:94, 2020.

[14] R. Canetti, K. Hogan, A. Malhotra, and M. Varia, "A Universally Composable Treatment of Network Time," in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 360–375.

[15] R. Canetti, A. Jain, and A. Scafuro, "Practical UC security with a Global Random Oracle," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014, pp. 597–608.

[16] R. Canetti, Y. T. Kalai, A. Lysyanskaya, R. L. Rivest, A. Shamir, E. Shen, A. Trachtenberg, M. Varia, and D. J. Weitzner, "Privacy-Preserving Automated Exposure Notification," Cryptology ePrint Archive, Tech. Rep. 2020/863, 2020.

[17] R. Canetti, D. Shahaf, and M. Vald, "Universally Composable Authentication and Key-Exchange with Global PKI," in *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 9615. Springer, 2016, pp. 265–296.

[18] I. Cascudo and B. David, "ALBATROSS: Publicly AttestabLe BATched Randomness Based On Secret Sharing," in *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III*, ser. Lecture Notes in Computer Science, vol. 12493. Springer, 2020, pp. 311–341.

[19] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[20] N. Chondros, B. Zhang, T. Zacharias, P. Diamantopoulos, S. Maneas, C. Patsonakis, A. Delis, A. Kiayias, and M. Roussopoulos, "D-DEMOS: A Distributed, End-to-End Verifiable, Internet Voting System," in *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 711–720.

[21] V. Cortier, J. Lallemand, and B. Warinschi, "Fifty Shades of Ballot Privacy: Privacy against a Malicious Board," in *IEEE 33rd Computer Security Foundations Symposium, CSF 2020, 22-25 July, 2020*. IEEE Computer Society, 2020.

[22] R. Cramer, R. Gennaro, and B. Schoenmakers, "A Secure and Optimally Efficient Multi-Authority Election Scheme," in *Advances in Cryptology — EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques*, ser. Lecture Notes in Computer Science, vol. 1233. Springer-Verlag, 1997.

[23] C. Culnane, P. Y. A. Ryan, S. A. Schneider, and V. Teague, "vVote: A Verifiable Voting System," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, pp. 3:1–3:30, 2015.

[24] C. Culnane and S. A. Schneider, "A Peered Bulletin Board for Robust Use in Verifiable Voting Systems," in *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*. IEEE Computer Society, 2014, pp. 169–183.

[25] 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.

[26] 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).

[27] R. del Pino, V. Lyubashevsky, G. Neven, and G. Seiler, "Practical Quantum-Safe Voting from Lattices," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 1565–1581.

[28] J. Feigenbaum, A. D. Jaggard, and R. N. Wright, "Open vs. Closed Systems for Accountability," in *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security, HotSoS 2014, Raleigh, NC, USA, April 08 - 09, 2014*. ACM, 2014, p. 4.

[29] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[30] 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.

[31] M. Graf, R. Küsters, and D. Rausch, "Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric," in *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*. Los Alamitos, CA, USA: IEEE, 2020, pp. 236–255.

[32] M. Graf, R. Küsters, and D. Rausch, "AUC: Accountable Universal Composability," Cryptology ePrint Archive, Tech. Rep. 1606, 2022.

[33] M. Graf, R. Küsters, and D. Rausch, "AUC: Accountable Universal Composability," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1148–1167.

[34] M. Graf, R. Küsters, D. Rausch, S. Egger, M. Bechtold, and M. Flinspach, "Fabric* - Impementation and Performance Testing," https://github.com/7Y61tY6W9vgScr65UKqm, 2023.

[35] M. Graf, D. Rausch, V. Ronge, C. Egger, R. Küsters, and D. Schröder, "A Security Framework for Distributed Ledgers," in *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. New York City, USA: ACM, 2021, pp. 1043–1064.

[36] S. Hauser and R. Haenni, "Modeling a Bulletin Board Service Based on Broadcast Channels with Memory," in *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 10958. Springer, 2018, pp. 232–246.

[37] J. Heather and D. Lundin, "The Append-Only Web Bulletin Board," in *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008, Malaga, Spain, October 9-10, 2008, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 5491. Springer, 2008, pp. 242–256.

[38] S. Heiberg and J. Willemson, "Verifiable Internet Voting in Estonia," in *6th International Conference on Electronic Voting: Verifying the Vote, EVOTE 2014, Lochau / Bregenz, Austria, October 29-31, 2014*. IEEE, 2014, pp. 1–8.

[39] L. Hirschi, L. Schmid, and D. A. Basin, "Fixing the Achilles Heel of E-Voting: The Bulletin Board," in *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 2021, pp. 1–17.

[40] D. Hofheinz and V. Shoup, "GNUC: A New Universal Composability Framework," *J. Cryptology*, vol. 28, no. 3, pp. 423–508, 2015.

[41] N. Huber, R. Küsters, T. Krips, J. Liedtke, J. Müller, D. Rausch, P. Reisert, and A. Vogt, "Kryvos: Publicly Tally-Hiding Verifiable E-Voting," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. ACM, 2022, pp. 1443–1457.

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

[43] ——, "fabricdocs: Chaincode for Developers," https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html, 2019, (Accessed on 06/16/2023).

[44] ——, "Github - hyperledger/caliper-benchmarks," https://github.com/hyperledger/caliper-benchmarks, 2022, (Accessed on 01/19/2023).

[45] ——, "Hyperledger caliper," https://www.hyperledger.org/use/caliper, 2022, (Accessed on 12/30/2022).

[46] A. Kiayias, A. Kuldmaa, H. Lipmaa, J. Siim, and T. Zacharias, "On the Security Properties of e-Voting Bulletin Boards," in *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11035. Springer, 2018, pp. 505–523.

[47] 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.

[48] R. Künnemann, I. Esiyok, and M. Backes, "Automated Verification of Accountability in Security Protocols," in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 2019, pp. 397–413.

[49] R. Küsters, J. Liedtke, J. Müller, D. Rausch, and A. Vogt, "Ordinos: A Verifiable Tally-Hiding E-Voting System," in *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*. IEEE, 2020, pp. 216–235.

[50] 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.

[51] R. Küsters, M. Tuengerthal, and D. Rausch, "The IITM Model: a Simple and Expressive Model for Universal Composability," *Journal of Cryptology*, vol. 33, no. 4, pp. 1461–1584, 2020.

[52] D. Lundin and P. Y. A. Ryan, "Human Readable Paper Verification of Prêt à Voter," in *European Symposium on Research in Computer Security (ESORICS 2008)*, 2008, pp. 379–395.

[53] D. Parkes, M. Rabin, S. Shieber, and C. Thorpe, "Practical Secrecy-preserving, Verifiably Correct and Trustworthy Auctions," in *Proceedings of the Eighth International Conference on Electronic Commerce (ICEC'06)*, 2006, pp. 70–81.

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

[55] M. Rivinius, P. Reisert, D. Rausch, and R. Küsters, "Publicly Accountable Robust Multi-Party Computation," in *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 2430–2449.

[56] P. Y. A. Ryan, D. Bismark, J. Heather, S. Schneider, and Z. Xia, "The Prêt à Voter Verifiable Election System," University of Luxembourg, University of Surrey, Tech. Rep., 2010, http://www.pretavoter.com/publications/PretaVoter2010.pdf.

[57] D. Sandler and D. S. Wallach, "Casting Votes in the Auditorium," in *2007 USENIX/ACCURATE Electronic Voting Technology Workshop, EVT'07, Boston, MA, USA, August 6, 2007.* USENIX Association, 2007.

[58] A. Shamis, P. Pietzuch, B. Canakci, M. Castro, C. Fournet, E. Ashton, A. Chamayou, S. Clebsch, A. Delignat-Lavaud, M. Kerner *et al.*, "IA-CCF: Individual Accountability for Permissioned Ledgers," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 467–491.

[59] 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.

[60] 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.

[61] D. Wikström, "A Universally Composable Mix-Net," in *Theory of Cryptography, First Theory of Cryptography Conference, TCC 2004, Proceedings*, ser. Lecture Notes in Computer Science, M. Naor, Ed., vol. 2951. Springer, 2004, pp. 317–335.

[62] D. Wikström, "Universally Composable DKG with Linear Number of Exponentiations," in *Security in Communication Networks, 4th International Conference, SCN 2004, Amalfi, Italy, September 8-10, 2004, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 3352. Springer, 2004, pp. 263–277.

[63] H. Zhong, Y. Sang, Y. Zhang, and Z. Xi, "Secure Multi-Party Computation on Blockchain: An Overview," in *Parallel Architectures, Algorithms and Programming - 10th International Symposium, PAAP 2019, Guangzhou, China, December 12-14, 2019, Revised Selected Papers*, ser. Communications in Computer and Information Science, vol. 1163. Springer, 2019, pp. 452–460.

Appendix

A. A Brief Introduction to the iUC Framework

This section provides a brief introduction to the iUC framework, which underlies all results in this paper. The iUC framework [12] is a highly expressive and user friendly model for universal composability. It allows for the modular analysis of different types of protocols in various security settings. This section is mainly taken verbatim from [35].

The iUC framework uses interactive Turing machines as its underlying computational model. Such interactive Turing machines can be connected to each other to be able to exchange messages. A set of machines $\mathcal{Q} = \{M_1, \ldots, M_k\}$ is called a *system*. In a run of $\mathcal{Q}$, there can be one or more instances (copies) of each machine in $\mathcal{Q}$. One instance can send messages to another instance. At any point in a run, only a single instance is active, namely, the one to receive the last message; all other instances wait for input. The active instance becomes inactive once it has sent a message; then the instance that receives the message becomes active instead and can perform arbitrary computations. The first machine to run is the so-called *master*. The master is also triggered if the last active machine did not output a message. In iUC, the environment (see next) takes the role of the master. In the iUC framework a special user-specified **CheckID** algorithm is used to determine which instance of a protocol machine receives a message and whether a new instance is to be created (see below).

To define the universal composability security experiment (cf. [12]), one distinguishes between three types of systems: protocols, environments, and adversaries. As is standard in universal composability models, all of these types of systems have to meet a polynomial runtime notion. Intuitively, the security experiment in any universal composability model compares a protocol $\mathcal{P}$ with another protocol $\mathcal{F}$, where $\mathcal{F}$ is typically an ideal specification of some task, called *ideal protocol* or *ideal functionality*. The idea is that if one cannot distinguish $\mathcal{P}$ from $\mathcal{F}$, then $\mathcal{P}$ must be "as good as" $\mathcal{F}$. More specifically, the protocol $\mathcal{P}$ is considered secure (written $\mathcal{P} \leq \mathcal{F}$) if for all adversaries $\mathcal{A}$ controlling the network of $\mathcal{P}$ there exists an (ideal) adversary $\mathcal{S}$, called *simulator*, controlling the network of $\mathcal{F}$ such that $\{\mathcal{A}, \mathcal{P}\}$ and $\{\mathcal{S}, \mathcal{F}\}$ are indistinguishable for all environments $\mathcal{E}$. Indistinguishability means that the probability of the environment outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$ is negligibly close to the probability of outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ (written $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$). The environment can also subsume the role of the network attacker $\mathcal{A}$, which yields an equivalent definition in the iUC framework. We usually show this equivalent but simpler statement in our proofs, i.e., that there exists a simulator $\mathcal{S}$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all environments.

A protocol $\mathcal{P}$ in the iUC framework is specified via a system of machines $\{M_1, \ldots, M_l\}$; the framework offers a convenient template for the specification of such systems. Each machine M_i implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. For example, a (real) protocol $\mathcal{P}_{\text{sig}}$ for digital signatures might contain a `signer` role for signing messages and a `verifier` role for verifying signatures. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly the environment subsuming a network attacker) via network interfaces. An instance of a machine M_i manages one or more so-called *entities*. An entity is identified by a

tuple $(pid, sid, role)$ and describes a specific party with party ID (PID) pid running in a session with session ID (SID) sid and executing some code defined by the role $role$ where this role has to be (one of) the role(s) of M_i according to the specification of M_i. Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances. More specifically, the I/O interfaces of both machines need to be connected to each other (because one machine specifies the other as a subroutine) to enable communication between entities of those machines.

Roles of a protocol can be either public or private. The I/O interfaces of private roles are only accessible by other (entities belonging to) roles of the same protocol, whereas I/O interfaces of public roles can also be accessed by other (potentially unknown) protocols/the environment. Hence, a private role models some internal subroutine that is protected from access outside of the protocol, whereas a public role models some publicly accessible operation that can be used by other protocols. One uses the syntax "$(\mathtt{pubrole_1}, \ldots, \mathtt{pubrole_n} \mid \mathtt{privrole_1}, \ldots, \mathtt{privrole_n})$" to uniquely determine public and private roles of a protocol. Two protocols $\mathcal{P}$ and $\mathcal{Q}$ can be combined to form a new more complex protocol as long as their I/O interfaces connect only via their public roles. In the context of the new combined protocol, previously private roles remain private while previously public roles may either remain public or be considered private, as determined by the protocol designer. The set of all possible combinations of $\mathcal{P}$ and $\mathcal{Q}$, which differ only in the set of public roles, is denoted by $\mathsf{Comb}(\mathcal{Q}, \mathcal{P})$.

An entity in a protocol might become corrupted by the adversary, in which case it acts as a pure message forwarder between the adversary and any connected higher-level protocols as well as subroutines. In addition, an entity might also consider itself (implicitly) corrupted while still following its own protocol because, e.g., a subroutine has been corrupted. Corruption of entities in the iUC framework is highly customizable; one can, for example, prevent corruption of certain entities during a protected setup phase.

The iUC framework supports the modular analysis of protocols via a so-called composition theorem:

Corollary 3 (Concurrent composition in iUC; informal)**.** *Let $\mathcal{P}$ and $\mathcal{F}$ be two protocols such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another protocol such that $\mathcal{Q}$ and $\mathcal{F}$ can be connected. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ agree on their public roles. Then $\mathcal{R} \leq \mathcal{I}$.*

By this theorem, one can first analyze and prove the security of a subroutine $\mathcal{P}$ independently of how it is used later on in the context of a more complex protocol. Once we have shown that $\mathcal{P} \leq \mathcal{F}$ (for some other, typically ideal protocol $\mathcal{F}$), we can then analyze the security of a higher-level protocol $\mathcal{Q}$ based on $\mathcal{F}$. Note that this is simpler than analyzing $\mathcal{Q}$ based on $\mathcal{P}$ directly as ideal protocols provide absolute security guarantees while typically also being less complex, reducing the potential for errors in proofs. Once we have shown that the combined protocol, say, $(\mathcal{Q} \mid \mathcal{F})$ realizes some other protocol, say, $\mathcal{F}'$, the composition theorem and transitivity of the $\leq$ relation then directly implies that this also holds true if we run $\mathcal{Q}$ with an implementation $\mathcal{P}$ of $\mathcal{F}$. That is, $(\mathcal{Q} \mid \mathcal{P})$ is also a secure realization of $\mathcal{F}'$. Please note that the composition theorem does not impose any restrictions on how the protocols $\mathcal{P}$, $\mathcal{F}$, and $\mathcal{Q}$ look like internally. For example, they might have disjoint sessions, but they could also freely share some state between sessions, or they might be a mixture of both. They can also freely share some of their subroutines with the environment, modeling so-called globally available state. This is unlike most other models for universal composability, such as the UC model, which impose several conditions on the structure of protocols for their composition theorem.

B. Notation in Pseudo Code

ITMs in our paper are specified in pseudo code. Most of our pseudo code notation follows the notation introduced by Camenisch et al. [12]. To ease readably of our figures, we provide a brief overview over the used notation here. This section is mainly taken verbatim from [35].

The description in the main part of the ITMs consists of blocks of the form **recv** ⟨*msg*⟩ **from** ⟨*sender*⟩ **to** ⟨*receiver*⟩ **s.t.** ⟨condition⟩:⟨code⟩ where ⟨*msg*⟩ is an input pattern, ⟨*sender*⟩ is the receiving interface (I/O or NET), ⟨*receiver*⟩ is the dedicated receiver of the message and ⟨condition⟩ is a condition on the input. ⟨code⟩ is the (pseudo) code of this block. The block is executed if an incoming message matches the pattern and the condition is satisfied. More specifically, ⟨*msg*⟩ defines the format of the message m that invokes this code block. Messages contain local variables, state variables, strings, and maybe special characters. To compare a message m to a message pattern *msg*, the values of all global and local variables (if defined) are inserted into the pattern. The resulting pattern p is then compared to m,

where uninitialized local variables match with arbitrary parts of the message. If the message matches the pattern p and meets ⟨condition⟩ of that block, then uninitialized local variables are initialized with the part of the message that they matched to and ⟨code⟩ is executed in the context of ⟨*receiver*⟩; no other blocks are executed in this case. If m does not match p or ⟨condition⟩ is not met, then m is compared with the next block. Usually a **recv from** block ends with a **send to** clause of form **send** $\langle \overline{msg} \rangle$ **to** $\langle \overline{sender} \rangle$ where $\overline{msg}$ is a message that is send via output interface $\overline{sender}$.

If an ITM invokes another ITM, e. g., as a subroutine, ITMs may expect an immediate response. In this case, in a **recv from** block, a **send to** statement is directly followed by a **wait for** statement. We write **wait for** $\langle \overline{msg} \rangle$ **from** $\langle \overline{sender} \rangle$ **s.t.** ⟨*condition*⟩ to denote that the ITM stays in its current state and discards all incoming messages until it receives a message m matching the pattern $\overline{msg}$ and fulfilling the **wait for** condition. Then the ITM continues the run where it left of, including all values of local variables.

To clarify the presentation and distinguish different types of variables, constants, strings, etc. we follow the naming conventions of Camenisch et al. [12]:

1. (Internal) state variables are denoted by sans-serif fonts.
2. Local (i.e., ephemeral) variables are denoted in *italic font*.
3. Keywords are written in **bold font** (e. g., for operations such as **sending** or **receiving**).
4. Commands, procedure, function names, strings and constants are written in `teletype`.

To increase readability, we use the following notation:

- For a set of tuples K, $K.\texttt{add}(_)$ adds the tuple to K.
- For a string S, $S.\texttt{add}(_)$ concatenates the given string to S.
- For a verdicts v_1 and v_2, we define $v_1.\texttt{add}(v_2) := v_1 \wedge v_2$.
- $K.\texttt{remove}(_)$ removes always the first appearance of the given element/string from the list/tuple/set/string K.

We use the following additional nomenclature from [12]:

- $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ denotes the currently active entity and $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ denotes the entity which called the currently active ITM.
- The macro $\mathbf{corr}(pid, sid, role)$ is simply a shortcut to invoke the ITM of $(pid, sid, role)$ and query it for its corruption status.
- The macro $\mathbf{init}(pid, sid, role)$ triggers the initialization of $(pid, sid, role)$ and returns the activation to the calling ITM.
- $K.\texttt{contains}(_)$ checks whether the requested element/string is contained in the list/tuple/set/string K and returns either `true` or `false`.
- We further assume that each element as a tuple in a list or set can be addressed by each element in that tuple if it is a unique key.
- Elements in a tuple are ordered can be addressed by index, starting from 0. We write $[n] = \{1, \ldots, n\}$.
- For tuples, lists, etc. we start index counting at 0.

C. A Brief Introduction to AUC

As already discussed in Section II, AUC mainly provides a generic transformation that allows replacing hard-coded preventive security properties of an ideal functionality $\mathcal{F}$ with accountability properties in any ideal functionality. Typically, AUC guarantees properties for honest users/components. The transformation includes two steps. The first step adds static code to the ideal functionality (cf. Figure 7 to 9). AUC's first transformation step adds the "infrastructure" and additional *judge* and *supervisor* roles. The second transformation step then changes the behavior of the ideal functionality to capture accountability-based security. The result of the transformation is the accountable variant of $\mathcal{F}$, called $\mathcal{F}^{\mathrm{acc}}$.

We provide only a brief presentation of AUC which focuses on adding *public individual accountability properties* to an ideal functionality. We refer an interested reader to [33] which explains all details and possibilities of AUC.

Step 1. The first transformation step adds a public *judge* to $\mathcal{F}$ as well as a *supervisor* (cf. Figure 7 to 9). As stated earlier, accountability incentivizes parties to behave honestly by identifying and blaming parties who violate a protocol's accountability properties. In the publicly accountable variant of AUC, a public *judge* – an incorruptible

Additional roles: judge, supervisor
Additional protocol parameters: {They may be polynomially checkable predicates

- $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$ {Accountability properties
- $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$ {Assumption-based security properties
- $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$ {set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)
- $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$ {set of entities/IDs where properties are ensured via assumptions

Additional subroutines: $\mathcal{F}_{\text{judgeParams}}$
Additional Corruption behavior:

- **AllowCorruption**$(pid, sid, role)$:
  Do not allow corruption of $(pid, sid, \texttt{supervisor})$.
  if $role = \texttt{judge}$:
  send $(\mathsf{Corrupt}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$ {$\mathcal{F}_{\text{judgeParams}}$ decides whether judges can be corrupted
  wait for b
  return b
- **DetermineCorrStatus**[a]$(pid, sid, role)$:
  if $role = \texttt{judge}$: {$\mathcal{F}_{\text{judgeParams}}$ may determine a judge's corruption status
  send $(\texttt{CorruptionStatus?}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$
  wait for b; **return** b
- **AllowAdvMessage**$(pid, sid, role, \mathsf{pid}_{\mathsf{receiver}}, \mathsf{sid}_{\mathsf{receiver}}, \mathsf{role}_{\mathsf{receiver}}, m)$
  Do not allow sending messages to $\mathcal{F}_{\text{judgeParams}}$. {$\mathcal{A}$ is not allowed to invoke $\mathcal{F}_{\text{judgeParams}}$ in the name of corrupted parties.

Additional internal state:

- $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\texttt{true}, \texttt{false}\}$ {Stores broken security properties per judge/id, initially false $\forall$entries
- $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$ {Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially ε
- $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\texttt{true}, \texttt{false}\}$ {Stores broken security assumptions per id, initially false $\forall$entries
- $\mathsf{corruptedIntParties} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{b} \cup \{\texttt{judge}, \texttt{supervisor}\})$, initially $\emptyset$ {The set of corrupted internal parties $(pid, sid, role)$

[a] **DetermineCorrStatus** allows protocol designers to specify whether an entity that is currently not directly controlled by the attacker should nevertheless consider itself to be corrupted. E.g., a local judge will typically consider itself to be corrupted already if its corresponding party is corrupted.

[b] $\mathsf{Roles}_{\mathcal{F}}$ is the set of (main) roles provided by $\mathcal{F}$ to the environment. For example, $\mathsf{Roles}_{\mathcal{F}} = \{\texttt{signer}, \texttt{verifier}\}$ for an ideal signature functionality $\mathcal{F} := \mathcal{F}_{\text{sig}}$.

Fig. 7: Parameters and state added by the transformation $\mathcal{T}_1(\mathcal{F})$ to an ideal functionality $\mathcal{F}$.

entity – is responsible for rendering verdicts in this case and she is also responsible for guaranteeing the fairness of the verdict.

The (ideal) judge in AUC allows $\mathcal{A}$ to break accountability properties defined in the parameter $\mathsf{Sec}^{\mathsf{acc}}$, e. g., $\texttt{consistency} \in \mathsf{Sec}^{\mathsf{acc}}$, in exchange for a verdict via the `BreakAccProp` interface. Verdicts in AUC are positive boolean formulas consisting of terms of the from $\mathsf{dis}(A)$ where A is a protocol participant. The judge ensures that *(i)* there is indeed a verdict if an accountability property is broken and *(ii)* the verdict is fair, i. e., the judge never blames honest parties for misbehavior. The detailed checks on the verdict or conditions when accountability properties may break are not a priori fixed in AUC. AUC requires to customize a new additional subroutine $\mathcal{F}_{\text{judgeParams}}$ to fix the judge's details.

The judge provides further interfaces to higher-level protocols: *(i)* `GetVerdict` provides access to the judge's rendered verdicts and *(ii)* `GetJudicialReport`[12] allows the judge to provide consolidated information for modular security/accountability analysis. While we construct our ideal functionalities such that they can be used in modular security analysis, we will not focus on this feature in what follows.

Public judges in real protocols formalize *(i)* the judging procedure (based on publicly available or checkable data) used for rendering verdicts, *(ii)* inputs and hence evidence needed for obtaining the verdict, and *(iii)* which parties are supposed to provide which information as evidence.

The *supervisor* provides higher-level protocols access to the corruption status of internal protocol participants

[12] Technically, AUC uses a customizable subroutine called $\mathcal{F}_{\text{judgeParams}}$ to define judicial reports for the protocol at hand as well as it details the restrictions and rules for breaking accountability properties, e. g., whether verdicts need to be individual or not.

Additional code for the judge role:

recv $(\texttt{BreakAccProp}, \mathit{verdict}, \mathit{toBreak})$ **from** NET[a] **to** $(\mathit{pid}, \mathit{sid}, \texttt{judge})$
s.t. $\mathit{toBreak} \subseteq \mathsf{Sec}^{\mathsf{acc}} \times \mathsf{pids}_{\mathsf{judge}} \wedge \mathit{verdict}$ maps from $\mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$:
$(\mathit{successful}, \mathit{leakage}) \leftarrow \texttt{breakAttempt}(\mathit{verdict}, \mathit{toBreak})$ {breakAttempt *is defined below*
reply $(\texttt{BreakAccProp}, \mathit{successful}, \mathit{leakage})$

recv GetVerdict **from** I/O **to** $(\mathit{pid}_j, \mathit{sid}, \texttt{judge})$: {*The environment can query the verdicts of local and public judges*
reply $(\texttt{GetVerdict}, \mathsf{verdicts}[\mathit{pid}_j])$

recv $(\texttt{GetJudicialReport}, \mathit{msg})$ **from** I/O **to** $(\mathit{pid}_j, \mathit{sid}, \texttt{judge})$: {*The environment may query for local or public judicial reports*
send $(\texttt{GetJudicialReport}, \mathit{msg}, \mathsf{internalState})$ **to** $(\mathit{pid}_j, \mathit{sid}, \mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams})$ {*Forward judicial report request to* $\mathcal{F}_{\mathrm{judgeParams}}$
wait for $(\texttt{GetJudicialReport}, \mathit{report})$
reply $(\texttt{GetJudicialReport}, \mathit{report})$

Helperfunctions:

procedure $\texttt{breakAttempt}(\mathit{verdict}, \mathit{toBreak})$: {*Process break attempt*
Check for all non-ε verdicts in $\mathit{verdict}$, i.e., $\forall \mathit{pid}_j$ **s.t.** $\mathit{verdict}[\mathit{pid}_j] \neq \varepsilon$:
1. it holds that $\mathit{verdict}[\mathit{pid}_j]$ is a positive boolean expression built from propositions of the form $\mathsf{dis}((\mathit{pid}, \mathit{sid}, \mathit{role}))$,
2. it holds that $\mathsf{eval}(\mathit{verdict}[\mathit{pid}_j]) = \texttt{true}$,[b]

Check that $\forall(\mathit{prop}, \mathit{pid}_j) \in \mathit{toBreak}$:

3. $\mathit{verdict}[\mathit{pid}_j] \neq \varepsilon$,
4. $\mathsf{brokenAssumptions}[\mathit{prop}, \mathit{pid}_j] = \texttt{true}$, if $\mathit{prop} \in \mathsf{Sec}^{\mathsf{assumption}} \wedge \mathit{pid}_j \in \mathcal{S}ec^{\mathsf{assumption}}$.

if any of the above check fails**:**
$\mathbf{return}(\texttt{false}, \varepsilon)$
send $(\texttt{BreakAccProp}, \mathit{verdict}, \mathit{toBreak}, \mathsf{internalState})$ **to** $(_, _, \mathcal{F}_{\mathrm{judgeParams}} : \texttt{judgeParams})$ {$\mathcal{F}_{\mathrm{judgeParams}}$ *can impose further conditions, e.g., on verdicts or whether it is allowed to violate breakable security properties.*
wait for $(\texttt{BreakAccProp}, \mathit{successful}, \mathit{leakage})$
if $\mathit{successful}$**:**
for all $\forall \mathit{pid}_j$ **s.t.** $\mathit{verdict}[\mathit{pid}_j] \neq \varepsilon$ **do:**
$\mathsf{verdicts}[\mathit{pid}_j] \leftarrow \mathit{verdict}[\mathit{pid}_j]$ {*Record accepted local and public verdict*
for all $(\mathit{prop}, \mathit{pid}_j) \in \mathit{toBreak}$ **do:**
$\mathsf{brokenProps}[\mathit{prop}, \mathit{pid}_j] \leftarrow \texttt{true}$
$\mathbf{return}(\mathit{successful}, \mathit{leakage})$

[a] NET denotes message from the network adversary. I/O denotes messages from the environment.
[b] eval evaluates the bolean expression, where $\mathsf{dis}(\mathit{pid}, \mathit{sid}, \mathit{role})$ evaluates to true if $(\mathit{pid}, \mathit{sid}, \mathit{role}) \in \mathsf{CorruptionSet}$ or $(\mathit{pid}, \mathit{sid}, \mathit{role}) \in \mathsf{corruptedIntParties}$. CorruptionSet is a predefined variable of iUC that contains all corrupted main parties of this functionality. We set $\mathsf{eval}(\varepsilon) :=$ true.

Fig. 8: Judge code added by the transformation $\mathcal{T}_1(\mathcal{F})$ to an ideal functionality $\mathcal{F}$.

(which are otherwise hidden to them). Without the supervisor, it would be impossible to evaluate whether a verdict is fair in the case that an internal protocol participant is blamed for misbehavior.

Step 2. The second step of the AUC transformation specifies the effects of a broken property. As the exact implications in terms of the behavior of $\mathcal{F}$ strongly depend on the individual accountability properties, AUC abstractly guidelines protocol designers for the adaptions in $\mathcal{F}$: Modeling the effects of a broken accountability property p, generally entails introducing (one or more) conditional clauses of the form "**if** (p, id) *is not marked as broken* **then** *<original behavior>* **else** *<new behavior>*". As the name suggests, *<original behavior>* denotes the original unchanged behavior of the functionality $\mathcal{F}$, i.e., the code that enforces p. The code *<new behavior>* then defines what "breaking p" actually means, typically by giving more power to the adversary (cf. [33]).

D. Additional Details on the Ideal BB Functionalities

In this section, we provide further details regarding the two BB functionalities we introduce in Section III.

D.1 Full Details on $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$: Here, we provide the full formal specification of $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ in Figure 10 to 12. The explanation regarding $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$'s inner workings can be found in Section III-A-

D.2 Full Details on $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$: In this section, we explain further details of the customizable ideal BB functionality $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$. As $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ and $\mathcal{F}_{\mathrm{ledger}}$ [35] share many similarities (cf. also Appendix E), also their descriptions are similar. Figure 15 provides an overview over $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ structure.

Description of $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$: Our functionality $\mathcal{F}_{\mathrm{ledger}}$ is defined in the iUC framework [12] – a recently proposed an easy-to-use framework for universal composability similar in spirit to Canetti's UC model [13]. Accountability features are

```
Additional code for the supervisor role:
    recv (BreakAssumption, toBreak) from NET to (_, _, supervisor) s.t. toBreak ⊆ Sec^assumption × ids_assumption:   {A may break these assumptions
        for all (prop, id) ∈ toBreak do:
            brokenAssumptions[prop, id] ← true                                   {Record broken assumptions
            if prop ∉ Sec^acc ∨ id ∉ pids_judge:
                brokenProps[prop, id] ← true
                                    {Record property as broken if not additionaly secured via accountability
        send (BreakAssumption, toBreak, internalState)
         to (pid, sid, F_judgeParams : judgeParams)
                                                                 {F_judgeParams provides leakage
        wait for (BreakAssumption, leakage)
        reply (BreakAssumption, leakage)
    recv (corruptInt, (pid, sid, role)) from NET to (_, _, supervisor)
       s.t. role ∉ Roles_F ∪ {judge, supervisor}:                {A is allowed to corrupt internal protocol parties
       corruptedIntParties.add((pid, sid, role))
       reply (corruptInt, ack)
    recv (IsAssumptionBroken?, prop, id) from I/O
       to (_, _, supervisor) s.t. id ∈ ids_assumption:           {The environment may ask whether properties are broken
        if prop ∈ Sec^assumption:
            reply (IsAssumptionBroken?, brokenAssumptions[prop, id])
        else:
            reply (IsAssumptionBroken?, ⊥)
    recv (corruptInt?, (pid, sid, role)) from I/O to (_, _, supervisor)
       s.t. role ∉ Roles_F ∪ {judge, supervisor}:
                                    {The environment may ask for the corruption status of internal parties
        if (pid, sid, role) ∈ corruptedIntParties:
            reply (corruptInt, true)
        else:
            reply (corruptInt, false)
```

Fig. 9: Supervisor code added by the transformation $\mathcal{T}_1(\mathcal{F})$ to an ideal functionality $\mathcal{F}$.

captured within AUC which is not explicitly explained here (cf. Section II). We present the formal specification of $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ in Figure 13 and 14 (adaptions/changes due to AUC are highlighted in blue.)

The functionality $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ is a single machine containing the logic for handling incoming read and write requests. In addition to this main machine, $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ uses several subroutine machines that serve as parameters which must be customized by a protocol designer to match the exact security guarantees provided by $\mathcal{F}_{\text{cBB}}^{\text{acc}}$.

Intuitively, $\mathcal{F}_{\text{cBB}}^{\text{acc}}$'s subroutines have the following purposes: $\mathcal{F}_{\text{write}}$ handles write requests and, e. g.ensures the validity of submitted items, $\mathcal{F}_{\text{read}}$ processes read requests and $\mathcal{F}_{\text{update}}$ handles updates to $\mathcal{F}_{\text{cBB}}^{\text{acc}}$'s global state, and $\mathcal{F}_{\text{updRnd}}$ controls updates to $\mathcal{F}_{\text{cBB}}^{\text{acc}}$'s in-built clock

In what follows, we explain $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ from the point of view of an honest party – the process of submitting new items to the BB, adding those to the BB's state, and then read from that state.

Submitting Messages/Items. A higher-level protocol or the environment $\mathcal{E}$ can instruct an honest party pid to submit a message msg. Upon receiving such a request, $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ forwards the request to the subroutine $\mathcal{F}_{\text{write}}$, which then decides whether the message is valid. $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ expects $\mathcal{F}_{\text{write}}$ to return a boolean value indicating whether the message is accepted. If the transaction msg is accepted, $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ adds msg together with the submitting party pid and a time stamp (see below) to a buffer list writeQueue that keeps track of write requests from honest parties which have not yet been added to the BB's state. The acceptance result including the full item is then leaked to the adversary.

As mentioned above, the specification of $\mathcal{F}_{\text{write}}$ is a parameter that is left to the protocol designer to customize. This allows for customizing what the format of a "valid message" looks like.

Adding message to the BB's global state. $\mathcal{F}_{\text{cBB}}^{\text{acc}}$'s msglist, contains the BB's state. The items in msglist are totally ordered and form the basis for honest party's read requests. Furthermore, items are stored together with some additional information: the ID of the party which submitted the transaction and two-time stamps indicating when the transaction was submitted, and when it was added to the BB's state. Similar to ideal functionalities for blockchains, the state of $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ is determined and updated by the adversary, subject to restrictions that ensure expected security properties.

More specifically, at any point in time, the adversary on the network can send an update request to $\mathcal{F}_{\text{cBB}}^{\text{acc}}$. This request, which contains an arbitrary bit string, is then forwarded (together with a copy of the internal state of $\mathcal{F}_{\text{cBB}}^{\text{acc}}$)

Description of the protocol $\mathcal{F}^{\text{acc}}_{\text{BB}} = (\texttt{client}, \texttt{judge}, \texttt{supervisor})$:

Participating roles: $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Corruption model: *dynamic corruption*

Description of M_{sBB}:

Implemented role(s): $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Subroutines: $\mathcal{F}^{\text{BB}}_{\text{judgeParams}} : \texttt{judgeParams}$
Internal state:

- $\mathsf{msglist} \subset \mathbb{N} \times (\{0,1\}^*)^2, \mathsf{msglist} = \emptyset$. {*(Totally ordered) sequence of recorded messages that is considered as stable/immutable of the form* (id, msg, pid).
- $\mathsf{writeQueue} \subset \mathbb{N} \times (\{0,1\}^*), \mathsf{writeQueue} = \emptyset$ {*The list of so far not ordered, honest, incoming messages. Format* $(tmpCtr, msg)$.
- $\mathsf{readQueue} \subset (\{0,1\}^*)^3 \times \mathbb{N} \times \{\texttt{true}, \texttt{false}\} \times \mathbb{N}, \mathsf{readQueue} = \emptyset$, {*The queue of read responses that need to be delivered* $(pid, responseId, delivered, ptr)$
- $\mathsf{readCtr} \in \mathbb{N}, \mathsf{readCtr} = 0$, {$\mathsf{readCtr}$ *is temporary ID for transactions in the* $\mathsf{readQueue}$.
- $\mathsf{writeCtr} \in \mathbb{N}, \mathsf{writeCtr} = 0$, {$\mathsf{writeCtr}$ *are temporary IDs for transactions in the* $\mathsf{writeQueue}$.
- $\mathsf{brokenProps} : \{\texttt{consistency}\} \times \{\texttt{public}\} \to \{\texttt{true}, \texttt{false}\}$ {*Stores broken accountability* properties *for the public judge, initially* $\texttt{false}$ $\forall$*entries*
- $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$ {*Verdicts per* $p \in \mathsf{pids}_{\mathsf{judge}}$, *initially* ε
- $\mathsf{corruptedIntParties} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{a} \cup \{\texttt{judge}, \texttt{supervisor}\})$, initially $\emptyset$ {*The set of corrupted internal parties* $(pid, sid, role)$

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

Corruption behavior:

- **AllowCorruption**$(pid, sid, role)$:
  Do not allow corruption of $(pid, sid, \texttt{supervisor})$.
  if $role = \texttt{judge}$**:**
  send $(\mathsf{Corrupt}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$ {$\mathcal{F}_{\text{judgeParams}}$ *decides whether judges can be corrupted*
  wait for b
  return b
- **DetermineCorrStatus**[b]$(pid, sid, role)$:
  if $role = \texttt{judge}$**:** {$\mathcal{F}_{\text{judgeParams}}$ *may determine a judge's corruption status*
  send $(\texttt{CorruptionStatus?}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$
  wait for b; **return** b
- **AllowAdvMessage**$(pid, sid, role, pid_{\texttt{receiver}}, sid_{\texttt{receiver}}, role_{\texttt{receiver}}, m)$:
  $\mathcal{A}$ is not allowed to call subroutines on behalf of a corrupted party.

[a] $\mathsf{Roles}_{\mathcal{F}}$ is the set of (main) roles provided by $\mathcal{F}$ to the environment. Here, $\mathsf{Roles}_{\mathcal{F}} = \{\texttt{client}\}$
[b] **DetermineCorrStatus** allows protocol designers to specify whether an entity that is currently not directly controlled by the attacker should nevertheless consider itself to be corrupted. E.g., a local judge will typically consider itself to be corrupted already if its corresponding party is corrupted.

Fig. 10: The basic ideal individually accountable bulletin board functionality $\mathcal{F}^{\text{acc}}_{\text{BB}}$ (Part 1).

to the subroutine $\mathcal{F}_{\text{update}}$. The exact format of the bit string provided by the adversary is not a priori fixed and can be freely interpreted by $\mathcal{F}_{\text{update}}$. This subroutine then computes and returns to $\mathcal{F}_{\text{update}}$ an extension of the current BB state and an update to the list writeQueue of submitted transactions that specify transactions that should be removed (as those have now become part of the BB's state).

Upon receiving the response from $\mathcal{F}_{\text{update}}$, $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ ensures that appending the proposed extension to msglist still results in an ordered list of items. If this is the case, $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ applies the changes to both lists. In any case, $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ leaks all information as well as a boolean indicating whether any changes have been applied to the adversary.

Reading the BB's state. A higher-level protocol can instruct a party of $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ to read from the BB's state. The read request is stored and asynchronously handled. The adversary is supposed to provide input to $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ to determine the output for the read request. $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ forwards the input to $\mathcal{F}_{\text{read}}$. $\mathcal{F}_{\text{read}}$ uses the $\mathcal{A}$'s input to generate the read request's final output. The exact format of $\mathcal{A}$'s input is not a priori fixed and can be freely interpreted by $\mathcal{F}_{\text{read}}$. Finally, the resulting output is forwarded by $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ to the higher-level protocol.

D.3 Covering Further BB Standard Security Properties With $\mathcal{F}^{\text{acc}}_{\text{cBB}}$: Here, we briefly discuss how one can capture other standard BB security properties mentioned in Section I with $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ which we have not discussed in Section III-B.

Description of $M_{\mathtt{sBB}}$ (continued):

Main:

```
recv (Write, msg) from I/O:                                        {Write request from a honest identity
    writeCtr ← writeCtr + 1
    writeQueue.add(writeCtr, msg)^a                  {Record message, identity and its state for "consensus".
    send (Write, writeCtr, msg) to NET                                                  {Leak full msg.
recv Read from I/O:                                                 {Read request from an honest identity
    readCtr ← readCtr + 1
    readQueue.add((pid_call, sid_call, role_call), readCtr, false, 0)       {Store request for later processing
    send (Read, readCtr) to NET                                                      {Leak full details
recv (DeliverRead, readCtr, (sugPtr, sugOutput)) from NET s.t. ((pid, sid, role), readCtr, false, 0) ∈ readQueue:
        {sugPtr points to the prefix to be delivered. sugOutput contains the message to be delivered (if consistency is broken)
    if brokenProps[consistency, public] = false:       {F^acc_BB provides consistency in the absence of a verdict
        if ∃((pid, sid, role), _, true, prt) ∈ readQueue, s.t. prt > sugPtr:
            send nack to NET                                                  {Delivery request of A was denied
        else:
            readQueue.remove((pid, sid, role), readCtr, false, 0)                         {Clean up readQueue.
            readQueue.add((pid, sid, role), readCtr, true, sugPtr)
            send (Read, msglist(sugPtr)^b) to (pid, sid, role)
    else:                                {A fully defines the output if accountability w.r.t. consistency is broken
        readQueue.remove((pid, sid, role), readCtr, false, 0)                             {Clean up readQueue.
        readQueue.add((pid, sid, role), readCtr, msg, true, 0)
        send (Read, sugOutput) to (pid, sid, role)
recv (Update, appendToMsglist, updRequestQueue) from NET s.t. updRequestQueue ⊂ writeQueue∧
        all entries from updRequestQueue appear in appendToMsglist:  {Update or maintain request triggered by the adversary.
    msglist.append(appendToMsglist)     {append adds each entry from appendToMsglist to msglist including consecutive IDs
    for all item ∈ updRequestQueue do:                               {Remove "consumed" elements from writeQueue
        writeQueue.remove(item)
    reply (Update, check, msglist)                            {Inform A if update was successful and leake data.
Include static code provided by AUC [33] here (see also Appendix C).^c
```

[a] writeQueue.add(_) equals writeQueue ← writeQueue ∪ {_}.

[b] For $n \in \mathbb{N}$, we define $\mathsf{msglist}(n), n \in \mathbb{N} = \{(id, msg) \mid (id, msg, _) \in \mathsf{msglist} \wedge id \leq n\}$.

[c] Technically, there need to be several minor adaptions in the AUC transformation However, the changes follow immediately from [33].

Fig. 11: The basic ideal individually accountable bulletin board functionality $\mathcal{F}_{\mathrm{BB}}^{\mathrm{acc}}$ (Part 2).

Certified Publishing: We note that this property is rather specific to the BB of Heather and Lundin [37] where the BB add all items to it sequentially and clients exactly determine the position of their item on the board via a timestamp (timestamps need to be increasing for each item). Heather and Lundin define certified as follows "A [...] BB has certified publishing if whenever a reader retrieves the contents of the board, either he can detect corruption of the board, or he will have proof, for each message on the board:

1. of who wrote the message;
2. that the writer intended the message to be published with the stated timestamp and at this point in the board's sequence of messages."

To capture that a read request of an item from the BB's state also includes the writing party, we need to define $\mathcal{F}_{\mathrm{read}}$ such that it also outputs the party that submitted an item to the BB (as stored in msglist). The second part requires that *(i)* $\mathcal{F}_{\mathrm{write}}$ enforces that the data format for BB items includes a timestamp and *(ii)* $\mathcal{F}_{\mathrm{update}}$ ensures that the timestamp increases when the next item is added to the BB.

Message Validity: As also discussed in Section III-B, $\mathcal{F}_{\mathrm{write}}$ is intended to ensure message validity. However, for items added by $\mathcal{A}$ it is also necessary to ensure correct message formats in $\mathcal{F}_{\mathrm{update}}$.

Receipt Consistency: Culnane and Schneider [24] and Kiayias et al. [46] require that "[a]ny item that has a valid receipt must appear on the [B]B." According to their constructions, a subset of the BB peers receives the votes and another BB component is responsible for finally storing the item and providing parties read-access to the item. To model such a requirement in $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$ one would define a sub-call of the Read interface to retrieve the desired Receipt. $\mathcal{F}_{\mathrm{updRnd}}$ would then ensure that the item enters $\mathcal{F}_{\mathrm{cBB}}^{\mathrm{acc}}$'s state according to some defined network delay, let us say δ. More specifically, $\mathcal{F}_{\mathrm{updRnd}}$ would deny time updates if there was a receipt handed δ ago but the corresponding item

Description of $\mathcal{F}^{\text{BB}}_{\text{judgeParams}} = (\texttt{judgeParams})$:

Participating roles: $\{\texttt{judgeParams}\}$
Corruption model: *incorruptible*

Description of $M^{\text{BB}}_{\texttt{judgeParams}}$:

Implemented role(s): $\{\texttt{judgeParams}\}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:

```
recv (BreakAccProp, verdict, toBreak, internalState) from I/O:
  if verdict[public] ensures individual accountability:      {Ensure public individual accountability
    if toBreak = {consistency, public}:                       {Handle violation of accountability w.r.t. consistency
      reply (BreakAccProp, true, ε)
  else:
    reply (BreakAssumption, false, ε)
recv (GetJudicialReport, msg, internalState) from I/O:       {Generate judicial report
  if verdicts ≠ ε:
    reply (GetJudicialReport, ε)                              {Cannot provide a judicial report in this case
  else:
    Extract maxPtr from F^acc_BB's transcript as the highest pointer that was used to answer a read request.
    send responsively GetState to NET    (*)                  {A may define the prefix to be delivered as judicial report
    wait for (GetState, ptr)
    if ptr < maxPtr ∨ ptr > |msglist|:
      Go to (*).
    list ← {(id, msg') | (id, msg', _) ∈ msglist[maxPtr]}
    reply (GetJudicialReport, list)                           {Return state as report
recv (Corrupt, (public, sid, judge), internalState) from I/O: {F^BB_judgeParams declines corruption requests for the public judge
  reply false                                                 {The public judge is incorruptible
recv (CorruptionStatus?(public, sid, judge), internalState) from I/O: {F^BB_judgeParams asks for the public judge's corruption status
  reply false                                                 {The public judge is incorruptible
```

Fig. 12: The judge parameter functionality $\mathcal{F}^{\text{BB}}_{\text{judgeParams}}$.

is not in $\mathcal{F}^{\text{acc}}_{\text{cBB}}$'s state yet. Also, $\mathcal{F}_{\text{read}}$ would ensure that the item is included in read responses of honest parties (also after δ).
Several BB constructions do not include receipts out of the box. In these cases, one could interpret that reading the client's item from the BB is a sufficient receipt to confirm that the item is actually part of the BB's state. In this case, receipt consistency collapses to traditional consistency.
Finally, there are also other constructions regarding receipts common. In, e. g., Helios [1], clients/voters do write their ballots directly to the BB. They send their ballots to the Helios election server. The server provides the voter a receipt when he accepts the ballot. The server then internally processes the ballot and writes it to the BB. In such a case, receipt consistency needs to be ensured in the higher-level (e-voting) protocol in combination with the accountable BB.

D.4 The $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ Instantiation to Cover $\mathcal{F}^{\text{acc}}_{\text{BB}}$: As a sanity check, that $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ is indeed a generalization of $\mathcal{F}^{\text{acc}}_{\text{BB}}$, we show that $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ realizes $\mathcal{F}^{\text{acc}}_{\text{BB}}$, resp. vice versa. In what follows, we call the instantiation that covers $\mathcal{F}^{\text{acc}}_{\text{BB}}$ $\mathcal{F}^{\text{acc}}_{\text{cBB,BB}}$. In $\mathcal{F}^{\text{acc}}_{\text{cBB,BB}}$, $\mathcal{F}_{\text{write,BB}}$ does not perform additional checks and allows all write requests to enter the writeQueue. $\mathcal{F}_{\text{read,BB}}$ records which state was provided to which party and as long as `consistency` hods, it enforces checks that the state provided to parties is equal or an extension of their previously received state. If public `consistency` is broken, it forwards $\mathcal{A}$ input as response to the parties request to $\mathcal{F}^{\text{acc}}_{\text{cBB}}$. As $\mathcal{F}^{\text{acc}}_{\text{BB}}$ does not include a clock, we do not impose limitations on clock updates in $\mathcal{F}_{\text{updRnd,BB}}$. $\mathcal{F}_{\text{update,BB}}$ also mimics $\mathcal{F}^{\text{acc}}_{\text{BB}}$'s behavior during update and accepts updates that match the $\mathcal{F}^{\text{acc}}_{\text{BB}}$'s update requirements. To handle accountability properties, $\mathcal{F}_{\text{read,BB}}$ simply uses $\mathcal{F}^{\text{acc}}_{\text{BB}}$'s AUC subroutine $\mathcal{F}^{\text{BB}}_{\text{judgeParams}}$ (cf. Figure 12).

We provide the specification of the subroutine of $\mathcal{F}^{\text{acc}}_{\text{cBB,BB}}$ in Figures 16 to 19. $\mathcal{F}^{\text{BB}}_{\text{judgeParams}}$ was already introduced in Figure 12.

To formally show that $\mathcal{F}^{\text{acc}}_{\text{cBB,BB}}$ realizes $\mathcal{F}^{\text{acc}}_{\text{BB}}$ and vice versa, we need to enhance $\mathcal{F}^{\text{acc}}_{\text{BB}}$ with a clock. Therefore,

Description of the protocol $\mathcal{F}^{\text{acc}}_{\text{cBB}} = (\texttt{client}, \texttt{judge}, \texttt{supervisor})$:

Participating roles: $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Corruption model: *static* or *dynamic corruption*
Protocol parameters:

- $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$ — {*Accountability propertie*
- $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$ — {*Assumption-based security properties*
- $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$ — {*set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)*
- $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$ — {*set of entities/IDs where properties are ensured via assumptions*

Description of M_{BB}:

Implemented role(s): $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Subroutines: $\mathcal{F}_{\text{write}} : \texttt{validate}, \mathcal{F}_{\text{update}} : \texttt{update}, \mathcal{F}_{\text{read}} : \texttt{read}, \mathcal{F}_{\text{updRnd}} : \texttt{updRnd}, \mathcal{F}^{\text{BB}}_{\text{judgeParams}} : \texttt{judgeParams}$
Internal state:

- $\mathsf{round} \in \mathbb{N}_{\geq 0}, \mathsf{round} = 0$ — {*Current (network) round in the protocol execution.*
- $\mathsf{msglist} \subset \mathbb{N} \times \mathbb{N} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, $\mathsf{msglist} = \emptyset$. — {*(Totally ordered) sequence of recorded messages that is considered as stable/immutable of the form $(id, commitRound, msg, submitRound, pid)$.*
- $\mathsf{writeQueue} \subset \mathbb{N} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, $\mathsf{writeQueue} = \emptyset$ — {*The list of so far not ordered, honest, incoming "transactions". Format $(tmpCtr, tx, submittingRound, submittingParty)$.*
- $\mathsf{readQueue} \subset \{0,1\}^* \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^*$, $\mathsf{readQueue} = \emptyset$, — {*The queue of read responses that need to be delivered $(pid, responseId, round, msg)$*
- $\mathsf{readCtr} \in \mathbb{N}, \mathsf{readCtr} = 0$, — {readCtr *is temporary ID for transactions in the* readQueue.
- $\mathsf{writeCtr} \in \mathbb{N}, \mathsf{writeCtr} = 0$, — {writeCtr *are temporary IDs for transactions in the* writeQueue.
- $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\texttt{true}, \texttt{false}\}$ — {*Stores broken security* properties *per judge/id, initially* `false` $\forall$*entries*
- $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$ — {*Verdicts per $p \in \mathsf{pids}_{\mathsf{judge}}$, initially ε*
- $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\texttt{true}, \texttt{false}\}$ — {*Stores broken security* assumptions *per id, initially* `false` $\forall$*entries*
- $\mathsf{corruptedIntParties} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{a} \cup \{\texttt{judge}, \texttt{supervisor}\})$, initially $\emptyset$ — {*The set of corrupted internal parties $(pid, sid, role)$*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

Corruption behavior:

- **AllowCorruption**$(pid, sid, role)$:
  Do not allow corruption of $(pid, sid, \texttt{supervisor})$.
  if $role = \texttt{judge}$**:**
  send $(\mathsf{Corrupt}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$ — {*$\mathcal{F}_{\text{judgeParams}}$ decides whether judges can be corrupted*
  wait for b
  return b
- **DetermineCorrStatus**[b]$(pid, sid, role)$:
  if $role = \texttt{judge}$**:** — {*$\mathcal{F}_{\text{judgeParams}}$ may determine a judge's corruption status*
  send $(\texttt{CorruptionStatus?}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  to $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$
  wait for b; **return** b
- **AllowAdvMessage**$(pid, sid, role, pid_{\texttt{receiver}}, sid_{\texttt{receiver}}, role_{\texttt{receiver}}, m)$: $\mathcal{A}$ is not allowed to call subroutines on behalf of a corrupted party.

[a] $\mathsf{Roles}_{\mathcal{F}}$ is the set of (main) roles provided by $\mathcal{F}$ to the environment. For example, $\mathsf{Roles}_{\mathcal{F}} = \{\texttt{signer}, \texttt{verifier}\}$ for an ideal signature functionality $\mathcal{F} := \mathcal{F}_{\text{sig}}$.
[b] **DetermineCorrStatus** allows protocol designers to specify whether an entity that is currently not directly controlled by the attacker should nevertheless consider itself to be corrupted. E.g., a local judge will typically consider itself to be corrupted already if its corresponding party is corrupted.

Fig. 13: The ideal individually accountable bulletin board functionality $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ (Part 1).

we put a wrapper $\mathcal{W}_{\text{BB}}$ in front of $\mathcal{F}^{\text{acc}}_{\text{BB}}$ which *(i)* forwards all requests to $\mathcal{F}^{\text{acc}}_{\text{BB}}$ with the exception of `GetCurRound` interface, *(ii)* $\mathcal{W}_{\text{BB}}$ has an internal clock, i. e., it contains a state variable $\mathsf{round} \in \mathbb{N}, \mathsf{round} = 0$, *(iii)* On the call `UpdateRound` via NET, $\mathcal{W}_{\text{BB}}$ increases round by one 1 and replies $(\texttt{Update}, \texttt{true})$. *(iv)* On `GetCurRound` via I/O interface, $\mathcal{W}_{\text{BB}}$ replies $(\texttt{GetCurRound}, \mathsf{round})$.

Description of M_{BB} (continued):

```
Main:
  recv (Write, msg) from I/O:                                              {Write request from a honest identity
    send (Write, msg, internalState) to (pid_cur, sid_cur, F_write : validate)    {Forward request to F_write
    wait for (Write, response) s.t. response ∈ {true, false}
    if response = true:
      writeCtr ← writeCtr + 1
      writeQueue.add(writeCtr, round, pid_cur, msg)^a     {Record message, round, identity and its state for "consensus".
    send (Write, response, msg) to NET                                    {Leak full msg.
  recv (Read, msg) from I/O:                                               {Read request from an honest identity
    readCtr ← readCtr + 1; readQueue.add((pid_call, sid_call, role_call), readCtr, round, msg)  {In case of network read, store request
    send (Read, readCtr, msg) to NET                        {If F_read leakes data, this is forwarded to A.
  recv (DeliverRead, readCtr, suggestedOutput) from NET s.t. ((pid, sid, role), readCtr, r, msg) ∈ readQueue:
        {A tiggers message delivery per message (this may include reordering of messages, non-delivery of messages, and manipulation of
         delivered data - if not enforced by F_updRnd).
    send (Read, msg, suggestedOutput, internalState) to (pid_cur, sid_cur, F_read : read)
    wait for (Read, output)
    if output ≠ ⊥:
      send responsively (Read, readCtr, output) to NET
      wait for ack
      readQueue.remove((pid, sid, role), readCtr, r, msg)                 {Clean up readQueue.
      send (Read, output) to (pid, sid, role)
    else:
      send nack to NET                                                    {Delivery request of A was denied
  recv (Update, msg) from NET:                          {Update or maintain request triggered by the adversary.
    send (Update, msg, internalState) to (ε, sid_cur, F_update : update)
    wait for (Update, msglist, updRequestQueue                  {F_update outputs which data to append to
          s.t msglist ⊂ ℕ × {round} × {0,1}* × ℕ × {0,1}*        msglist and an updated writeQueue.
    max ← max{i|(i, _, _, _, _) ∈ msglist}          {Check that msglist is a totally ordered sequence, extending the existing
    check ← msglist ≠ ∅ ∨ updRequestQueue ≠ ∅        msglist. If msglist = ∅ then max defaults to −1
    for i = max + 1 to max + |msglist| do:
      if ∄_1 (i, _, _, _, _) ∈ msglist:             {Check that there exists exactly one entry for every id i in
        check ← false                                a continous sequence (no gaps)
    if check:                                       {If the update is totally ordered and no new messages were added to
                                                     writeQueue, we accept the update.
      msglist.add(msglist)
      for all item ∈ updRequestQueue do:            {Remove elements "consumed" elements from writeQueue
        writeQueue.remove(item)
    reply (Update, check, msglist)                  {Inform A if update was successful and leake data.
  recv UpdateRound from NET:                        {A triggers round update if current round satisfies rules of F_updRnd.
    send (UpdateRound, internalState) to (pid_cur, sid_cur, F_updRnd : updRnd)
    wait for (UpdateRound, response)
    if response = true:
      round ← round + 1
    reply (UpdateRound, response)
  recv GetCurRound:                                 {A and E are allowed to query the current round.
    reply (GetCurRound, round)
  Include static code from the AUC transformation T_1(·) [33] here.

^a writeQueue.add(_) equals writeQueue ← writeQueue ∪ {_}.
```

Fig. 14: The ideal individually accountable bulletin board functionality $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB}}$ (Part 2).

Lemma 4 ($(\mathcal{W}_{\mathrm{BB}} \mid \mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}})$[13] $\leq \mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB,BB}}$ and vice versa)**.** *Let $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$, $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB,BB}}$, and $\mathcal{W}_{\mathrm{BB}}$ as defined above. Let $\mathcal{F}_{\mathrm{update}}$ be the ppt update subroutine of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$ and $\mathcal{F}_{\mathrm{update,BB}} = \mathcal{F}_{\mathrm{update}}$.[14] Then, we conclude*

$$(\mathcal{W}_{\mathrm{BB}} \mid \mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}) \leq \mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB,BB}} \text{ and}$$
$$\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB,BB}} \leq (\mathcal{W}_{\mathrm{BB}} \mid \mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}).$$

Proof. Note that the overall systems run in ppt if $\mathcal{F}_{\mathrm{update}}/\mathcal{F}_{\mathrm{update,BB}}$ also run in ppt. The result follows directly from the definitions of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{BB}}$, $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB,BB}}$, and $\mathcal{W}_{\mathrm{BB}}$ as the differences between both functionalities are only semantical. □

[13] The notation $(\mathcal{Q} \mid \mathcal{F})$ describes the system consisting of the two protocols where $\mathcal{Q}$ has a direct I/O interface to the environment $\mathcal{E}$ and $\mathcal{Q}$ uses $\mathcal{F}$ as a subroutine.

[14] For common instantiations the update state process for BBs, this is a realistic requirement.

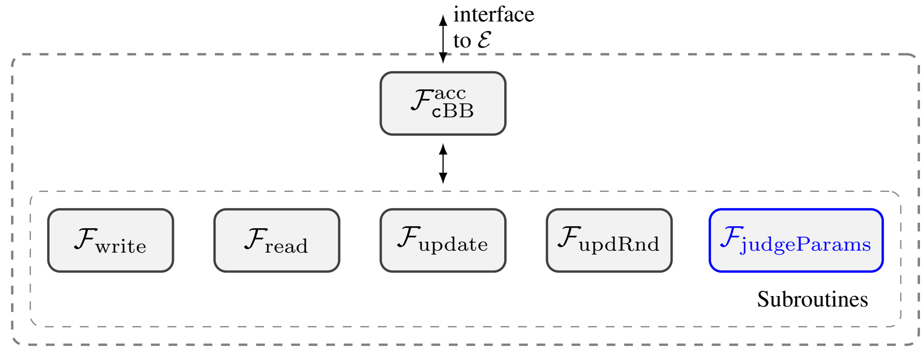


Fig. 15: Overview of $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ and its subroutines. All machines are also connected to $\mathcal{A}$.

Description of the subroutine $\mathcal{F}_{\text{read,BB}} = (\texttt{read})$:

Participating roles: $\{\texttt{read}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{read}}$:

Implemented role(s): $\{\texttt{read}\}$
Internal state:
- $\mathsf{readQueue} \subset \{0,1\}^* \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^* \times \{\texttt{true}, \texttt{false}\} \times \mathbb{N}$,
  $\mathsf{readQueue} = \emptyset$, *{The queue of read responses that need to be delivered* $(pid, responseId, round, msg, delivered, ptr)$

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:

```
recv (Read, msg, suggestedOutput, internalState^a) from I/O s.t. suggestedOutput can be parsed as (sugPtr, sugOutput):
    if brokenProps[consistency, public] = false:        {F^acc_BB provides consistency in the absence of a verdict
        if ∃(pid, _, _, true, prt) ∈ readQueue, s.t. ptr > sugPtr:
            reply (FinishRead, ⊥)
        else:
            readQueue.add(pid, readCtr, r, msg, true, sugPtr)
            send (Read, msglist(sugPtr)^b) to (pid, sid_cur, client)
    else:                                               {A fully defines the output if accountability w.r.t. consistency is broken
        reply (FinishRead, sugOutput)
```

[a] For brevity we use data from $internalState$ with the local variant of the variable name from $\mathcal{F}^{\text{acc}}_{\text{cBB}}$. This includes local variables such as $msglist$, $requestQueue$, $readQueue$, and $round$.
[b] For $n \in \mathbb{N}$, we define $msglist(n), n \in \mathbb{N} = \{(id, msg) \mid (id, _, msg, _, _) \in msglist \wedge id \leq n\}$.

Fig. 16: The read functionality $\mathcal{F}_{\text{read,BB}}$

Description of the subroutine $\mathcal{F}_{\text{updRnd,BB}} = (\texttt{updRnd})$:

Participating roles: $\{\texttt{updRnd}\}$
Corruption model: *incorruptible*
Protocol parameters:
- $\delta \in \mathbb{N}$ *{The upper bound in rounds after which a honest tx should be in the state.*

Description of $M^{\text{BB}}\texttt{updRnd}$:

Implemented role(s): $\{\texttt{updRnd}\}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:

```
recv (UpdateRound, msg, internalState) from I/O:    {See Figure 13 for definition of internalState and the local variables it includes
    reply (UpdateRound, true)
```

Fig. 17: The round update/time update functionality $\mathcal{F}_{\text{updRnd,BB}}$.

Note: To prove Lemma 4, one technically needs to define a simulator $\mathcal{S}$ and show that the environment can not distinguish whether she interacts with the $\mathcal{S}$ and $\mathcal{F}^{\text{acc}}_{\text{cBB,BB}}$ or with $(\mathcal{W}_{\text{BB}} \mid \mathcal{F}^{\text{acc}}_{\text{BB}})$. The simulator to prove Lemma 4 internally simulates the real protocol, i. e., $\mathcal{F}^{\text{acc}}_{\text{cBB,BB}}$ or $(\mathcal{W}_{\text{BB}} \mid \mathcal{F}^{\text{acc}}_{\text{BB}})$, depending on the direction we are currently

Description of the subroutine $\mathcal{F}_{\text{write,BB}} = (\texttt{validate})$:

Participating roles: $\{\texttt{validate}\}$
Corruption model: *incorruptible*

Description of $M^{\text{BB}}_{\texttt{validate}}$:

Implemented role(s): $\{\texttt{validate}\}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:
recv $(\texttt{Write}, msg, internalState)$ **from** $\texttt{I/O}$: {*See Figure 13 for definition of internalState and the local variables it includes*
reply $(\texttt{Write}, \texttt{true})$ {$\mathcal{F}^{\text{acc}}_{\text{BB}}$ *accepts all write requests*

Fig. 18: The write functionality $\mathcal{F}_{\text{write,BB}}$

Description of the subroutine $\mathcal{F}_{\text{update}}\text{BB} = (\texttt{update})$:

Participating roles: $\{\texttt{update}\}$
Corruption model: *incorruptible*

Description of $M_{\texttt{update}}$:

Implemented role(s): $\{\texttt{update}\}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:
recv $(\texttt{Update}, msg, internalState)$ **from** $\texttt{I/O}$:
if $msg = (appendToMsglist, updRequestQueue) \wedge$
$updRequestQueue \subset$ writeQueue$\wedge$ all entries from $updRequestQueue$ appear in $appendToMsglist$:
reply $(\texttt{Update}, msgListAppend, updRequestQueue, msglist \cup msgListAppend)$
{*Return list extension and updated queue, leak full new state*
else:
reply $(\texttt{Update}, \varepsilon, \varepsilon, \varepsilon)$

Fig. 19: The update functionality $\mathcal{F}_{\text{update,BB}}$

considering. The simulator needs to keep – as usual – the corruption status of entities in the simulated real protocol and the ideal protocol synchronous. For corrupted entities, $\mathcal{S}$ forwards inputs and outputs from/to $\mathcal{A}$. $\mathcal{S}$ forwards messages from NET typically to its internal simulation. When the ideal protocol outputs some message on its network interface, $\mathcal{S}$ forwards the message (maybe with some minor mapping, e. g., to match the input format) to its internal simulation of the real protocol: with the following exceptions: iUC subroutines call $\mathcal{A}$ to query for their corruption status on initialization. That means, in case $\mathcal{F}^{\text{acc}}_{\text{BB}} \leq (\mathcal{W}_{\text{BB}} \mid \mathcal{F}^{\text{acc}}_{\text{cBB,BB}})$ $\mathcal{S}$ would block these messages to $\mathcal{A}$. The other way round, $\mathcal{S}$ would generate these corruption requests and send them to $\mathcal{A}$ if the interface that would call the subroutine is called for the first time. Additionally, the simulator has to decline $\mathcal{A}$ `SetCorruptionStatus` requests with `false` and no leakage. Also, the simulator instructs the ideal world to output data to I/O if this occurs in the simulation.

E. $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ Realizes $\mathcal{F}^{\text{acc}}_{\text{cBB}}$

In this section, we formally prove the folk wisdom that distributed ledgers may be good BBs, i. e., a suitable instantiation of the (accountable) general ledger functionality $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ – an accountable variant of the distributed ledger functionality $\mathcal{F}_{\text{ledger}}$ [35] – realizes $\mathcal{F}^{\text{acc}}_{\text{cBB}}$. We use this intermediate result to show that the distributed ledger Fabric* realizes (an instantiation of) $\mathcal{F}^{\text{acc}}_{\text{ledger}}$. iUC's composition theorem then allows us to conclude that Fabric* also realizes (an appropriate instantiation of) $\mathcal{F}^{\text{acc}}_{\text{cBB}}$.

$\mathcal{F}_{\text{ledger}}$. We briefly recall the high-level concepts of $\mathcal{F}_{\text{ledger}}$ by comparing $\mathcal{F}_{\text{ledger}}$ to $\mathcal{F}^{\text{acc}}_{\text{cBB}}$. We refer an interested reader to [35] for a detailed discussion of the possibilities and limitations of $\mathcal{F}_{\text{ledger}}$.

Similar to $\mathcal{F}^{\text{acc}}_{\text{cBB}}$, $\mathcal{F}_{\text{ledger}}$ is meant to impersonate different ledger protocols. Thus, $\mathcal{F}_{\text{ledger}}$ is also customizable via its subroutines. Figure 20 depicts the structure of $\mathcal{F}_{\text{ledger}}$ including its subroutines. As expected, $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ and $\mathcal{F}_{\text{ledger}}$ share many similarities. From I/O perspective, $\mathcal{F}_{\text{ledger}}$ also mainly provides the same read- and write interfaces (modulo naming of the operations) which allow modeling different network models (cf. $\mathcal{F}^{\text{acc}}_{\text{cBB}}$). Both operations are also customizable, the write interface via $\mathcal{F}_{\text{submit}}$ – which essentially has the same meaning as $\mathcal{F}_{\text{write}}$ – and the read

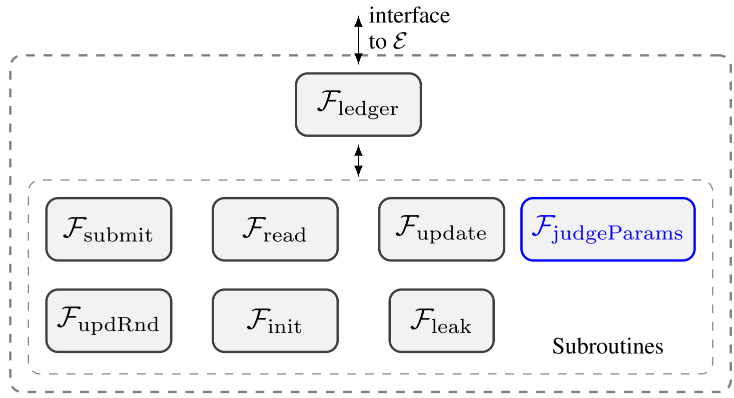


Fig. 20: Overview of $\mathcal{F}_{\text{ledger}}$, resp. $\mathcal{F}^{\text{acc}}_{\text{ledger}}$, and its subroutines [35]. Changes between both functionalities are highlighted in blue. All machines are also connected to $\mathcal{A}$.

interface via $\mathcal{F}_{\text{read}}$. While the core concepts behind both $\mathcal{F}_{\text{read}}$ variants are the same, $\mathcal{F}_{\text{ledger}}$'s $\mathcal{F}_{\text{read}}$ covers more functionality (see below). Also, $\mathcal{F}_{\text{ledger}}$'s $\mathcal{F}_{\text{read}}$ is meant to include smart contracts, i. e., calls to $\mathcal{F}_{\text{read}}$ may process data from the state and compute a response based on the data.[15] One key difference of the `Read` interface between both functionalities is that $\mathcal{F}_{\text{ledger}}$ allows modeling *local reads*. Local reads allow answering read requests immediately without simulating network traffic. $\mathcal{F}_{\text{ledger}}$' s `Update` interface including the subroutines $\mathcal{F}_{\text{update}}$ and its internal clock and $\mathcal{F}_{\text{updRnd}}$ work analogously as in $\mathcal{F}^{\text{acc}}_{\text{cBB}}$.

Another key difference between both functionalities is $\mathcal{F}_{\text{ledger}}$ registering process for clients. $\mathcal{F}_{\text{ledger}}$ automatically registers when parties first participate in the protocol and allows them to leave the protocol via the `DeRegister` interface. With these processes, $\mathcal{F}_{\text{ledger}}$ should be able to capture security properties dependent on the time a party is participating in a ledger protocol. BBs do not aim for such security properties.

As ledgers may strive for privacy properties of stored data/transactions, all (above and upcoming) operations may or may not leak data (as specified by the protocol designer). In contrast, $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ always leaks all details submitted items and current state to $\mathcal{A}$. As $\mathcal{A}$ may have direct access to some parts of the private data, $\mathcal{F}_{\text{ledger}}$ allows reading data from its state via the `CorruptedRead`. In $\mathcal{F}_{\text{read}}$ protocol designers then customize the data $\mathcal{A}$ is allowed to access.

Besides these differences, $\mathcal{F}_{\text{ledger}}$ also uses two additional subroutines which do not match on $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ subroutines: $\mathcal{F}_{\text{leak}}$ and $\mathcal{F}_{\text{init}}$. $\mathcal{F}_{\text{leak}}$ allows customizing data leaked upon the corruption of a party. Such functionality is not necessary for $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ as all data is publicly known anyways. $\mathcal{F}_{\text{init}}$ is meant to handle so-called "Genesis blocks" in distributed ledgers – the externally agreed initial state of a ledger. As one typically does not consider BBs with an initial state, this feature is not relevant for $\mathcal{F}^{\text{acc}}_{\text{cBB}}$.[16]

To derive the accountable ledger functionality $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ – which can additionally handle assumption-based security properties and accountability properties – we apply Step 1 of the AUC transformation to $\mathcal{F}_{\text{ledger}}$ (which solely adds static code to $\mathcal{F}_{\text{ledger}}$). To translate preventive security properties from an instantiation of $\mathcal{F}_{\text{ledger}}$ to its assumption-based or accountable variant, protocol designers need to apply AUC's second transformation.

$\mathcal{F}^{\text{acc}}_{\text{ledger}}$ realizes $\mathcal{F}^{\text{acc}}_{\text{cBB}}$. To formally prove that (accountable) distributed ledgers may indeed serve as suitable (accountable) BB, we prove that a suitable instantiation of $\mathcal{F}^{\text{acc}}_{\text{ledger}}$, called $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$, realizes $\mathcal{F}^{\text{acc}}_{\text{cBB}}$. Before explaining $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$, we remark that – similarly to Lemma 4 – it is necessary to use a wrapper $\mathcal{W}_{\text{BB}}$ in front of $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$ to formally prove the realization statement. $\mathcal{W}_{\text{BB}}$ *(i)* maps $\mathcal{F}_{\text{ledger}}$'s `Submit` interface to $\mathcal{F}^{\text{acc}}_{\text{cBB}}$'s write interface and *(ii)* makes $\mathcal{F}_{\text{ledger}}$'s `DeRegister` interface inaccessible. In $\mathcal{F}_{\text{ledger,BB}}$ $\mathcal{F}_{\text{write}}$ is used as $\mathcal{F}_{\text{submit}}$ with the difference that $\mathcal{F}_{\text{submit}}$ needs to add that the full input message is leaked to $\mathcal{A}$. Similarly, $\mathcal{F}_{\text{ledger,BB}}$ used $\mathcal{F}^{\text{acc}}_{\text{cBB}}$'s $\mathcal{F}_{\text{read}}$ subroutine also as $\mathcal{F}_{\text{read}}$ subroutine. Again, $\mathcal{F}_{\text{ledger,BB}}$'s $\mathcal{F}_{\text{read}}$ needs to ensure that the full output of a read request is leaked to

[15] We remark that we did not restrict $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ here, i. e., $\mathcal{F}^{\text{acc}}_{\text{cBB}}$ can also process "smart contracts". However, one would typically use $\mathcal{F}^{\text{acc}}_{\text{cBB}}$'s $\mathcal{F}_{\text{read}}$ to model filtering of BB content, etc.

[16] We note that one can also use $\mathcal{F}^{\text{acc}}_{\text{cBB}}$'s $\mathcal{F}_{\text{read}}$ to model initial state if desired.

Name	Message Structure	S	R	Description
Proposal	$(\texttt{Propose}, \mathsf{txId}, \mathsf{c}, \mathsf{n_T}, ch, chaincodeId, txPayload, \sigma_\mathsf{T})$	C	P	Call of chaincode
Endorsement	$(\texttt{Endorsed}, p, \mathsf{txId}, \mathsf{n_E}, chaincodeId, tx, resp, readset, writeset, \sigma_\mathsf{E})$	P	C	Vote for proposal
Commitment	$(\texttt{Commit}, c, \mathsf{txId}, proposal, txEnd_1, txEnd_2, \ldots)$	C	O	Request transaction commitment
Block Delivery	$(\texttt{Deliver}, blockNum, h, B, o, \sigma_\mathcal{B})$	O	P	Block distribution
Kafka Delivery	$(\texttt{Deliver}, ch, startOffset, T')$	K	O	Stream segment distribution
Chain Evidence	$(\texttt{Evidence}, blockNum, h, B, o, \sigma_\mathcal{B})$	P	J	Forward accepted blocks

S denotes the intended sending ITM, **R** denotes the intended receiving ITM, commands are not repeated in other message types

Variables: c is a client identity, n_T is a sequence number of c, *txPayload* specifies function call and input, $tx = c, ch, chaincodeId, txPayload$, $\sigma_T = \texttt{Sign}_{sk(c)}(tx)$, $\mathsf{txId} = \mathsf{H}(tx, \sigma_\mathsf{T})$, where H is a collision-resistant hash function, p is a peer identity, n_E is a sequence number of p, $\sigma_E = \texttt{Sign}_{sk(p)}(p, \mathsf{txId}, \mathsf{n_E}, chaincodeId, tx, readset, writeset)$, $proposal$ is a transaction proposal, $txEnd_1, txEnd_2, \ldots$ are endorsements, o is an orderer identity, h is the previous block hash, B is a block body, $\sigma_\mathcal{B} = \texttt{Sign}_{sk(o)}(blockNum, h, B)$ *startOffset* is an offset number, T' is a segment of a Kafka message stream, T' starts at offset *startOffset*, and S is a state.

TABLE II: Important Messages in the Fabric* Model (cf. [31])

$\mathcal{A}$.[17] For $\mathcal{F}_{\text{update}}$, $\mathcal{F}_{\text{ledger}}$ uses the same subroutines as $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$ with the following adaptions: *(i)* All state entries are of the technical type `transaction`. $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$ also to store data of type `meta` in the state – such a concept is not used in $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$. *(ii)* The leakage for both subroutines is the full $\mathsf{msglist}$[18] For $\mathcal{F}_{\text{updRnd}}$, $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ also uses the same $\mathcal{F}_{\text{updRnd}}$ subroutine as $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$ (without providing additional leakage).

We define $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$ "additional" subroutines in Figures 24 and 25. As we do not consider initial state in $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$, $\mathcal{F}_{\text{init,BB}}$ does not include/generate an initial state for $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$ and as all data is already publicly known in the context of $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$, $\mathcal{F}_{\text{leak,BB}}$ does not provide additional leakage to $\mathcal{A}$.

To cover the same accountability properties as $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$, $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$ uses $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$'s $\mathcal{F}_{\text{judgeParams}}$.

Lemma 5 **(**$(\mathcal{W}_{\text{BB}} \mid \mathcal{F}^{\text{acc}}_{\text{ledger,BB}}) \leq \mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$**).** *Let* $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$, $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$, *and* $\mathcal{W}_{\text{BB}}$ *be as defined above. Let all subroutines of* $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$ *such that they are ppt and the overall system* $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$ *is also ppt.*[19] *Then, we conclude*

$$(\mathcal{W}_{\text{BB}} \mid \mathcal{F}^{\text{acc}}_{\text{ledger,BB}}) \leq \mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}.$$

Proof. Follows directly as the $(\mathcal{W}_{\text{BB}} \mid \mathcal{F}^{\text{acc}}_{\text{ledger,BB}})$ behaves exactly like $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$. The differences here are only semantical changes. The note after Lemma 4 also holds for Lemma 5. □

We note that there are also other possibilities to prove that $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ realizes $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$.

F. Full Details on Fabric$^*_{\text{BB}}$

Here, we provide the formal definition of the Fabric$^*_{\text{BB}}$ protocol $\mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}} = (\mathcal{P}^{\mathsf{FAB}^*}_{\texttt{client}} : \texttt{client} \mid \mathcal{P}^{\mathsf{FAB}^*}_{\texttt{peer}} : \texttt{peer}, \mathcal{P}^{\mathsf{FAB}^*}_{\texttt{orderer}} : \texttt{orderer}, \mathcal{F}_\mathcal{K} : \texttt{consens}, \mathcal{F}_{\text{init}} : \texttt{init}, \mathcal{F}_{\text{judge}} : \texttt{judge}, \mathcal{F}_{\text{sv}} : \texttt{supervisor}, \mathcal{F}_{\text{cert}} : (\texttt{signer}, \texttt{verifier}), \mathcal{F}_{\text{ro}} : \texttt{randomOracle})$. In Figures 26 to 40, we provide the formal specification of $\mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}}$ in iUC. We refer to Graf et al. [31] for a detailed description of the Fabric* model that mainly matches the Fabric$^*_{\text{BB}}$ model here. Differences between Graf et al.'s model and our model here are explained in Section IV. In Table II, we also provide an overview of the different message types in the Fabric/Fabric*/Fabric$^*_{\text{BB}}$ protocol.

F.1 Details on Fabric$^*_{\text{BB}}$ realizes $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$: In this section, we provide additional specifications and details regarding Theorem 2. In particular, we provide the missing formal definitions of the subroutine of $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ in Figures 41 to 45.

[17] We note that $\mathcal{F}^{\text{acc}}_{\text{ledger,BB}}$'s local read function is never used in this context, thus we specify the read functionality to contain

```
recv (InitRead, msg, internalState) from I/O:
    reply (InitRead, false, ε))
```

[18] Technically, the output needs to adapted and the type `transaction` needs to be fully removed.

[19] For common instantiations of $\mathcal{F}^{\text{acc}}_{\texttt{c}\text{BB}}$ this is a realistic assumption.

Description of the protocol $\mathcal{F}^{\text{acc}}_{\text{ledger}} = (\texttt{client}, \texttt{judge}, \texttt{supervisor})$:

Participating roles: $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Corruption model: *dynamic corruption*
Protocol parameters:

- $\mathsf{Sec}^{\mathsf{acc}} \subset \{0,1\}^*$ — {*Accountability properties*
- $\mathsf{Sec}^{\mathsf{assumption}} \subset \{0,1\}^*$ — {*Assumption-based security properties*
- $\mathsf{pids}_{\mathsf{judge}} \subset \{0,1\}^*$ — {*set of judge entities/(P)IDs in the protocol (which are often directly related to some protocol participants)*
- $\mathsf{ids}_{\mathsf{assumption}} \subset \{0,1\}^*$ — {*set of entities/IDs where properties are ensured via assumptions*

Description of $M_{\texttt{client}}$:

Implemented role(s): $\{\texttt{client}, \texttt{judge}, \texttt{supervisor}\}$
Subroutines:

$\mathcal{F}_{\text{submit}} : \texttt{submit}, \mathcal{F}_{\text{update}} : \texttt{update}, \mathcal{F}_{\text{read}} : \texttt{read}, \mathcal{F}_{\text{updRnd}} : \texttt{updRnd}, \mathcal{F}_{\text{init}} : \texttt{init}$,
$\mathcal{F}_{\text{leak}} : \texttt{leak}, \mathcal{F}^{\text{BB}}_{\text{judgeParams}} : \texttt{judgeParams}$

Internal state:

- $\mathsf{identities} \subset \{0,1\}^* \times \mathbb{N}$, $\mathsf{identities} = \emptyset$ — {*The set of participants and the round when they occurred first.*
- $\mathsf{round} \in \mathbb{N}_{>0}$, $\mathsf{round} = 0$ — {*Current (network) round in the protocol execution.*
- $\mathsf{msglist} \subset \bar{\mathbb{N}} \times \mathbb{N} \times \{\texttt{tx}, \texttt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, $\mathsf{msglist} = \emptyset$. — {*(Totally ordered) sequence of recorded messages that is considered as stable/immutable of the form $(id, commitRound, type, msg, submitRound, pid)$. If $type = \texttt{meta}$, $pid = submitRound = \bot$.*
- $\mathsf{requestQueue} \subset \mathbb{N} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, $\mathsf{requestQueue} = \emptyset$ — {*The list of so far not ordered, honest, incoming "transactions". Format $(tmpCtr, tx, submittingRound, submittingParty)$.*
- $\mathsf{readQueue} \subset (\{0,1\}^*)^3 \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^*$, $\mathsf{readQueue} = \emptyset$, — {*The queue of read responses that need to be delivered $((pid, sid, role), responseId, round, msg)$*
- $\mathsf{readCtr} \in \mathbb{N}$, $\mathsf{readCtr} = 0$, — {$\mathsf{readCtr}$ *is temporary ID for transactions in the* $\mathsf{readQueue}$.
- $\mathsf{reqCtr} \in \mathbb{N}$, $\mathsf{reqCtr} = 0$, — {reqCtr *are temporary IDs for transactions in the* $\mathsf{requestQueue}$.
- $\mathsf{brokenProps} : (\mathsf{Sec}^{\mathsf{assumption}} \cup \mathsf{Sec}^{\mathsf{acc}}) \times (\mathsf{pids}_{\mathsf{judge}} \cup \mathsf{ids}_{\mathsf{assumption}}) \to \{\texttt{true}, \texttt{false}\}$ — {*Stores broken security* properties *per judge/id, initially* $\texttt{false}$ $\forall$*entries*
- $\mathsf{verdicts} : \mathsf{pids}_{\mathsf{judge}} \to \{0,1\}^*$ — {*Verdicts per* $p \in \mathsf{pids}_{\mathsf{judge}}$, *initially* ε
- $\mathsf{brokenAssumptions} : \mathsf{Sec}^{\mathsf{assumption}} \times \mathsf{ids}_{\mathsf{assumption}} \to \{\texttt{true}, \texttt{false}\}$ — {*Stores broken security* assumptions *per id, initially* $\texttt{false}$ $\forall$*entries*
- $\mathsf{corruptedIntParties} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \setminus (\mathsf{Roles}_{\mathcal{F}}{}^{a} \cup \{\texttt{judge}, \texttt{supervisor}\})$, initially $\emptyset$ — {*The set of corrupted internal parties $(pid, sid, role)$*

In the following, we pass through the complete internal state of $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ to its subroutines. Thus, we use the variable $\mathsf{internalState}$ as follows: $\mathsf{internalState} \leftarrow (\mathsf{identities}, \mathsf{round}, \mathsf{msglist}, \mathsf{requestQueue}, \mathsf{readQueue}, \delta, \mathsf{CorruptionSet}, \mathsf{transcript})$

We often use the $\mathsf{CorruptionSet}$ as specified in [12]. We often write $pid \in \mathsf{CorruptionSet}$ instead of $(pid, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}) \in \mathsf{CorruptionSet}$ for brevity.

CheckID$(pid, sid, role)$**:** Accept all messages with the same sid.

Corruption behavior:

- **LeakedData**$(pid, sid, role)$:
  if $\exists (pid, registrationRound) \in \mathsf{identities}, registrationRound \in \mathbb{N}$**:**
  $\quad \mathsf{identities}.\texttt{remove}(pid, registrationRound)$
  send $(\mathsf{corrupt}, pid, sid, \mathsf{internalState})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{leak}} : \texttt{leak})$
  wait for $(\mathsf{corrupt}, leakage)$
  return$(leakage)$ — {*Depending on the desired properties of* $\mathcal{F}^{\text{acc}}_{\text{ledger}}$, *output after corruption needs to be specified*
- **AllowCorruption**$(pid, sid, role)$:
  Do not allow corruption of $(pid, sid, \texttt{supervisor})$.
  if $role = \texttt{judge}$**:**
  $\quad$**send** $(\mathsf{Corrupt}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  $\quad$**to** $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$ — {$\mathcal{F}_{\text{judgeParams}}$ *decides whether judges can be corrupted*
  $\quad$**wait for** b
  $\quad$**return** b
- **DetermineCorrStatus**[b]$(pid, sid, role)$:
  if $role = \texttt{judge}$**:** — {$\mathcal{F}_{\text{judgeParams}}$ *may determine a judge's corruption status*
  $\quad$**send** $(\texttt{CorruptionStatus?}, (pid, sid, \texttt{judge}), \mathsf{internalState})$
  $\quad$**to** $(pid, sid, \mathcal{F}_{\text{judgeParams}} : \texttt{judgeParams})$
  $\quad$**wait for** b; **return** b
- **AllowAdvMessage**$(pid, sid, role, pid_{\texttt{receiver}}, sid_{\texttt{receiver}}, role_{\texttt{receiver}}, m)$: $\mathcal{A}$ is not allowed to call subroutines on behalf of a corrupted party.

[a] $\mathsf{Roles}_{\mathcal{F}}$ is the set of (main) roles provided by $\mathcal{F}$ to the environment. For example, $\mathsf{Roles}_{\mathcal{F}} = \{\texttt{signer}, \texttt{verifier}\}$ for an ideal signature functionality $\mathcal{F} := \mathcal{F}_{\text{sig}}$.

[b] **DetermineCorrStatus** allows protocol designers to specify whether an entity that is currently not directly controlled by the attacker should nevertheless consider itself to be corrupted. E.g., a local judge will typically consider itself to be corrupted already if its corresponding party is corrupted.

Fig. 21: The ideal ledger functionality $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ (Part 1).

Description of $M_{\texttt{client}}$ (continued):

Initialization:

send InitMe **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{init}} : \mathtt{init})$ — {$\mathcal{F}_{\mathrm{init}}$ *handels initilaization if necessary.*

wait for $(\mathtt{Init}, \mathit{identities}, \mathit{msglist}, \mathit{corrupted}, \mathit{leakage})$ **s.t.**

1. $\mathit{identities} \subset \{0,1\}^* \times \{0\}$, $\mathit{round} \in \mathbb{N}$, $\mathit{msglist} \subset \mathbb{N} \times \mathbb{N} \times \{\mathtt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*\}$,
2. $\mathit{corrupted} \subset \{0,1\}^* \times \{\mathsf{sid}_{\mathsf{cur}}\} \times \{\mathtt{client}\}$,
3. $\mathit{msg} \in \mathit{msglist}$ are consecutivley enumarted started at 0,
4. $\exists(_,_,_,_,a,b) \in \mathit{msglist}$, **s.t.** $a \neq \bot \vee b \neq \bot$

identities $\leftarrow$ $\mathit{identities}$, msglist $\leftarrow$ $\mathit{msglist}$, CorruptionSet $\leftarrow$ $\mathit{corrupted}$ — {*We enforce formats and total odrder in* msglist.

send responsively $(\mathtt{Init}, \mathit{leakage})$ **to** NET — {*Send leaked information from initilaization to* $\mathcal{A}$.

wait for ack **from** NET

MessagePreprocessing:

recv $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}, \mathit{msg})$ **from** I/O:

  if $(\mathsf{pid}_{\mathsf{cur}}, _) \notin$ identities $\wedge$ msg starts with Submit or Read: — {*Register unknown party before its first submit/read operation*

    identities.add$(\mathsf{pid}_{\mathsf{cur}}, \mathsf{round})$

Main:

recv $(\mathtt{Submit}, \mathit{msg})$ **from** I/O: — {*Submission request from a honest identity*

  send $(\mathtt{Submit}, \mathit{msg}, \mathsf{internalState})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{submit}} : \mathtt{submit})$ — {*Forward request to* $\mathcal{F}_{\mathrm{submit}}$

  wait for $(\mathtt{Submit}, \mathit{response}, \mathit{leakage})$ **s.t.** $\mathit{response} \in \{\mathtt{true}, \mathtt{false}\}$

  if $\mathit{response} = \mathtt{true}$:

    reqCtr $\leftarrow$ reqCtr + 1

    requestQueue.add(reqCtr, round, $\mathsf{pid}_{\mathsf{cur}}$, msg) — {requestQueue.add(_) *equals* requestQueue $\leftarrow$ requestQueue $\cup$ {_}. *Records message, round, identity and its state for "consensus"*

  send $(\mathtt{Submit}, \mathit{response}, \mathit{leakage})$ **to** NET — {*If* $\mathcal{F}_{\mathrm{submit}}$ *leakes data regarding the submitted transaction, this is forwarded to* $\mathcal{A}$.

recv $(\mathtt{Read}, \mathit{msg})$ **from** I/O: — {*Read request from an honest identity*

  send $(\mathtt{InitRead}, \mathit{msg}, \mathsf{internalState})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{read}} : \mathtt{read})$ — {*Forward the request to* $\mathcal{F}_{\mathrm{read}}$.

  wait for $(\mathtt{InitRead}, \mathit{local}, \mathit{leakage})$ **s.t.** $\mathit{local} \in \{\mathtt{true}, \mathtt{false}\}$ — {$\mathit{local} = \mathtt{true}$ *models a "local" read, clients get an immediate response otherwise, it is a network read*

  if local:

    send responsively $(\mathtt{InitRead}, \mathit{leakage})$ **to** NET $(\star)$ — {$\mathcal{F}_{\mathrm{read}}$ *leakes data, this is forwarded to* $\mathcal{A}$.

    wait for $(\mathtt{InitRead}, \mathit{suggestedOutput})$ — {$\mathcal{A}$ *may influence the read processing*

    send $(\mathtt{FinishRead}, \mathit{msg}, \mathit{suggestedOutput}, \mathsf{internalState})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{read}} : \mathtt{read})$

    wait for $(\mathtt{FinishRead}, \mathit{output}, \mathit{leakage}')$

    if $\mathit{output} = \bot$: — {*If* $\mathcal{A}$*'s input for* $\mathcal{F}_{\mathrm{read}}$ *is not accepted, he is triggered again.*

      Go back to $(\star)$ and repeat the request (local variables $\mathit{suggestedOutput}$, output, and $\mathit{leakage}'$ are cleared)

    send responsively $(\mathtt{FinishRead}, \mathit{leakage}')$ **to** NET — {$\mathcal{F}_{\mathrm{read}}$ *leakes data, this is forwarded to* $\mathcal{A}$.

    wait for ack

    reply $(\mathtt{Read}, \mathit{output})$

  else:

    readCtr $\leftarrow$ readCtr+1; readQueue.add(($\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}}$), readCtr, round, msg) — {*In case of network read, store request*

    send $(\mathtt{Read}, \mathsf{readCtr}, \mathit{leakage})$ **to** NET — {*If* $\mathcal{F}_{\mathrm{read}}$ *leakes data, this is forwarded to* $\mathcal{A}$.

recv $(\mathtt{DeliverRead}, \mathit{readCtr}, \mathit{suggestedOutput})$ **from** NET **s.t.** $((\mathit{pid}, \mathit{sid}, \mathit{role}), \mathit{readCtr}, r, \mathit{msg}) \in$ readQueue:

  {$\mathcal{A}$ *tiggers message delivery per message (this may include reordering of messages, non-delivery of messages, and manipulation of delivered data - if not enforced by* $\mathcal{F}_{\mathrm{updRnd}}$*).*

  send $(\mathtt{FinishRead}, \mathit{msg}, \mathit{suggestedOutput}, \mathsf{internalState})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{read}} : \mathtt{read})$

  wait for $(\mathtt{FinishRead}, \mathit{output}, \mathit{leakage}')$

  if $\mathit{output} \neq \bot$:

    send responsively $(\mathtt{FinishRead}, \mathit{readCtr}, \mathit{leakage}')$ **to** NET

    wait for ack

    readQueue.remove(($\mathit{pid}, \mathit{sid}, \mathit{role}$), $\mathit{readCtr}$, r, msg) — {*Clean up* readQueue.

    send $(\mathtt{Read}, \mathit{output})$ **to** $((\mathit{pid}, \mathit{sid}, \mathit{role}), \mathtt{I/O})$

  else:

    send nack **to** NET — {*Delivery request of* $\mathcal{A}$ *was denied*

recv $(\mathtt{CorruptedRead}, \mathit{pid}, \mathit{msg})$ **from** NET **s.t.** $\mathit{pid} \in$ CorruptionSet: — {*Read request from a corrupted identity.*

  send $(\mathtt{CorruptedRead}, \mathit{pid}, \mathit{msg}, \mathsf{internalState})$ **to** $(\mathit{pid}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{read}} : \mathtt{read})$ — {*Forward request to* $\mathcal{F}_{\mathrm{read}}$

  wait for $(\mathtt{FinishRead}, \mathit{leakage})$

  send $(\mathtt{Read}, \mathit{pid}, \mathit{leakage})$ **to** NET — {*Forwarded data to* $\mathcal{A}$.

Fig. 22: The ideal ledger functionality $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger}}$ (Part 2).

After having defined both the real protocol $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ and the ideal protocol $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$, we can now formally state the main result of this section (cf.Theorem 2 in Section IV).

As $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ does not include a clock, we need to adjust interfaces between $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ and $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$ to prove our intended realization statement. We use the wrapper $\mathcal{W}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ that handles clock requests to prove that $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ realizes

Description of $M_{\texttt{client}}$ (continued):

Main:

recv $(\texttt{Update}, msg)$ **from** NET: {*Update or maintain request triggered by the adversary.*

send $(\texttt{Update}, msg, \mathsf{internalState})$ **to** $(\epsilon, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{update}} : \texttt{update})$

wait for $(\texttt{Update}, msglist, updRequestQueue, leakage)$

s.t $msglist \subset \mathbb{N} \times \{\mathsf{round}\} \times \{\texttt{tx}, \texttt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$

{$\mathcal{F}_{\text{update}}$ *outputs which data to append to* msglist *and an updated* requestQueue.

$max \leftarrow \max\{i | (i, _, _, _, _, _) \in \mathsf{msglist}\}$ {*Check that msglist is a totally ordered sequence, extending the existing* msglist. *If* msglist $= \emptyset$ *then* max *defaults to* -1

$check \leftarrow msglist \neq \emptyset \vee updRequestQueue \neq \emptyset$

for $i = max + 1$ to $max + |msglist|$ **do:** {*Check that there exists exactly one entry for every id i in a continous sequence (no gaps)*

if $\nexists_1 (i, _, _, _, _, _) \in msglist$**:**

$check \leftarrow \texttt{false}$

if $\exists (i, _, \texttt{meta}, _, a, b) \in msglist \wedge (a \neq \bot \vee b \neq \bot)$**:** {*Check that* `meta` *data has correct format*

$check \leftarrow \texttt{false}$

if $check$**:** {*If the update is totally ordered and no new messages were added to* requestQueue, *we accept the update.*

msglist.add$(msglist)$

for all $item \in updRequestQueue$ **do:** {*Remove elements "consumed" elements from* requestQueue

requestQueue.remove$(item)$

reply $(\texttt{Update}, check, leakage)$ {*Inform* $\mathcal{A}$ *if update was successful and leake data.*

recv UpdateRound **from** NET: {$\mathcal{A}$ *triggers round update if current round satisfies rules of* $\mathcal{F}_{\text{updRnd}}$.

send $(\texttt{UpdateRound}, \mathsf{internalState})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{updRnd}} : \texttt{updRnd})$

wait for $(\texttt{UpdateRound}, response, leakage)$

if $response = \texttt{true}$**:**

$\mathsf{round} \leftarrow \mathsf{round} + 1$

reply $(\texttt{UpdateRound}, response, leakage)$

recv GetCurRound: {$\mathcal{A}$ *and* $\mathcal{E}$ *are allowed to query the current round.*

reply $(\texttt{GetCurRound}, \mathsf{round})$

recv DeRegister **from** I/O: {*De-register honest party*

Remove the unique tuple $(\mathsf{pid}_{\mathsf{cur}}, r)$ from identities

send responsively DeRegister **to** NET {*Inform* $\mathcal{A}$ *on the deregistration*

wait for ack

reply DeRegister

Include static code from the AUC transformation $\mathcal{T}_1(\cdot)$ [33] here.

Fig. 23: The ideal ledger functionality $\mathcal{F}_{\text{ledger}}^{\text{acc}}$ (Part 3).

Description of the protocol $\mathcal{F}_{\text{init,BB}} = (\texttt{Init})$:

Participating roles: {init}
Corruption model: incorruptible

Description of $M_{\texttt{init}}^{\text{BB}}$:

Implemented role(s): {init}
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

Main:

recv Init: {*Empty initialization.*

reply $(\texttt{Init}, \epsilon, \epsilon, \epsilon, \epsilon)$

Fig. 24: The initialization functionality $\mathcal{F}_{\text{init,BB}}$.

$\mathcal{F}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}^{\text{acc}}$.[20] Also, $\mathcal{W}_{\mathsf{FAB}^*_{\text{BB}}}$ emulates the DeRegister interface an always forwards the message responsively to NET, waits for the replay ack, and then replys DeRegister to the initial sender (cf. the specification for $\mathcal{F}_{\text{ledger}}^{\text{acc}}$ in Figure 23).

Theorem 6. *Let* $\eta \in \mathbb{N}$ *be the security parameter and* $\Sigma = (\mathsf{gen}(1^\eta), \mathsf{sig}, \mathsf{ver})$ *be an EUF-CMA secure signature scheme. Let* $\mathcal{P}_{\mathsf{FAB}^*_{\text{BB}}}^{\text{acc}}$ *be the Fabric*$^*_{\text{BB}}$ *protocol that uses the signature scheme* Σ, $\mathsf{Sec}^{\mathsf{assumption}} = \emptyset$, $\mathsf{Sec}^{\mathsf{acc}} = \{\texttt{consistency}, \texttt{smartRead}\}$, $\mathsf{pids}_{\mathsf{judge}} = \{\texttt{public}\}$, $\mathsf{ids}_{\mathsf{assumption}} = \emptyset$, *and* simulateTx *such that they ensure that read outputs of chaincodeId* $= 0$ *is the message list (as specified, e. g., in Figure 27) and all read contracts such that they ensure that the output is prefixed by the length of the executing peer's blockchain and* v_{ep}, simulateTx

[20] $\mathcal{W}_{\mathsf{FAB}^*_{\text{BB}}}$ is essentially equal to $\mathcal{W}_{\text{BB}}$ from Section III-B but forwards messages to $\mathcal{P}_{\mathsf{FAB}^*_{\text{BB}}}^{\text{acc}}$ for all non-clock related interfaces.

Description of the subroutine $\mathcal{F}_{\text{leak,BB}} = (\texttt{leak})$:

Participating roles: $\{\texttt{leak}\}$ **Corruption model:** incorruptible

Description of $M^{\text{BB}}_{\text{leak}}$:

Implemented role(s): $\{\texttt{leak}\}$ **CheckID**$(pid, sid, role)$**:** Accept all messages with the same sid. **Main:** **recv** $(\texttt{Corrupt}, pid, internalState)$ **from** I/O: {*See Figure 21 for definition of internalState and the local variables it includes* **reply** $(\text{corrupt}, \varepsilon)$

Fig. 25: The leakage subroutine $\mathcal{F}_{\text{leak,BB}}$.

both are deterministic and in polynomial time, further parameters be selected arbitrarily such that all parameterized algorithms are deterministic and in polynomial time, all chaincode IDs are in $\mathbb{N}$, $\mathsf{ch} \in \mathbb{N}$, *and there exists only one write smart contract which appends the transaction to the state without further processing. Let the interaction of the set of clients, orderers, and peers be the empty set and* $\mathsf{S}_{init} \in \{0,1\}^*$. *Let* applyBlock *be a deterministic algorithm matching the explanation in Section IV-A (see the "Block Generation or Cutting" paragraph there) Let* $\mathcal{W}_{\mathsf{FAB}^*_{\text{BB}}}$ *be the wrapper as defined above and* $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ *be the instantiation of* $\mathcal{F}^{\text{acc}}_{\text{ledger}}$ *as described above, where the internal subroutines use the same parameters as* $\mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}}$. *Then:*

$$(\mathcal{W}_{\mathsf{FAB}^*_{\text{BB}}} \mid \mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}}) \leq \mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$$

Proof. As part of the proof, we first define a responsive simulator $\mathcal{S}$ such that the real world running the protocol $\mathcal{R} := (\mathcal{W}_{\mathsf{FAB}^*_{\text{BB}}} \mid \mathcal{P}^{\text{acc}}_{\mathsf{FAB}^*_{\text{BB}}})$ is indistinguishable from the ideal world running $\{\mathcal{S}, \mathcal{I}\}$, with the protocol $\mathcal{I} := \mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$, for every ppt environment $\mathcal{E}$.

The simulator $\mathcal{S}$ is defined as follows: it is a single machine that is connected to $\mathcal{I}$ and the environment $\mathcal{E}$ via their network interfaces. In a run, there is only a single instance of the machine $\mathcal{S}$ that accepts and processes all incoming messages. The simulator $\mathcal{S}$ internally simulates the realization $\mathcal{R}$, including its behavior on the network interface connected to the environment, and uses this simulation to compute responses to incoming messages. For ease of presentation, as mentioned above we will refer to this internal simulation by $\mathcal{R}'$.

Before digging into the details, we emphasize that $\mathcal{I}$ does *not* ensure privacy properties, i. e., all communication/data sent to/from $\mathcal{I}$ via I/O are leaked in plain on NET to $\mathcal{A}/\mathcal{S}$. This allows us to perfectly simulate the real protocol as explained later on.

Network communication from/to the environment

- Messages that $\mathcal{S}$ receives on the network connected to the environment (and which are hence meant for $\mathcal{R}$) are forwarded to internal simulation $\mathcal{R}'$.
- Any messages sent by $\mathcal{R}'$ on its network interface (that are hence meant for the environment or $\mathcal{A}$) are forwarded on the network interface to $\mathcal{E}/\mathcal{A}$.

Corruption handling

- The simulator $\mathcal{S}$ keeps the corruption status of (main and internal) entities in $\mathcal{R}'$ and $\mathcal{I}$ synchronized. That is, whenever a main or internal entity in $\mathcal{R}'$ starts to consider itself corrupted, the simulator first corrupts the corresponding entity of $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ in $\mathcal{I}$ before continuing its simulation.
- Incoming messages from corrupted entities of $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ in $\mathcal{I}$ are forwarded on the network interface to the environment/$\mathcal{A}$ in the name of the corresponding entity in $\mathcal{R}'$. Conversely, whenever a corrupted entity of $\mathcal{R}'$ wants to output a message to a higher-level protocol, $\mathcal{S}$ instructs the corresponding entity of $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ to output the same message to the higher-level protocol.

Transaction submission

Whenever an honest client $(pid, sid, role)$ submits a new message via I/O, i. e.$\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ receives $(\texttt{Submit}, msg)$, the subroutine $\mathcal{F}_{\text{submit},\mathsf{FAB}^*_{\text{BB}}}$ processes the request and $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ then leaks the full submission message to $\mathcal{S}$. When $\mathcal{S}$ receives the leakage from $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$, he inputs the request into $\mathcal{R}'$ and simulates the input.

Description of the protocol $\mathcal{P}^{\mathsf{FAB}^*}_{\mathtt{client}} = (\mathtt{client})$:

Participating roles: $\{\mathtt{client}\}$
Corruption model: *Dynamic corruption without secure erasures*

Description of $M_{\mathtt{client}}$:

Implemented role(s): $\{\mathtt{client}\}$
Subroutines: $\mathcal{F}_{\text{cert}} : \mathtt{signer}, \mathcal{F}_{\text{cert}} : \mathtt{verifier}, \mathcal{F}_{\text{ro}} : \mathtt{randomOracle}, \mathcal{F}_{\text{init}} : \mathtt{init}$
Internal state:

- $\mathsf{corrupted_{init}} \in \{\mathtt{true}, \mathtt{false}\}, \mathsf{corrupted_{init}} = \mathtt{false}$ *{Record whether instance was corrupted on initialization*
- $\mathsf{seqNum} \in \mathbb{N}, \mathsf{seqNum} = 0$ *{Sequence number, replacement for the timestamp*
- $\mathsf{peers_C} \subseteq \{0,1\}^*, \mathsf{peers_C} = \emptyset$ *{The set of peers*
- $\mathsf{orderers_C} \subseteq \{0,1\}^*, \mathsf{orderers_C} = \emptyset$ *{The set of orderers*
- $\mathsf{txPropSet} \subset \{0,1\}^*, \mathsf{txPropSet} = \emptyset$ *{The storage for transaction proposals*
- $\mathsf{endorsments} \subseteq \{0,1\}^{\eta} \times \{0,1\}^*, \mathsf{endorsments} = \emptyset$ *{The endorsement storage*
- $\mathsf{chaincodes} \subsetneq \mathbb{N}, \mathsf{ch} \in \mathbb{N}, \mathsf{chaincodes} = \emptyset$ *{The set of known chaincode ids and the current channel ID*
- $\mathtt{v_{ep}}$[a] *{Algorithms check whether endorsement policies are met*
- $\mathsf{readQueue} : \mathbb{N} \to \mathsf{peers_C} \to \{0,1\}^*$, *{Stores read request for asynchronous handling, initially all ε*
- $\mathsf{receiver} : \mathbb{N} \to (\{0,1\}^*)^3$ *{Maps attaches a receiver to each read request, initially all entries $\perp$*
- $\mathsf{readCtr} \in \mathbb{N}, \mathsf{readCtr} = 0$ *{Counter/IDs for read requests*
- $\mathsf{recentState} \in \mathbb{N}$, initially $\mathsf{recentState} = 0$ *{The most recent state the client saw during a read request*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid and $role = \mathtt{client}$.

Corruption behavior:
DetermineCorrStatus$(pid, sid, role)$:

```
if corrupted = true ∨ corrupted_init = true:          {Checks whether node itself is corrupted.
    return true
corrRes ← corr(pid_cur, (pid_cur, sid_cur, role_cur), signer)     {Request corruption status at F_cert
if corrRes = true:                                  {Checks whether F_cert instance is corrupted
    return true
```

See Appendix B for notation details. We start index counting at 1.

Initialization:

```
send InitClient to (pid_cur, sid_cur, F_init : init)              {Request initialization at InitClient
wait for (InitClient, peers_C, orderers_C, chaincodes, channel, v_ep, corrupted)
peers_C ← peers_C; orderers_C ← orderers_C; chaincodes ← chaincodes; ch ← channel; v_ep ← v_ep; corrupted_init ← corrupted
```

Main:

```
recv (Submit, (ch, chaincodeId, txPayload)) from I/O s.t. chaincodeId ∈ chaincodes:   {Submission of Transaction proposal
  seqNum ← seqNum + 1
  send (Sign, (pid_cur, seqNum, ch, chaincodeId, txPayload)) to (pid_cur, (pid_cur, sid_cur, role_cur), F_cert : signer)
                                                                    {Sign transaction proposal
  wait for (Signature, σ)
  proposal ← (pid_cur, seqNum, ch, chaincodeId, txPayload, σ)
  send (pid_cur, proposal) to (pid_cur, sid_cur, F_ro : randomOracle)          {Generate transaction id
  wait for (pid_cur, txId)
  proposal ← (txId, proposal)
  if w_T(proposal) = true:                               {Check well-formdness of generated transaction
    msg ← ⟨Propose, proposal⟩
    txPropSet.add({proposal})
    msg' ← ε
    for P ∈ peers_C do:                    {For simplicity, broadcast tx proposal to all known peers
      msg' ← msg' ||(P, msg)
    send msg' to NET^C                  {Broadcasting of transaction proposal is a duty of the adversary
recv (Endorsed, txEnd, σ) from NET:                {Receive a Transaction Endorsement From Peers
  if txEnd = (pid_P, txId, seqNum_P, chaincodeId, proposal, resp, readset, writeset) ∧ proposal ∈ txPropSet
   ∧ pid_P ∈ peers_C ∧ txId, chaincodeId match txId_proposal, chaincodeId_proposal from proposal:
    send (Verify, txEnd, σ) to (pid_cur, (pid_P, sid_cur, peer), F_cert : verifier)
    wait for (VerResult, b)
    if b:                                  {If the signature is valid, the endorsement is recorded
      endorsments.add({(txId, txEnd, σ)})
```

[a] Note that the endorsement verification procedure $\mathtt{v_{ep}}$ has two duties: $\mathtt{v_{ep}}$ allows clients to verify whether a planned transaction commit should be accepted without having access to a state, $\mathtt{v_{ep}}$ allows peers to verify whether they accept transactions during endorsement generation and state generation

Fig. 26: The Fabric model $\mathcal{P}^{\mathsf{FAB}^*}_{\mathtt{client}}$, specification of the Fabric* client $M_{\mathtt{client}}$ (Part 1)

After simulating the input to the (honest) client $entity$ in $\mathcal{R}'$, $\mathcal{S}$ outputs the result of the activation, i. e., the simulated transaction proposal, via NET to $\mathcal{E}$, resp. $\mathcal{A}$.

Description of $M_{\texttt{client}}$ (cont.):

Main:

```
recv (Commit, pid_O, txId) from NET s.t. pid_O ∈ orderers_C ∧ (txId, ·) ∈ txPropSet:        {Send commit request to orderer
    T_txId ← {(txId, proposal) | (txId, proposal) ∈ txPropSet}
    T_txId.add({(t, msg) ∈ endorsments | t = txId})                    {If the transaction commitment T_txId meets
    if v_ep(T_txId) = true                                             {endorsement policy, request the ordering:
        send (pid_O, ⟨Commit, txId, T_txId⟩) to NET                    {service to commit the transaction
recv (Read, msg) from I/O:                             {E/a higher-level protocol calls a chaincode/queries Fabric*
                                               {We model, w.l.o.g. that chaincode ID 0 outputs the full message list of
    if msg = (pid_P, 0, _) ∧ pid_P ∈ peers_C:  {a channel to model a BB. Therefore, our statements only hold for a
        readCtr ← readCtr + 1; seqNum ← seqNum + 1      {restricted set of parameters, e. g., simulateTx.
        receiver[readCtr] ← (pid_call, sid_call, role_call)
        readQueue[readCtr, pid_P] ← open
        send (Sign, (readCtr, pid_cur, seqNum, ch, 0, 0)) to (pid_cur, (pid_cur, sid_cur, role_cur), F_cert : signer)
        wait for (Signature, σ)
        send (Read, (readCtr, pid_cur, seqNum, ch, 0, 0, σ), pid_P) to NET        {A is responsible for dispatching the messages.
    else if msg = (pid_P, id, query) ∧ pid_P ∈ peers_C:               {Query that does not aim for Fabric* full message list
        readCtr ← readCtr + 1; seqNum ← seqNum + 1; receiver[readCtr] ← (pid_call, sid_call, role_call)
        readQueue[readCtr, pid_P] ← open
        send (Sign, (readCtr, pid_cur, seqNum, ch, id, query)) to (pid_cur, (pid_cur, sid_cur, role_cur), F_cert : signer)
        wait for (Signature, σ)
        send (Read, (readCtr, pid_cur, seqNum, ch, id, query, σ), pid_P) to NET   {A is responsible for dispatching the messages.
recv (DeliverRead, pid_cur, (pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset), σ_P) from NET
  s.t. readQueue(queryID, pid_P) = open ∧ queryToBeProcessed = (queryID, seqNum_C, pid_cur, ch, chaincodeId, query, σ):
                                                                                {Response to Read request
    send (Verify, (queryID, seqNum_C, pid_cur, ch, chaincodeId, query), σ) to (pid_cur, (pid_cur, sid_cur, role_cur), F_cert : verifier)
    wait for (VerResult, b_1)                                                   {Check correctness of request
    send (Verify, (pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset, σ_P))
      to (pid_cur, (pid_P, sid_cur, peer), F_cert : verifier)
    wait for (VerResult, b_2)                                                   {Check correcly signed endorsement
    if b_1 ∧ b_2:
        if ∃ an entry readQueue[queryID, _] = open:
            if chaincodeId = 0:                       {In case of chaincodeId = 0, resp is a message list (cf. Figure 37)
                if recentState ≤ |resp|:                   {Check that client has not seen newer data than resp
                    recentState ← |resp|
                    send (Evidence, (pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset, σ_P))
                      to (public, sid_cur, F_judge : judge)                     {Forward evidence to F_judge
                    wait for ack
                else:
                    if resp is prefixed with ptr' and ptr' ≥ recentState:   {Only accept output that indicated to be generated on recent state
                        recentState ← ptr'
                        send (Evidence, (pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset, σ_P))
                          to (public, sid_cur, F_judge : judge)                 {Forward evidence to F_judge
                        wait for ack
            readQueue[queryID, _] ← done
            send (Read, resp) to receiver[readCtr]
```

Fig. 27: Specification of the Fabric* client $\mathcal{P}^{\mathsf{FAB}^*}_{\texttt{client}}$ (Part 2)

$\mathcal{S}$ stores the leaked request IDs from $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$ (including message and submitter) for later processing the updates of the write queue.

Read requests

Whenever an honest client *entity* in $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$ receives a request $(\texttt{Read}, msg)$, $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$ leaks this request to $\mathcal{S}$ and waits for $\mathcal{S}$ to (asynchronously) provide the necessary details to produce and deliver the response to *entity*. $\mathcal{S}$ simulates the read request in $\mathcal{R}'$ and stores the ID of the read request for later processing internally. $\mathcal{S}$ extracts the output of the read request the simulated entity would output via I/O. If `consistency` is `publicly` broken, $\mathcal{S}$ sends the output directly to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$. Otherwise, if the request aimed for getting the full state from the BB, i. e., $\mathit{chaincodeId} = 0$, $\mathcal{S}$ extracts the highest ID from the output generated in $\mathcal{R}'$ and forwards this pointer to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$. If $\mathit{chaincodeId} \neq 0$ and `smartRead` is `publicly` broken, $\mathcal{S}$ sends the output directly to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$. Otherwise, it extracts the state pointer from the endorsement which triggers the I/O output and sends this as $\mathit{suggestedOutput}$ to $\mathcal{I}$. In all cases above, the instruction to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$ includes the formerly stored read request ID which $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$ initially leaked to $\mathcal{S}$.

Verdicts

In the case that the judge renders a verdict in $\mathcal{R}'$. $\mathcal{S}$ extracts the verdict v from the judge and provides it to $\mathcal{I}$. As we do not provide further guarantees as soon we have a verdict, $\mathcal{S}$ sends $(\texttt{BreakAccProp}, v, \{(\texttt{consistency}, \texttt{publicly}), (\texttt{smartRead}, \texttt{public})\})$ to $\mathcal{I}$.

Judicial reports

In the case of someone requesting a judicial report via I/O, $\mathcal{I}$ will trigger $\mathcal{S}$ via the GetState interface. In this case, $\mathcal{S}$ simulates the input of GetJudicialReport to its simulated judge in $\mathcal{R}'$. It then extracts the highest ID id from the output judge produces and forwards it via $(\texttt{GetState}, id)$ to $\mathcal{I}$.

State updates

As soon as $\mathcal{S}$ sees an state update in $\mathcal{R}'$, he updates $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$'s state. More specifically, the following cases are interpreted as a "state update" in $\mathcal{R}'$: *(i)* msglist of an honest client, *(ii)* msgStream of the judge, *(iii)* chain of an honest peer, *(iv)* msgStream of an honest orderer, or *(v)* msgStream of an honest Kafka cluster extend such that the former "longest" state $\mathcal{S}$ was aware of is a prefix of the new state (and after mapping them from their data object to a totally ordered sequence of messages) – in what follows, we will sometimes call the longest list of messages extracted from these parties the *state of* $\mathcal{R}'$. $\mathcal{S}$ extracts the differences between the former state and the state update in the form $[(i, msg_i), (i+1, msg_{i+1}), \ldots]$. It checks whether the message matches a message from the stored write queue (see message submission) and extends with this information to $[(i, reqCtr_i, msg_i), (i+1, reqCtr_{i+1}, msg_{i+1}), \ldots]$ where $reqCtr_j = \varepsilon$ if $\mathcal{S}$ cannot find a matching submission ID. $\mathcal{S}$ then sends $(\texttt{Update}, [(i, reqCtr_i, msg_i), (i+1, reqCtr_{i+1}, msg_{i+1}), \ldots])$ to $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$. Further, $\mathcal{S}$ also removes the items with a valid $reqCtr$ from its internal write queue.

Further details

$\mathcal{S}$ keeps the clocks/rounds of $\mathcal{R}'$ and $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ synchronous. That is, $\mathcal{S}$ sends UpdateRound to $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ whenever a round update in the simulated $\mathcal{W}_{\mathsf{FAB}^*_{\text{BB}}}$ is performed and before continuing the simulation.

This concludes the description of the simulator. It is easy to see that *(i)* $\{\mathcal{S}, \mathcal{I}\}$ is environmentally bounded [21] and *(ii)* $\mathcal{S}$ is a responsive simulator for $\mathcal{I}$, i. e., restricting messages from $\mathcal{I}$ are answered immediately as long as $\{\mathcal{S}, \mathcal{I}\}$ runs with a responsive environment. We now argue that $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$ are indeed indistinguishable for any (responsive) environment $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$.

Now, let $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$[22] be an arbitrary but fixed environment. In the following, we will go over all possible interactions on the network and the I/O interface and argue, by induction, that all of those interactions result in identical behavior towards the $\mathcal{E}$, i. e., $\mathcal{I}$ and $\mathcal{R}$ are indistinguishable. At the start of a run, there were no interactions on the network, resp. I/O, interface yet. Thus, the induction base case holds true. In the following, assume that all network, resp. I/O, interactions so far have resulted in the same behavior visible towards the environment in both the real and ideal world. In the argumentation below, we mainly argue the case where no verdict is present – as in this case, $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$ ensures its properties and its restrictions need to be met. We discuss afterward that $\mathcal{S}$'s simulation also ensures indistinguishable between real and ideal world in the presence of verdicts.

We note that the argumentation in the following indirectly relies on the case that signatures cannot be forged as the signature scheme we use in the real world is EUF-CMA. Thus, the probability that the signature can be faked in $\mathcal{R}$ is negligible.

Corrupting parties: In the following, we argue that $\mathcal{S}$ is always able to keep corrupting in $\mathcal{R}'$ and $\mathcal{I}$ synchronous. As there are no restrictions for corruption in $\mathcal{I}$, we have nothing to show here: For corrupted parties, $\mathcal{S}$ forwards the output $\mathcal{A}$ instructed the corrupted party to output to $\mathcal{I}$ with the same forwarding request. As parties are also corrupted in $\mathcal{I}$, $\mathcal{I}$ will also forward the message. Thus, the I/O behavior in real and ideal world are identical in this case. This holds true for main parties as well as for internal parties.

Interaction via network:

As already mentioned, $\mathcal{I}$ leaks all data in plain to $\mathcal{S}$ including sufficient information about all requests performed by higher-level protocols. In particular, $\mathcal{S}$ has access to all necessary data to provide the correct leakage via NET to $\mathcal{E}/\mathcal{A}$

[21] As all algorithms are in polynomial time and parameters ensure that the execution of non-a-priori fixed code finishes in polynomial time.

[22] For some system $\mathcal{Q}$, we denote by $\mathsf{Env}(\mathcal{Q})$ the set of all environments $\mathcal{E}$ that can be connected to $\mathcal{Q}$.

or to provide data in plain in case corrupted parties are involved in a transaction. As a result, the network behavior simulated by $\mathcal{S}$ towards the environment is indistinguishable from the network behavior of $\mathcal{R}$.

Interaction (of honset parties) via I/O: Firstly, we show that the I/O behavior simulated by $\mathcal{S}$ towards the environment is indistinguishable from the I/O behavior of $\mathcal{R}$.

Submission requests: By construction of $\mathcal{R}$ and $\mathcal{I}$, submission requests do not directly result in an input to the environment but they might influence future read requests. Thus, the main goal here is to prove that the states of $\mathcal{R}'$ and $\mathcal{I}$ stay "synchronized" after message submissions, i. e., the set of potential messages submitted by honest clients is equal in $\mathcal{R}'$ and $\mathcal{I}$. If a party submits a message to $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$, it adds the message to its write queue ($\mathsf{requestQueue}$) if it matches the expected format and is dedicated to the $\text{Fabric}^*_{\text{BB}}$ channel we are currently considering. The submission message is forwarded to $\mathcal{S}$ which simulates its input to $\mathcal{R}'$. Also, the simulation considers the message as a valid update candidate for the state as $\mathcal{R}$ and $\mathcal{I}$ execute the same message validations. In $\mathcal{R}'$, we consider a message to be queued as soon as a client tries to send it to a peer. Thus, the write queues in $\mathcal{I}$ and $\mathcal{R}$ are, resp. stay, synchronized when an honest transaction is newly submitted to $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$.

State updates:

Though state updates do not directly lead to outputs to I/O, $\mathcal{I}$'s state is the basis for answers to read requests. Thus, we show here that $\mathcal{I}$ always accepts state updates provided by $\mathcal{S}$ and thus – by construction of $Ssys$ – $\mathcal{I}$'s state is always a prefix of the state of all honest parties in $\mathcal{R}'$. This allows us to conclude later on, that read requests are processed as $\mathcal{S}$ desired.

We already concluded above that the set of $\mathsf{requestQueue}$ is synchronized between $\mathcal{I}$ and $\mathcal{R}'$. Further, we note that $\mathcal{I}$ does not impose further limitations to state updates besides that messages need to be totally ordered, starting at the current state's highest ID plus one. Indeed, $\mathcal{S}$ matches these requirements. By induction base case, we assume that the last states between $\mathcal{I}$ and $\mathcal{R}'$ are synchronized, i. e., they also have the same highest ID. Thus, $\mathcal{S}$ extracted extension starts with the same highest ID plus one. As defined, $\mathcal{S}$ provides a state extension to $\mathcal{I}$ which is the current "longest" state known by honest parties (and this state is a prefix of the states of all honest parties). Thus, $\mathcal{I}$'s new state is also a prefix of all honest parties' states. Thus, states in $\mathcal{R}'$ and $\mathcal{I}$ are the same after $\mathcal{S}$ updates $\mathcal{I}$'s state.

Though less relevant, the update provided by $\mathcal{S}$ will also keep the $\mathsf{requestQueues}$ of $\mathcal{I}$ and $\mathcal{R}'$ synchronized (as by construction IDs match between $\mathcal{I}$ and $\mathcal{R}'$).

Read requests:

Whenever an honest entity $entity$ receives a request $(\texttt{Read}, msg)$ to read from the BB's state, $\mathcal{I}$ forwards this read request (in plain) to $\mathcal{S}$ (including an identifier) and waits to receive a suggested output. $\mathcal{S}$ simulates the read request internally by forwarding the request to the simulated instance of $\mathcal{P}^{\mathsf{FAB}^*}_{\texttt{client}}$ in $\mathcal{R}'$. As defined above, $\mathcal{S}$ extracts the output from the (asynchronously) simulated response, i. e., (as we only consider the absence of verdicts here) *(i)* the highest ID of the message list in case of $chaincodeId = 0$ or *(ii)* the state pointer in case of a call to another smart contract.

In particular, $\mathcal{S}$'s suggested output for the read request will not fail $\mathcal{I}$'s validations and will lead to the same output:

1. By construction of the state updates, $\mathcal{I}$'s state is always a prefix of $\mathcal{R}'$ state. Thus, the ID extracted from honest parties in $\mathcal{R}'$ is always lower (or equal) to the highest ID in $\mathcal{I}$.
2. As consistency holds true in this argumentation here, $\mathcal{I}$ will output its prefix up to the message ID provided by $\mathcal{S}$ in case we consider $chaincodeId = 0$. By construction, this is the same message sequence in $\mathcal{R}'$ and $\mathcal{I}$. Thus, the output to I/O is the same. Thus, we can conclude that real and ideal world remain indistinguishable in this case.
3. In case of another read request (and as long as $\texttt{smartRead}$ holds), $\mathcal{S}$ provides the state pointer based on which the output was computed to $\mathcal{I}$ (as above, the ID provided will be smaller or equal to the highest message ID in $\mathcal{I}$). By construction of $\mathcal{F}_{\text{read},\mathsf{FAB}^*_{\text{BB}}}$ in $\mathcal{F}^{\text{acc}}_{\text{ledger},\mathsf{FAB}^*_{\text{BB}}}$, the input for the smart contract (contained in $\mathsf{simulateTx}$) is the same in $\mathcal{R}'$ and $\mathcal{I}$. Thus, also their output is the same (recall that $\mathsf{simulateTx}$ is equal in $\mathcal{R}'$ and $\mathcal{I}$ and is required to be deterministic). Thus, the output to I/O is the same and real and ideal world remain indistinguishable.

   Remark:

   We briefly discuss two cases where the above explanation might fall short:

   a) **Full reads: A corrupted peer provides a longer full read output than the currently longest $\mathsf{msgStream}$:** As we consider the case, where no verdict is present so far and a corrupted peer provides a message stream

$\mathsf{msgStream}'$ to an honest client, such that the $|\mathsf{msgStream}'| > |\mathsf{msgStream}|$ (from $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$). When this case appears, the simulation provides this output message to the judge which then renders a verdict against the corrupted peer (as the state is not a prefix of the currently valid state) and the verdict is then "forwarded" to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$. Thus, we are no longer in the case, that there is no verdict and simulation will succeed as explained later on.

b) **Smart contracts: A corrupted peer provides a smart contract output based on a state pointer that is longer than the current** $\mathsf{msgStream}$**:** Consider the case, where no verdict is present so far and a corrupted peer provides smart contract output to an honest client, such that $ptr > |\mathsf{msgStream}|$ where ptr is the state pointer from the response to the client and $\mathsf{msgStream}$ the variable from $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$. When this case appears, the simulation provides this output message to the judge which then renders a verdict against the corrupted peer (as the judge is not able to recreate the state based on which the computation is executed). Again, the simulator "forwards" the verdict to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$. Thus, we are no longer in the case, that there is no verdict and simulation will succeed as explained later on.

Further interactions:

Verdicts:

Verdicts from $\mathcal{S}$ do not violate the rules of $\mathcal{I}$, thus $\mathcal{I}$ will accept verdicts provided by $\mathcal{S}$. Also, note that verdicts always blame corrupted parties. This was already mainly shown in [31]. In what follows, we recall the argumentation from Graf et al. and additionally discuss that the new (blue) parts of $\mathcal{F}_{\mathrm{judge}}$ in Figure 32 also only blames corrupted parties. Note that we assume here that signatures are correctly generated and cannot be forged (we already handled the probability of a forged signature above). In what follows, we mainly step over $\mathcal{F}_{\mathrm{judge}}$'s code/verdicts in the order they appear:

When peers provide evidence to the judge, then honest peers do not report *(i)* malformed blocks, *(ii)* chains with gaps, *(iii)* blocks that are not generated by orderers, *(iv)* blocks with invalid signatures, or *(v)* blocks containing messages with invalid Merkle proofs or wrong Kafka signature. Thus, none of these checks leads to a verdict against an honest party.

In the function `analyzeConflBlocks`, the judge blames the Kafka cluster, if there exist two messages for the same message ID which were correctly generated and signed by a Kafka entity. By construction of $\mathcal{F}_{\mathcal{K}}$, this case never appears if $\mathcal{F}_{\mathcal{K}}$ operates honestly.

Also, the judge blames orderers if the judge cannot recreate the blocks they created (according to $\mathrm{Fabric}^*_{\mathrm{BB}}$'s deterministic block-cutting algorithm). Honest peers would not create blocks that deviate from the intended block-cutting algorithm-

When clients report data, the judge blames a peer when it provides a manipulated full-read output to a client that deviates from the so-far unique message stream. Assuming the peer acted honestly and an orderer and/or the Kafka cluster delivered a manipulated message stream/blockchain to the peer. However, assuming the peer is honest leads to the fact that the peer would have reported its blockchain (including manipulated data from the orderer and Kafka) already previously to the judge and the judge would then have rendered a verdict blaming the orderer or Kafka. As this is not the case, this contradicts the assumption and shows that the peer is indeed corrupted.

When the judge checks the output of a smart contract/read was computed correctly, it first checks whether the state the peer used to calculate the output (indicated by the state pointer from the signed response) is a valid starting point for the computation. By construction, only corrupted peers hand in invalid starting points for the computation.

In the second step, the judge recomputes the output of the peer. The recomputation is indeed possible, as the smart contracts (in $\mathsf{simulateTx}$) are all deterministic (by assumption) and we can reconstruct the same input(s) to $\mathsf{simulateTx}$ in the judge (based on the correct state reconstructed from the state pointer provided by $\mathcal{S}$). By construction, only corrupted peers would not provide the output from $\mathsf{simulateTx}$ in this case.

Thus, we can conclude that $\mathcal{I}$ always accepts verdicts from $\mathcal{S}$.

Judicial Reports:

If the consistency and smart read are not yet broken, $\mathcal{I}$ always accepts the input (pointer) from $\mathcal{S}$ for generating the judicial report: The state of $\mathcal{R}'$ is always a superset of the state of $\mathcal{I}$ (typically equal) and $\mathcal{S}$ updates $\mathcal{I}$ always such that provided pointers are in the state of $\mathcal{I}$.

When consistency or smart read are violated, the judicial report is empty in $\mathcal{R}$ and $\mathcal{I}$.

Thus, $\mathcal{R}$ and $\mathcal{I}$ remain indistinguishable when parties query judicial reports via I/O.

Current time requests:
As there are no restrictions for updating the time in $\mathcal{R}$ and $\mathcal{I}$ the simulator can hold clocks in $\mathcal{R}'$ and $\mathcal{I}$ synchronized. Thus, requests to the clock cannot be used to distinguish both worlds.

Deregistration: As $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ does not include the concept of deregistration, $\mathcal{W}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ mimics the behavior of $\mathcal{I}$'s `DeRegister` interface. As the code is the same here, the behavior on this interface is indistinguishable between $\mathcal{I}$ and $\mathcal{R}$.

Simulation in presence of a verdict
The simulation in case of broken accountability properties obviously is indistinguishable between $\mathcal{R}$ and $\mathcal{I}$: *(i)* We stress that $\mathcal{S}$ can perfectly simulate $\mathcal{R}'$ as he has access to all necessary data. *(ii)* All I/O output of the `Read` interface can be freely determined by $\mathcal{S}$ (who extracts when and what to send to whom as part of the simulation). *(iii)* Judicial reports are empty in both worlds in this case. *(iv)* Verdicts stay synchronized between $\mathcal{R}'$ and $\mathcal{I}$. For all other interfaces, the argumentation from above holds true.

Altogether, $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$ behave identically in terms of behavior visible to the environment $\mathcal{E}$ and thus are indistinguishable.

□

As the subroutines of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$ are compatible with the subroutine definitions in front of Lemma 5, we can conclude that $(\mathcal{W}_{\mathrm{BB}} \mid \mathcal{W}_{\mathsf{FAB}^*_{\mathrm{BB}}}, \mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}})$ realizes $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$ (where $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$ is the instance of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB}}$ which we get, if we reversely apply the subroutine construction before Lemma 5 to $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$).

Thus, we can conclude:

Corollary 7. *Let $\eta \in \mathbb{N}$ be the security parameter and $\Sigma = (\mathsf{gen}(1^\eta), \mathsf{sig}, \mathsf{ver})$ be an EUF-CMA secure signature scheme. Let $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$ be the Fabric$^*_{\mathrm{BB}}$ protocol that uses the signature scheme Σ, $\mathsf{Sec}^{\mathsf{assumption}} = \emptyset$, $accSecProperties = \{\mathtt{consistency}, \mathtt{smartRead}\}$, $\mathsf{pids}_{\mathsf{judge}} = \{\mathtt{public}\}$, $\mathsf{ids}_{\mathsf{assumption}} = \emptyset$, and $\mathsf{simulateTx}$ such that they ensure that read outputs of chaincodeId $= 0$ is the message list (as specified, e. g., in Figure 27) and all read contracts such that they ensure that the output is prefixed by the length of the executing peer's blockchain and $\mathtt{v_{ep}}, \mathsf{simulateTx}$ both are deterministic and in polynomial time, further parameters be selected arbitrarily such that all parameterized, all chaincode IDs are in $\mathbb{N}$ algorithms are deterministic and in polynomial time, and there exists only one write smart contract which appends the transaction to the state without further processing. Let $Wsys_{\mathsf{FAB}^*_{\mathrm{BB}}}$ and $\mathcal{W}_{\mathrm{BB}}$ be the wrappers as defined above and $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}$, resp. $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$, be the instantiation of $\mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger}}$, resp. $\mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB}}$, as described above, where the internal subroutines use the same parameters as $\mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}$. Then:*

$$(\mathcal{W}_{\mathrm{BB}} \mid \mathcal{W}_{\mathsf{FAB}^*_{\mathrm{BB}}}, \mathcal{P}^{\mathrm{acc}}_{\mathsf{FAB}^*_{\mathrm{BB}}}) \leq (\mathcal{W}_{\mathrm{BB}} \mid \mathcal{F}^{\mathrm{acc}}_{\mathrm{ledger},\mathsf{FAB}^*_{\mathrm{BB}}}) \leq \mathcal{F}^{\mathrm{acc}}_{\mathrm{cBB},\mathsf{FAB}^*_{\mathrm{BB}}}$$

Proof. The proof follows directly from Theorem 6, Lemma 5, and iUC's composition theorem. □

Description of $\mathcal{F}_{\mathcal{K}} = (\texttt{consens})$:

Participating roles: $\{\texttt{consens}\}$
Corruption model: *Dynamic corruption without secure erasures*

Description of $M_{\texttt{consens}}$:

Implemented role(s): $\{\texttt{init}\}$
Subroutines: $\mathcal{F}_{\text{cert}} : \texttt{signer}, \mathcal{F}_{\text{ro}} : \texttt{randomOracle}$
Internal state:

- $\mathsf{corrupted_{init}} \in \{\texttt{true}, \texttt{false}\}, \mathsf{corrupted_{init}} = \texttt{false}$ — *{Record whether instance was corrupted on initialization*
- $\mathsf{orderers} \subseteq \{0,1\}^*, \mathsf{orderers} = \emptyset$ — *{The set of orderers*
- $\mathsf{kafka} \subset \{0,1\}^*, \mathsf{kafka} = \emptyset$ — *{The set of Kafka borkers*
- $\mathsf{pid_{pid_{KL}}} \in \mathsf{kafka}, \mathsf{ch} \in \mathbb{N}$ — *{Current "leader" of the Kafka cluster and the channel id*
- $\mathsf{seqNum} \in \mathbb{N}$, initially $\mathsf{seqNum} = 0$ — *{Sequence number, resp. message counter or offset*
- $\mathsf{msgStream} \subseteq \mathbb{N} \times \mathsf{kafka} \times \{0,1\}^* \times (\{0,1\}^* \cup \{\bot\})^3, \mathsf{msgStream} = \emptyset,$ — *{The ordered stream of messages/transactions of the form $(offset, leader, msg, merkleRoot, merkleProof, signature)$*

CheckID$(pid, sid, role)$:
If $\mathsf{kafka} = \emptyset$, accept all messages with the same sid and $role = \texttt{kafka}$, otherwise, accept all messages with same sid, $role = \texttt{kafka}$, and $pid \in \mathsf{kafka}$.

Corruption behavior:
DetermineCorrStatus$(pid, sid, role)$:
if $\mathsf{corrupted} = \texttt{true} \vee \mathsf{corrupted_{init}} = \texttt{true}$**:** *{Checks whether instance itself is corrupted.*
  return true
$corrRes \leftarrow \texttt{false}$
for all $pid \in \mathsf{kafka}$ **do:**
  $corrRes \leftarrow corrRes \vee \mathbf{corr}^{a}(pid, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), \texttt{signer})$ *{Request corruption status of Kafka brokers at $\mathcal{F}_{\text{cert}}$*
if $corrRes = \texttt{true}$**:** *{Checks whether $\mathcal{F}_{\text{cert}}$ instance is corrupted*
  return true

See Appendix B for notation details. We start index counting at 1.

Initialization:
send $\texttt{InitKafka}$ **to** $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\text{init}} : \texttt{init})$
wait for $(\texttt{InitKafka}, ch, pid_{pid_{KL}}, orderers, kafka, corrupted)$
$\mathsf{ch} \leftarrow ch$; $\mathsf{pid_{pid_{KL}}} \leftarrow pid_{pid_{KL}}$; $\mathsf{kafka} \leftarrow kafka$; $\mathsf{orderers} \leftarrow orderer$; $\mathsf{corrupted_{init}} \leftarrow corrupted$

Main:
recv $(\texttt{Commit}, msg, pid_O)$ **from** NET **s.t.** $pid_O \in \mathsf{orderers}, \mathsf{pid_{cur}} = \mathsf{pid_{pid_{KL}}}, \texttt{extCh}(msg) = \mathsf{ch}$**:** *{Leader should add and order messages*
  $\mathsf{seqNum} \leftarrow \mathsf{seqNum} + 1$ *{Messages are ordered according to their arrival at $\mathcal{F}_{\mathcal{K}}$*
  $\mathsf{msgStream.add}(\mathsf{seqNum}, \mathsf{pid_{pid_{KL}}}, msg, \bot, \bot, \bot)$ *{Merkle root and proof are generated upon request*

recv $(\texttt{Pull}, pid_O, \mathsf{ch}, r)$ **from** NET **s.t.** $pid_O \in \mathsf{orderers}, \mathsf{pid_{cur}} = \mathsf{pid_{pid_{KL}}}, r \leq \mathsf{seqNum}$**:** *{Pull request: request messages with ID $\geq r$*
  $msgStream' \leftarrow \{(seq, pid, msg, mr, mp, \sigma) \mid (seq, pid, msg, \sigma) \in \mathsf{msgStream} \wedge seq \geq r\}$ *{Extract requested data from msgStream*
  if $\exists(_,_,_,\bot,\bot,\bot) \in msgStream'$**:** *{There were no Merkle proof/valid signatures attached to these messages before*
    $j \leftarrow \min\{i \mid (i,_,_,\bot,\bot,\bot) \in msgStream'\}$; $M \leftarrow \varepsilon$; $N \leftarrow \emptyset$ *{M, N will be used to update msgStream*
    for $i = j$ to $|msgStream'|$ **do:** *{Generate Merkle proofs and signatures*
      $t \leftarrow (i, \mathsf{pid_{pid_{KL}}}, msg)$, **s.t.** $(i, pid, msg, \bot, \bot, \bot) \in msgStream'$
      $\bar{t} \leftarrow (i, pid, msg, \bot, \bot, \bot)$, **s.t.** $(i, pid, msg, \bot, \bot, \bot) \in msgStream'$
      $M.\mathsf{add}(t), N.\mathsf{add}(\bar{t})$
    $U \leftarrow \texttt{processMsgReq}(M)$ *{See definition of* processMsgReq *below*
    $\mathsf{msgStream} \leftarrow \mathsf{msgStream} \setminus N \cup U$ *{Add Merkle proofs to storage*
    $msgStream' \leftarrow msgStream' \setminus N \cup U$ *{Add updated data in prepared answer*
  reply $(\texttt{Deliver}, pid_O, \mathsf{ch}, r, msgStream')$

recv $(\texttt{SetLeader}, pid_{KL}, \mathsf{ch})$ **from** NET **s.t.** $\mathsf{pid_{cur}}, pid_{KL} \in \mathsf{kafka}$**:** *{Set new Kafka leader*
  $\mathsf{pid_{KL}} \leftarrow pid_{KL}$
  send ack **to** NET *{Acknowledge*

Procedures and Functions:
function $\texttt{processMsgReq}$ $(M$ **s.t.** $M = (msg_1, msg_2, \dots, msg_m), msg_j \in \{0,1\}^*, j \in \{1, \dots, m\})$**:**
  $mr \leftarrow \texttt{genMerkleTreeRoot}^{b}(M)$
  send $(\texttt{Sign}, mr)$ **to** $(\mathsf{pid_{cur}}, (\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathsf{role_{cur}}), \mathcal{F}_{\text{cert}} : \texttt{signer})$ *{Sign Merkle Root*
  wait for $(\texttt{Signature}, \sigma)$
  for $i = 1$ to m **do:**
    $mp_i \leftarrow \texttt{genMerkleProof}^{c}(M, i)$
  return $\{(msg_1, mr, mp_1, \sigma), \dots, (msg_m, mr, mp_m, \sigma)\}$

[a] The **corr** macro is a iUC macro which sends CorruptionStatus? to the specified entity, waits for the response, and then outputs that response.

[b] genMerkleTreeRoot gets as input a sequence of messages. It outputs the root of the Merkle tree over the input. Note that it is possible to generate a Merkle root in $O(\log(n))$ where n is the length of the input sequence. For a detailed specification, we refer to [59].

[c] genMerkleProof gets as input a sequence of messages and an index x. It outputs Merkle proof for the message at position x. Note that it is possible to generate a Merkle proof in $O(\log(n))$ where n is the length of the input sequence. For a detailed specification, we refer to [59].

Fig. 28: The model of an idealized Kafka cluster $\mathcal{F}_{\mathcal{K}}$ for the Fabric* model.

Description of the protocol $\mathcal{F}_{\text{init}} = (\mathtt{init})$:

Participating roles: $\{\mathtt{init}\}$
Corruption model: *incorruptible*
Protocol parameters:

- $\mathsf{clients}, \mathsf{orderers}, \mathsf{peers}, \mathsf{kafka}$ — $\{$*The set of clients, orderers, peers, and Kafka brokers such that* $\mathsf{clients} \cap \mathsf{peers} \cap \mathsf{orderers} \cap \mathsf{kafka} \neq \emptyset$
- $\mathsf{S}_{\mathsf{init}}$ — $\{$*The initial state of the blockchain*
- $\mathsf{chaincodes} \subset \mathbb{N}, \mathsf{ch} \in \mathbb{N}$ — $\{$*The set of chaincodes/chaincode IDs and current channel*
- $\mathsf{orderers}_{\mathsf{C}} \subseteq \mathsf{orderers}, \mathsf{peers}_{\mathsf{C}} \subseteq \mathsf{peers}, \forall \mathsf{C} \in \mathsf{clients}$ — $\{$*The orderers, resp. peers, each client knows.*
- $\mathtt{v_{ep}}, \mathsf{simulateTx}$ — $\{$*Algorithms check whether endorsement policies are fulfilled and simulate all chaincodes*

Description of $M_{\mathtt{init}}$:

Implemented role(s): $\{\mathtt{init}\}$
Subroutines: $\mathcal{F}_{\text{ro}} : \mathtt{randomOracle}$
Internal state:

- $\mathsf{pvHash} \in \{0,1\}^{\eta}, \mathsf{pvHash} = \bot$ — $\{$*The hash of* S_{init}

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

See Appendix B for notation details. We start index counting at 1.

Initialization:

send $(\mathsf{pid_{cur}}, (0, \bot, \mathsf{S_{init}}))$ **to** $(\mathsf{pid_{cur}}, \mathsf{sid_{cur}}, \mathcal{F}_{\text{ro}} : \mathtt{randomOracle})$ — $\{$*Generate hash of initial state instead of genesis block*
  wait for $(\mathsf{pid_{cur}}, h)$
$\mathsf{pvHash} \leftarrow h$

Main:

recv $\mathtt{InitClient}$ **from** $\mathtt{I/O}$ **s.t.** $\mathsf{pid_{cur}} \in \mathsf{clients}$**:** — $\{$*Client Initialization:*
  send responsively $(\mathtt{InitClient}, \mathsf{pid_{cur}})$ **to** $\mathtt{NET}$ — $\{$*Request initialization information from* $\mathcal{A}$
  wait for $(\mathtt{InitClient}, corrupted)$ **s.t.** $corrupted \in \{\mathtt{true}, \mathtt{false}\}$
  reply $(\mathtt{InitClient}, \mathsf{peers}_C, \mathsf{orderers}_C, \mathsf{chaincodes}, \mathsf{ch}, \mathtt{v_{ep}}, corrupted)$

recv $\mathtt{InitOrderer}$ **from** $\mathtt{I/O}$ **s.t.** $\mathsf{pid_{cur}} \in \mathsf{orderers}$**:** — $\{$*Orderer initialization*
  send responsively $(\mathtt{InitOrderer}, \mathsf{pid_{cur}})$ **to** $\mathtt{NET}$
  $\{$*Request initialization information from* $\mathcal{A}$*: reported Kafka leader and corruption status*
  wait for $(\mathtt{InitOrderer}, corrupted, pid_{pid_{KL}}, peers_{\mathsf{receiver}})$
    s.t. $corrupted \in \{\mathtt{true}, \mathtt{false}\}, pid_{pid_{KL}} \in \mathsf{kafka}, peers_{\mathsf{receiver}} \subset \mathsf{peers}$
  reply $(\mathtt{InitOrderer}, \mathsf{clients}, \mathsf{peers}, \mathsf{kafka}, \mathsf{ch}, \mathsf{pvHash}, pid_{pid_{KL}}, peers_{\mathsf{receiver}}, corrupted)$

recv $\mathtt{InitPeer}$ **from** $\mathtt{I/O}$ **s.t.** $\mathsf{pid_{cur}} \in \mathsf{peers}$**:** — $\{$*Peer initialization*
  send responsively $(\mathtt{InitPeer}, \mathsf{pid_{cur}})$ **to** $\mathtt{NET}$ — $\{$*Request initialization information from* $\mathcal{A}$
  wait for $(\mathtt{InitPeer}, corrupted)$ **s.t.** $corrupted \in \{\mathtt{true}, \mathtt{false}\}$
  reply $(\mathtt{InitPeer}, \mathsf{clients}, \mathsf{peers}, \mathsf{orderers}, \mathsf{kafka}, \mathsf{chaincodes}, \mathsf{ch}, \mathsf{pvHash}, \mathsf{S_{init}}, \mathtt{v_{ep}}, \mathsf{simulateTx}, corrupted)$

recv $\mathtt{InitKafka}$ **from** $\mathtt{I/O}$ **s.t.** $\mathsf{pid_{cur}} \in \mathsf{kafka}$**:** — $\{$*Kafka initialization*
  send responsively $(\mathtt{InitKafka}, \mathsf{pid_{cur}})$ **to** $\mathtt{NET}$ — $\{$*Request initialization information from* $\mathcal{A}$ *to get the initial Kafka leader*
  wait for $(\mathtt{InitKafka}, corrupted, pid_{pid_{KL}})$ **s.t.** $corrupted \in \{\mathtt{true}, \mathtt{false}\}, pid_{pid_{KL}} \in \mathsf{kafka}$
  reply $(\mathtt{InitKafka}, \mathsf{ch}, pid_{pid_{KL}}, \mathsf{orderers}, \mathsf{kafka}, corrupted)$

recv $\mathtt{InitJudge}$ **from** $\mathtt{I/O}$**:** — $\{$*Judge initialization*
  reply $(\mathtt{InitJudge}, \mathsf{clients}, \mathsf{orderers}, \mathsf{peers}, \mathsf{kafka}, \mathsf{ch}, \mathsf{S_{init}}, \mathsf{pvHash}, \mathsf{chaincodes}, \mathsf{simulateTx})$

Fig. 29: The initialization functionality $\mathcal{F}_{\text{init}}$ for the Fabric* model

Description of $\mathcal{F}_{\text{judge}} = (\texttt{judge})$:

Participating roles: $\{\texttt{judge}\}$
Corruption model: *incorruptible*
Protocol parameters:
- applyBlock *{deterministic polynomial time algorithm that updates the ledger state on input a block*

Description of $M_{\texttt{judge}}$:

Implemented role(s): $\{\texttt{judge}\}$
Subroutines: $\mathcal{F}_{\text{cert}} : \texttt{verifier}, \mathcal{F}_{\text{ro}} : \texttt{randomOracle}, \mathcal{F}_{\text{init}} : \texttt{init}$
Internal state:
- $\mathsf{clients}_\mathsf{J} \subseteq \{0,1\}^*, \mathsf{clients} = \emptyset$ *{The set of clients*
- $\mathsf{peers}_\mathsf{J} \subseteq \{0,1\}^*, \mathsf{peers} = \emptyset$ *{The set of peers*
- $\mathsf{orderers}_\mathsf{J} \subseteq \mathsf{orderers} \times \{0,1\}^*$ *{The set of orderers*
- $\mathsf{kafka}_\mathsf{J} \subseteq \mathsf{kafka} \times \{0,1\}^*$ *{The set of Kafka broker (identities)*
- $\mathsf{ch} \in \mathbb{N}, \mathsf{S}_{\mathsf{init}} \in \{0,1\}^*, pvHash \in \{0,1\}^\eta$ *{The current channel id, initial state, and its hash*
- $\mathsf{W}_{\mathfrak{B}} \subseteq \mathsf{peers} \times \mathsf{orderers} \times \mathbb{N} \times \{0,1\}^\eta \times \{0,1\}^* \times \{0,1\}^*, \mathsf{W}_{\mathfrak{B}} = \emptyset.$ *{The set of evidence blocks, including reporter, creator, block number, hash, and signature*
- $\mathsf{msgStream} \subseteq \mathbb{N} \times \mathsf{kafka} \times \{0,1\}^* \times \{0,1\}^*, \mathsf{msgStream} = \emptyset$ *{The collected Kafka messages including offset, leader, msg, signature*
- $\mathsf{S} \subseteq \mathsf{peers} \times \mathbb{N} \times \{0,1\}^*, \mathsf{S} = \emptyset$ *{The reported states by peer (peer, delivery number, state)*
- $\mathsf{verdicts} \in \{0,1\}^*, \mathsf{verdicts} = \varepsilon$ *{Recorded verdict*
- $\mathsf{chaincodes} \subset \mathbb{N}, \mathsf{ch} \in \mathbb{N}$ *{The set of known chaincodes and the current channel id*
- simulateTx *{Algorithm to execute all chaincodes*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

See Appendix B for notation details. We start index counting at 1.

Initialization:
send $\texttt{InitJudge}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init}} : \texttt{init})$
wait for $(\texttt{InitJudge}, clients, orderers, peers, kafka, ch, S_{init}, pvHash, chaincodes, simulateTx)$
$\mathsf{peers}_\mathsf{J} \leftarrow peers; \mathsf{orderers}_\mathsf{J} \leftarrow orderers; \mathsf{kafka}_\mathsf{J} \leftarrow kafka; \mathsf{ch} \leftarrow channel; \mathsf{S}_{\mathsf{init}} \leftarrow S_{init}, \mathsf{pvHash} \leftarrow pvHash, \mathsf{chaincodes} \leftarrow chaincodes, \mathsf{simulateTx} \leftarrow simulateTx$

MessagePreprocessing:
if $\mathsf{verdicts} \neq \varepsilon$**:** *{If there is a verdict, $\mathcal{F}_{\text{judge}}$ does not produce further verdicts*
reply $(\texttt{ack})$ *{Construction to ensure direct execution of the next step*

Fig. 30: The judging functionality $\mathcal{F}_{\text{judge}}$ for the Fabric* model (Part 1)

Description of $M_{\mathtt{judge}}$ (cont.):

Main:

recv $(\mathtt{Evidence}, msg)$ **from** $\mathtt{I/O}$ **s.t.** $\mathsf{pid}_{\mathsf{call}} \in \mathsf{peers} \wedge role = \mathsf{peer} \wedge \mathsf{sid}_{\mathsf{call}} = \mathsf{sid}_{\mathsf{cur}}$**:** *{Process (block) evidence from peers*

  send responsively $(\mathtt{Evidence}, msg)$ **to** $\mathtt{NET}$ *{Leak data to $\mathcal{A}$ - because we want to model a public judge*

  wait for $\mathtt{ack}$

  if $msg \neq \langle blockNum, pvHash, B, pid_o, \sigma \rangle$, **s.t.** $blockNum \in \mathbb{N}, pvHash \in \{0,1\}^{\eta}, \mathtt{extCh}(B) = \mathsf{ch}$ and $B, pid_o, \sigma \in \{0,1\}^*$**:**

    $\mathsf{verdicts.add}(\mathsf{dis}((\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})))$ *{Peers have to report well-formed evidence*

  else:

    $chain \leftarrow \{\overline{B} | (\mathsf{pid}_{\mathsf{call}}, \cdot, \cdot, \cdot, \overline{B}, \cdot) \in \mathsf{W}_{\mathfrak{B}}\}$ *{Reported chain of $\mathsf{pid}_{\mathsf{call}}$*

    if $\mathsf{v}_{\mathfrak{B}}(B) = \mathtt{false}$**:** *{Orderers should produce well-formed blocks, peers should not accept malformed block*

      $\mathsf{verdicts.add}(\mathsf{dis}((\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})))$

    else if $\mathtt{extMaxMsgId}(chain) \neq \mathtt{extMinMsgId}(B) - 1$**:** *{$\mathtt{extMaxMsgId}(\emptyset) = 0$; The lowest message ID in the block body needs to be 1. Blocks, resp. the chain needs to be a the sequence of consecutive Kafka messages*

      $\mathsf{verdicts.add}(\mathsf{dis}((\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})))$

    else if $pid_o \notin \mathsf{orderers}$**:**

      $\mathsf{verdicts.add}(\mathsf{dis}((\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})))$ *{Peers should only accept blocks from orderers*

    else:

      send $(\mathtt{Verify}, \langle blockNum, pvHash, B, pid_o \rangle, \sigma)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (pid_o, \mathsf{sid}_{\mathsf{cur}}, \mathtt{orderer}), \mathcal{F}_{\mathrm{cert}} : \mathtt{verifier})$ *{Check whether signature is valid*

      wait for $(\mathtt{Signature}, b)$

      if $\neg b$**:** *{Honest peers report valid signatures*

        $\mathsf{verdicts.add}(\mathsf{dis}((\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})))$

      else:

        parse $((n, pid_K^n, msg_n, mr_n, mp_n, \sigma_n), \ldots, (n+m, pid_K^{n+m}, msg_{n+m}, mr_{n+m}, mp_{n+m}, \sigma_{n+m}))$ **from** B

        $check \leftarrow \mathtt{true}$

        for $i = 0$ to m **do:** *{Check whether Kafka messages are protected by a Merkle root and a signature*

          $v \leftarrow \mathtt{verifyMerkleProof}(\langle n+i, pid_K^{n+i}, msg_{n+i} \rangle, mr_{n+i}, mp_{n+i})$ *{Check whether $n+i, pid_K^{n+i}, msg_{n+i}$ was part of the Merkle tree which root was signed*

          if $v \wedge pid_K^{n+i} \in \mathsf{kafka}$**:**

            send $(\mathtt{Verify}, \langle mr_{n+1} \rangle, \sigma_{n+i})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (pid_K^{n+i}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{kafka}), \mathcal{F}_{\mathrm{cert}} : \mathtt{verifier})$

            wait for $(\mathtt{Signature}, a)$

            $check \leftarrow check \wedge a$

            if a**:**

              $\mathsf{msgStream} \leftarrow \mathsf{msgStream} \cup \{(n+i, pid_K^{n+i}, msg_{n+i}, \sigma_{n+i})\}$ *{Record correctly signed messages from the message stream*

          else:

            $check \leftarrow \mathtt{false}$

        if $\nexists (\mathsf{pid}_{\mathsf{call}}, \cdot, blockNum - 1, \cdot, \cdot, \cdot) \in \mathsf{W}_{\mathfrak{B}} \wedge blockNum \neq 1$**:** *{Honest peers/orderers report consecutive blocks starting at 1 and do not report wrong signatures*

          $\mathsf{verdicts.add}(\mathsf{dis}((\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})))$

        else if $\neg check$**:**

          $\mathsf{verdicts.add}(\mathsf{dis}((\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})))$

          {Orderers should build blocks from correct message streams and peers should not accept such blocks

        else:

          $\mathsf{W}_{\mathfrak{B}}.\mathsf{add}(\mathsf{pid}_{\mathsf{call}}, pid_o, blockNum, pvHash, B, \sigma)$ *{Record block that fullfills basic correctness*

          if $\exists (pid_{P,1}, pid_{o,1}, i, pvHash_1, B_1, \sigma_1), (pid_{P,2}, pid_{o,2}, i, pvHash_2, B_2, \sigma_2) \in \mathsf{W}_{\mathfrak{B}}$,
          s.t. $B_1 \neq B_2 \wedge (pid_{o,1}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{orderer}), (pid_{o,2}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{orderer}) \notin \mathsf{verdicts} \wedge$
          $\nexists pid_K \in \mathsf{kafka}_{\mathsf{judge}}$, **s.t.** $(pid_K, \mathsf{sid}_{\mathsf{cur}}, \mathtt{kafka}) \in \mathsf{verdicts}$**:**

            $\mathtt{analyzeConflBlocks}[(pid_{P,1}, pid_{o,1}, i, pvHash_1, B_1, \sigma_1), (pid_{P,2}, pid_{o,2}, i, pvHash_2, B_2, \sigma_2)]$

            {See $\mathtt{analyzeConflBlocks}$ specification for details

            reply $\mathtt{ack}$ *{Construction to ensure direct execution of the next step*

Fig. 31: The judging functionality $\mathcal{F}_{\mathrm{judge}}$ for the Fabric* model (Part 1)

Description of $M_{\mathtt{judge}}$ (cont.):

Main (continued):

recv (Evidence, $(pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset, \sigma_P)$) **from** I/O
s.t. $\mathsf{pid}_{\mathsf{call}} \in \mathsf{clients} \wedge role = \mathtt{client} \wedge \mathsf{sid}_{\mathsf{call}} = \mathsf{sid}_{\mathsf{cur}} \wedge$
$queryToBeProcessed = (queryID, seqNum_C, \mathsf{pid}_{\mathsf{call}}, ch, chaincodeId, query, \sigma_C) \wedge$
$pid_P \in \mathsf{peers}_J$: {Process evidence from clients
send (Verify, $(queryID, seqNum_C, \mathsf{pid}_{\mathsf{cur}}, \mathsf{ch}, chaincodeId, query), \sigma_C$) **to** $(\mathsf{pid}_{\mathsf{cur}}, (pid_C, \mathsf{sid}_{\mathsf{cur}}, \mathtt{client}), \mathcal{F}_{\mathrm{cert}} : \mathtt{verifier})$
wait for (VerResult, b_1) {Check correctness of request
send (Verify, (Evidence, $(pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset, \sigma_P)$))
to $(\mathsf{pid}_{\mathsf{cur}}, (pid_P, \mathsf{sid}_{\mathsf{cur}}, \mathtt{peer}), \mathcal{F}_{\mathrm{cert}} : \mathtt{verifier})$
wait for (VerResult, b_2) {Check correcly signed endorsement
if $b_1 \wedge b_2$:
send responsively (Evidence, $(pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset, \sigma_P)$)
to NET
wait for ack {Leak data to $\mathcal{A}$ - because we want to model a public judge
if $chaincodeId = 0$: {Peer should output a prefix of the message list
Let $msgStream = \{(i, tx) \mid (i, _, tx, _) \in \mathsf{msgStream}\}$ {Remove Kafka broker ID and signature
if $resp$ is not a prefix of $msgStream$:
verdicts.add(dis($pid_P, \mathsf{sid}_{\mathsf{cur}}, \mathtt{peer}$)) {Peer deviated from consensus result
else: {Otherwise check whether $resp$, $readset$, and $writeset$ are consistent according to simulateTx
Check whether $resp$ **can be parsed** $(ptr, resp') \wedge ptr \leq |\mathsf{msgStream}| \wedge readset$ **is valid:** Generate blocks according to the Fabric specification from msgStream up to ptr (cf. Figure 33) and generate the world state afterward (cf. Figure 35). Check whether the $readset$ can be found in the world state.
if check above fails: {Peer used an readset which is not based on the msgStream
verdicts.add(dis($pid_P, \mathsf{sid}_{\mathsf{cur}}, \mathtt{peer}$)) {Peer used readset which did not match the state of the blockchain
else:
if simulateTx($(pid_P, queryID, seqNum, chaincodeId, queryToBeProcessed, resp, readset, writeset, \sigma_P)) \neq resp$:
{We assume that we can directly use simulateTx based on the data of the endorsement,
verdicts.add(dis($pid_P, \mathsf{sid}_{\mathsf{cur}}, \mathtt{peer}$)) {Peer did not provide deterministically computed response
reply ack

recv GetVerdict **from** I/O:
reply (GetVerdict, verdicts)

recv GetJudicialReport **from** I/O:
Let $msgStream = \{(i, tx) \mid (i, _, tx, _) \in \mathsf{msgStream}\}$ {Remove Kafka broker ID and signature
reply (GetJudicialReport, $msgStream$) {Output the maximal message list seen so far as judicial report

Procedures and Functions:

function analyzeConflBlocks ($[(pid_{P,1}, pid_{o,1}, i, pvHash^1, B_i^1, \sigma_i^1), (pid_{P,2}, pid_{o,2}, i, pvHash^2, B_i^2, \sigma_i^2)]$):
if $\exists (n, \widehat{pid}^1_K, \widehat{msg}^1, \widehat{mr}^1, \widehat{mp}^1, \hat{\sigma}^1) \neq (n, \widehat{pid}^2_K, \widehat{msg}^2, \widehat{mr}^2, \widehat{mp}^2, \hat{\sigma}^2)$
s.t. $(n, \widehat{pid}^1_K, \widehat{msg}^1, \widehat{mr}^1, \widehat{mp}^1, \hat{\sigma}^1) \in B_i^1, (n, \widehat{pid}^2_K, \widehat{msg}^2, \widehat{mr}^2, \widehat{mp}^2, \hat{\sigma}^2) \in B_i^2$:
for all $pid_K \in \mathsf{kafka}_J$ **do:**
verdicts.add(dis($(pid_K, \mathsf{sid}_{\mathsf{cur}}, \mathtt{peer})$)) {Kafka cluster should send unambiguous messages per offset
else:{Check whether B_1 and B_2 were generated honestly, maxTx denotes the maximum of transactions in a block as specified in cutBlock
if $\exists (n, pid_K, msg, mr, mp, \sigma) \in B_1$, **s.t.** $msg = \langle \mathtt{TTC}, \mathsf{ch}, i, \overline{pid_o} \rangle \ \wedge$
$(\nexists (m, pid'_K, msg', mr', mp', \sigma') \in B_1$, **s.t.**
$m < n \wedge msg' = \langle \mathtt{TTC}, \mathsf{ch}, i, \overline{pid_o}' \rangle) \wedge |B_1| < \mathtt{maxTx}$: {Cut was created based on cut block message
if $\exists (n+1, pid^*_K, msg^*, mr^*, mp^*, \sigma^*) \in B_1$: {Orderer did not use the first cut message to create the block
verdicts.add($pid_{o,1}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{orderer}$)

else if $|B_1| > \mathtt{maxTx}$: {maxTx denotes the maximum number of transactions in a block as specified in cutBlock
verdicts.add($pid_{o,1}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{orderer}$)
else if $\exists (n, pid_K, msg, mr, mp, \sigma) \in B_2$ **s.t.** $msg = \langle \mathtt{TTC}, \mathsf{ch}, i, \overline{pid_o} \rangle \ \wedge \ (\nexists (m, pid'_K, msg', mr', mp', \sigma') \in B_2$,
s.t. $m < n \wedge msg' = \langle \mathtt{TTC}, \mathsf{ch}, i, \overline{pid_o}' \rangle) \wedge |B_2| < \mathtt{maxTx}$: {Cut was created based on cut block message
if $\exists (n+1, pid^*_K, msg^*, mr^*, mp^*, \sigma^*) \in B_2$: {Orderer did not use the first cut message to create the block
verdicts.add($pid_{o,2}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{orderer}$)

else if $|B_2| > \mathtt{maxTx}$: {maxTx denotes the maximum number of transactions in a block as specified in cutBlock
verdicts.add($pid_{o,2}, \mathsf{sid}_{\mathsf{cur}}, \mathtt{orderer}$)

Fig. 32: The judging functionality $\mathcal{F}_{\mathrm{judge}}$ for the Fabric* model (Part 3)

Description of $\mathcal{P}^{\mathsf{FAB}^*}_{\texttt{orderer}} = (\texttt{orderer})$:

Participating roles: $\{\texttt{orderer}\}$
Corruption model: *Dynamic corruption without secure erasures*
Protocol parameters:
- $\eta \in \mathbb{N}$ {*The security parameter*

Description of $M_{\texttt{orderer}}$:

Implemented role(s): $\{\texttt{init}\}$
Subroutines: $\mathcal{F}_{\text{cert}} : \texttt{signer}, \mathcal{F}_{\text{cert}} : \texttt{verifier}, \mathcal{F}_{\text{ro}} : \texttt{randomOracle}$
Internal state:
- $\mathsf{corrupted}_{\mathsf{init}} \in \{\texttt{true}, \texttt{false}\}, \mathsf{corrupted}_{\mathsf{init}} = \texttt{false}$ {*Record whether instance was corrupted on initialization*
- $\mathsf{clients}_{\mathsf{O}} \subseteq \{0,1\}^*, \mathsf{clients}_{\mathsf{O}} = \emptyset$ {*The set of clients*
- $\mathsf{peers}_{\mathsf{O}} \subseteq \mathsf{peers} \times \{0,1\}^*, \mathsf{peers}_{\mathsf{O}} = \emptyset$ {*The set of peers*
- $\mathsf{kafka}_{\mathsf{O}} \subseteq \mathsf{kafka} \times \{0,1\}^*, \mathsf{kafka}_{\mathsf{O}} = \emptyset$ {*The set of Kafka brokers*
- $\mathsf{ch} \in \mathbb{N}, \mathsf{ch} = 0$ {*The channel ID*
- $\mathsf{pvHash} \in \{0,1\}^\eta, \mathsf{pvHash} = \varepsilon$ {*The previous blocks hash*
- $\mathsf{chain} \in \mathfrak{C}$[a]$\mathsf{chain} = \varepsilon, \mathsf{blockNum} \in \mathbb{N}, \mathsf{blockNum} = 0$ {*The blockchain and the "current" block number*
- $\mathsf{submittedTx} \subset \{0,1\}^*, \mathsf{submittedTx} = \emptyset$ {*The set of known transactions*
- $\mathsf{msgStream} \subseteq \mathbb{N} \times \mathsf{kafka} \times \{0,1\}^* \times (\{0,1\}^* \cup \{\bot\})^3, \mathsf{msgStream} = \emptyset$ {*The ordered stream of transactions of the form (message ID, leader, message, Merkle root, path for Merkle proof, signature)*
- $\mathsf{pid}_{\mathsf{KL}} \in \mathsf{kafka}, \mathsf{pid}_{\mathsf{KL}} = \varepsilon$ {*The current Kafka leader*
- $\mathsf{peers}_{\mathsf{receiver}} \subseteq \mathsf{peers}_{\mathsf{O}}, \mathsf{peers}_{\mathsf{receiver}} = \emptyset$ {*The set of peers connected to the orderer*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid and $role = \texttt{orderer}$.

Corruption behavior:
DetermineCorrStatus$(pid, sid, role)$:
if $\mathsf{corrupted} = \texttt{true} \vee \mathsf{corrupted}_{\mathsf{init}} = \texttt{true}$**:** {*Checks whether node itself is corrupted.*
  return $\texttt{true}$
$corrRes \leftarrow \mathbf{corr}(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \texttt{signer})$ {*Request corruption status at* $\mathcal{F}_{\text{cert}}$
if $corrRes = \texttt{true}$**:** {*Checks whether* $\mathcal{F}_{\text{cert}}$ *instance is corrupted*
  return $\texttt{true}$

See Appendix B for notation details. We start index counting at 1.

Initialization:
send $\texttt{InitOrderer}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{init}} : \texttt{init})$
wait for $(\texttt{InitOrderer}, clients, peers, kafka, ch, pvHash, pid_{KL}, peers_{\mathsf{receiver}}, corrupted)$
$\mathsf{clients}_{\mathsf{O}} \leftarrow clients, \mathsf{peers}_{\mathsf{O}} \leftarrow peers; \mathsf{orderers}_{O} \leftarrow orderers; \mathsf{kafka}_{\mathsf{O}} \leftarrow kafka; \mathsf{ch} \leftarrow channel$
$\mathsf{pvHash} \leftarrow pvHash; \mathsf{peers}_{\mathsf{receiver}} \leftarrow peers_{\mathsf{receiver}}; \mathsf{corrupted}_{\mathsf{init}} \leftarrow corrupted$

[a] $\mathfrak{C}$ contains well-formed (block)chains according to the definition of Fabric*, i. e., consecutive sequences of blocks, $chain = (\mathcal{B}_0, \mathcal{B}_1, \ldots)$, including $\mathsf{chain} = \varepsilon$.

Fig. 33: Specification of the Fabric* orderer $\mathcal{P}^{\mathsf{FAB}^*}_{\texttt{orderer}}$ (Part 1)

Description of $M_{\texttt{orderer}}$ (cont.):

Main:

recv $(\texttt{Commit}, T_{txId})$ **from** NET: {*Forward commit requests to Kafka*
  if $T_{txId} \notin \mathsf{submittedTx}$:
    $\mathsf{submittedTx.add}(T_{txId})$
    send $(\texttt{Commit}, T_{txId}, \mathsf{pid}_{\mathsf{cur}})$ **to** NET {*Forward commit request to Kafka leader*

recv $(\texttt{Deliver}, \mathsf{pid}_{\mathsf{cur}}, \mathsf{ch}, seqNum, msgStream')$ **from** NET
  s.t. $seqNum = \texttt{extractMessageId}(\mathsf{msgStream})$,[a] $\texttt{chkKafkaDelivery}$[b]$(msgStream', seqNum) = \texttt{true}$:
  {*Delivery of messages and block dispatching. We set* $\texttt{extractMessageId}(\emptyset) := 1$
  $((n, pid_K^n, msg_n, mr_n, mp_n, \sigma_n), \ldots, (n+m, pid_K^{n+m}, msg_{n+m}, mr_{n+m}, mp_{n+m}, \sigma_{n+m})) \leftarrow msgStream'$ $(*)$
  {*Extract messages from the message stream segment*
  $check \leftarrow \texttt{true}$
  for $i = 0$ to m **do**:
    if $pid_K^{n+i} \in \mathsf{kafka}$: {*Messages need to be ordered by Kafka brokers*
      $v \leftarrow \texttt{verifyMerkleProof}$[c]$(\langle n+i, pid_K^{n+i}, msg_{n+i}\rangle, mr_{n+i}, mp_{n+i})$ {*Check Merkle proof*
      if v:
        send $(\texttt{Verify}, mr_{n+i}, \sigma_{n+i})$ **to** $(\mathsf{sid}_{\mathsf{cur}}, (pid_K^{n+1}, \mathsf{sid}_{\mathsf{cur}}, \texttt{kafka}), \mathcal{F}_{\mathrm{cert}} : \texttt{signer})$
        wait for b
        $check \leftarrow check \vee b$
      else:
        $check \leftarrow check \vee v$
    else:
      $check \leftarrow \texttt{false}$
  if $check$: {*The received message stream segment is considered valid*
    $\mathsf{msgStream.add}(msgStream')$; $chain' \leftarrow \texttt{cutBlock}(\mathsf{msgStream})$ {*Start generating and dispatching blocks*
    $B_1, \ldots B_d \leftarrow \texttt{extractBody}(\mathsf{chain})$[d]; $B_1, \ldots, B_d, B_{d+1}, \ldots, B_{d+g} \leftarrow chain'$; $\overline{msg} \leftarrow \varepsilon$
    {*Generate message to "broadcast" blocks*
    for $i = 1$ to g **do**: {*Generate block related data and dispatch block-wise*
      $\mathsf{blockNum} \leftarrow \mathsf{blockNum} + 1$; $\mathsf{chain.add}(\mathsf{blockNum}, \mathsf{pvHash}, B_{d+i})$
      send $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{blockNum}, \mathsf{pvHash}, B_{d+i}))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ro}} : \texttt{randomOracle})$ {*Generate hash for block*
      wait for $(\mathsf{pid}_{\mathsf{cur}}, h)$
      $\mathsf{pvHash} \leftarrow h$
      send $(\texttt{Sign}, (\mathsf{blockNum}, \mathsf{pvHash}, B_{d+i}, \mathsf{pid}_{\mathsf{cur}}))$ **to** $(\mathsf{sid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \mathcal{F}_{\mathrm{cert}} : \texttt{signer})$ {*Sign block*
      wait for $(\texttt{Signature}, \sigma)$
      $\mathsf{chain.add}(\mathsf{blockNum}, \mathsf{pvHash}, B_{d+i})$; $msg \leftarrow (\texttt{Deliver}, \mathsf{blockNum}, \mathsf{pvHash}, B_{d+i}, \mathsf{pid}_{\mathsf{cur}}, \sigma)$
      for all $pid_P \in \mathsf{peers}_{\mathsf{receiver}}$ **do**: {*Send block to peers*
        $\overline{msg}.\mathsf{add}(pid_P, msg,)$
    send $\overline{msg}$ **to** NET {*Adversary is responsible for broadcasting*

recv $(\texttt{SetLeader}, pid_{KL})$ **from** NET **s.t.** $pid_{KL} \in \mathsf{kafka}$: {*Adversary defines which broker to be the leader*
  $\mathsf{pid}_{\mathsf{KL}} \leftarrow pid_{KL}$
  reply ack

recv TTC **from** NET: {*Cut block/TTC message as specified in Fabric's Kafka-based ordering service*
  reply $(\mathsf{pid}_{\mathsf{KL}}, (\texttt{TTC}, \mathsf{ch}, \mathsf{blockNum}, \mathsf{pid}_{\mathsf{cur}}))$

recv Pull **from** NET: {*Pull request at Kafka leader to generate a block afterwards*
  $seqNum \leftarrow \texttt{extractMessageId}(\mathsf{msgStream})$[e] {*Extract message ID from locally stored message stream*
  reply $(\texttt{Pull}, \mathsf{pid}_{\mathsf{cur}}, \mathsf{ch}, seqNum)$

recv $(\texttt{setBlockRec}, peers_{\mathsf{receiver}})$ **from** NET **s.t.** $peers_{\mathsf{receiver}} \subseteq \mathsf{peers}_{\mathsf{O}}$:
  {*The adversary is allowed to update the peers that receive the blocks from* $\mathsf{pid}_{\mathsf{cur}}$
  $\mathsf{peers}_{\mathsf{receiver}} \leftarrow peers_{\mathsf{receiver}}$
  reply ack

[a] The algorithm `extractMessageId` outputs the current message ID. On input a message stream msgStream it outputs the highest message ID in msgStream.

[b] `chkKafkaDelivery` verifies that K' has the correct form (cf. $(*)$ below the `Deliver` call) and that it is a consecutive sequence of messages starting at $seqNum$.

[c] `verifyMerkleProof` gets as input a message, a Merkle root, and Merkle proof. It outputs `true` or `false`. Note that it is possible to evaluate a Merkle proof in $O(n)$ where n is the size of the Merkle proof. For a detailed specification, we refer to [59].

[d] Let $\mathcal{B} = (blockNum, pvHash, B, pid, \sigma)$ be a block according to the Fabric* definition. $\texttt{extractBody}(\mathcal{B})$ outputs B

[e] The algorithm `extractMessageId` outputs the current message ID. On input a message stream msgStream it outputs the highest message ID in msgStream.

Fig. 34: Specification of the Fabric* orderer $\mathcal{P}^{\mathsf{FAB}^*}_{\texttt{orderer}}$ (Part 2)

Description of the protocol $\mathcal{P}^{\mathsf{FAB}^*}_{\mathtt{peer}} = (\mathtt{peer})$:

Participating roles: $\{\mathtt{peer}\}$
Corruption model: *Dynamic corruption without secure erasures*
Protocol parameters:
- applyBlock {*deterministic polynomial time algorithm that updates the ledger state on input a block*

Description of $M_{\mathtt{peer}}$:

Implemented role(s): $\{\mathtt{peer}\}$
Subroutines: $\mathcal{F}_{\mathrm{cert}} : \mathtt{signer}, \mathcal{F}_{\mathrm{cert}} : \mathtt{verifier}, \mathcal{F}_{\mathrm{ro}} : \mathtt{randomOracle}, \mathcal{F}_{\mathrm{init}} : \mathtt{init}$
Internal state:
- $\mathsf{corrupted}_{\mathsf{init}} \in \{\mathtt{true}, \mathtt{false}\}, \mathsf{corrupted}_{\mathsf{init}} = \mathtt{false}$ {*Record whether instance was corrupted on initialization*
- $\mathsf{clients}_{\mathsf{P}} \subseteq \{0,1\}^*$ {*The set of clients*
- $\mathsf{peers}_{\mathsf{P}} \subseteq \{0,1\}^*$ {*The set of peers*
- $\mathsf{orderers}_{\mathsf{P}} \subseteq \{0,1\}^*$ {*The set of orderers*
- $\mathsf{kafka}_{\mathsf{P}} \subseteq \{0,1\}^*$ {*The set of Kafka broker (identities)*
- $\mathsf{seqNum} \in \mathbb{N}$, initially $\mathsf{seqNum} = 0$ {*Sequence number, replacement for the timestamp*
- $\mathsf{chaincodes} \subset \mathbb{N}, \mathsf{ch} \in \mathbb{N}$ {*The set of known chaincodes and the current channel id*
- $\mathsf{chain} \in \mathfrak{C}, \mathsf{blockNum} \in \mathbb{N}$, initially $\mathsf{chain} = \bot, \mathsf{blockNum} = 0$ {*The blockchain and the "current" block number*
- $\mathsf{pvHash} \in \{0,1\}^\eta, \mathsf{S} \in \{0,1\}^*$ {*The previous blocks hash and the state based on the chain*
- v_{ep},[a] simulateTx[b] {*Algorithms to endorsement policies and to simulate all chaincodes*
- $\mathsf{txValidity} \subseteq \mathbb{N} \times \{\mathtt{true}, \mathtt{false}\}$, initially $\mathsf{txValidity} = \emptyset$ {*Stores whether* $\mathsf{pid}_{\mathsf{cur}}$ *considers transactions as valid*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid and $role = \mathtt{peer}$.

Corruption behavior:
DetermineCorrStatus$(pid, sid, role)$:
if $\mathtt{corrupted} = \mathtt{true} \vee \mathsf{corrupted}_{\mathsf{init}} = \mathtt{true}$**:** {*Checks whether node itself is corrupted.*
  return true
$corrRes \leftarrow \mathbf{corr}(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \mathtt{signer})$ {*Request corruption status at* $\mathcal{F}_{\mathrm{cert}}$
if $corrRes = \mathtt{true}$**:** {*Checks whether* $\mathcal{F}_{\mathrm{cert}}$ *instance is corrupted*
  return true

See Appendix B for notation details. We start index counting at 1.

Initialization:
send InitPeer **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{init}} : \mathtt{init})$ {*Request initialization at* $\mathcal{F}_{\mathrm{init}}$
wait for $(\mathtt{InitPeer}, clients_P, peers_P, orderers_P, kafka_P, chaincodes, channel, prevhash, chainState, v_{ep}, simulateTx, corrupted)$
$\mathsf{clients}_{\mathsf{P}} \leftarrow clients_P; \mathsf{peers}_{\mathsf{P}} \leftarrow peers_C; \mathsf{orderers}_{\mathsf{P}} \leftarrow orderers_C; \mathsf{kafka}_{\mathsf{P}} \leftarrow kafka_P; \mathsf{chaincodes} \leftarrow chaincodes$
$\mathsf{ch} \leftarrow channel; \mathsf{pvHash} \leftarrow prevhash; \mathsf{S} \leftarrow chainState; \mathsf{v}_{\mathsf{ep}} \leftarrow v_{ep}; \mathsf{simulateTx} \leftarrow simulateTx; \mathsf{corrupted}_{\mathsf{init}} \leftarrow corrupted$

[a] Note that v_{ep} has two duties: *(i)* v_{ep} allows clients to verify whether a planned transaction commit should be accepted without having access to a state, *(ii)* v_{ep} allows peers to verify whether they accept transactions during endorsement generation and state generation.
[b] simulateTx should include all chaincodes. It should output $readset$ and $writeset$.

Fig. 35: Specification of the Fabric* peer $\mathcal{P}^{\mathsf{FAB}^*}_{\mathtt{peer}}$ (Part 1)

Description of $M_{\texttt{peer}}$ (cont.):

Main:

```
recv (Deliver, blockNum_B, pvHash, B, pid_O, σ) from NET s.t. pid_O ∈ orderers, blockNum_B = blockNum + 1:   {Process blocks
  if v_𝔅(B)^a = true ∧ extCh(B)^b = ch ∧ extMaxMsgId(chain)^c = extMinMsgId(B)^d − 1:
                {Basic checks whether the new block is valid and whether it validly extends the chain
    send (Verify, ⟨blockNum_B, pvHash, B⟩, σ) to (pid_cur, (pid_O, sid_cur, orderer), F_cert : verifier)   {Check orderer signature
    wait for (Signature, b)
    if b:
      ((n, pid_K^n, msg_n, mr_n, mp_n, σ_n), ..., (n + m, pid_K^{n+m}, msg_{n+m}, mr_{n+m}, mp_{n+m}, σ_{n+m})) ← B   {Disassemble B into single messages
      check ← true; txValidity ← ∅, S ← S
      for i = 0 to m do:
        v ← verifyMerkleProof^e(⟨n + i, pid_K^{n+i}, msg_{n+i}⟩, mr_{n+i}, mp_{n+i})   {Check whether n + i, pid_K^{n+i}, msg_{n+i} was part of the Merkle tree which root was signed
        if v ∧ pid_K^{n+i} ∈ kafka:
          send (Verify, pk(pid_K^{n+i}), mr_{n+i}, σ_{n+i}) to (pid_cur, sid_cur, F_cert : verifier)
          wait for b
          check ← check ∨ b
          if verifyTxCommitment(msg_{n+i})^f:   {Check whether msg_{n+i} can be parsed as transaction commitment
            (txId, seqNum_C, pid_C, ch, chaincodeId, txPayload, σ) ← extractProp^g(msg_{n+i})
                                                   {Extract the tx proposal from msg_{n+i}
            tx ← (txId, seqNum_C, pid_C, ch, chaincodeId, txPayload)
            send (Verify, tx, σ) to (pid_cur, (pid_C, sid_cur, client), F_cert : verifier)
            wait for (Signature, d)                                          {Check signature validity
            txValidity.add(n + i, d ∧ (pid_C ∈ clients) ∧ v_ep(msg_{n+i}) ∧ applicableTx(S, msg_{n+i})^h)
                    {Mark tx as valid if signature is valid, pid_C is a client identity, endorsing
                     policy is fulfilled, and the transaction fits into the current state.
            S.add(msg_{n+i})
          else:
            txValidity.add(n + i, false)
        else:
          check ← false
      if check:                                                   {The received block is considered valid
        send (Evidence, blockNum_B, pvHash, B, pid_O, σ) to (public, sid_cur, F_judge : judge)
                                                   {Forward an accepted block to the judge judge
        wait for ack
        chain.add(blockNum_B, pvHash, B); blockNum ← blockNum + 1      {Apply updates to internal state
        S.applyBlock(B);txValidity.add(txValidity)
        send (pid_cur, (blockNum_B, pvHash, B)) to (pid_cur, sid_cur, F_ro : randomOracle)
                                                   {PrevHash has as input the blockheader of a block
        wait for (pid_cur, h); pvHash ← h
```

[a] $\mathsf{v}_{\mathfrak{B}}$ checks whether a block B is well-formed according to the Fabric* definition.

[b] `extCh` extracts the block's channel from B.

[c] `extMaxMsgId` extracts the maximal transaction counter from chain.

[d] `extMinMsgId` extracts the smallest transaction counter from a block B.

[e] `verifyMerkleProof` gets as input a message, a Merkle root, and Merkle proof. It outputs `true` or `false`.

[f] `verifyTxCommitment` output `true` if a transaction commitment is well-formed, otherwise `false`.

[g] According to the specification, msg_{n+i} should be a transaction commitment (which starts with the proposal). If the first part of msg_{n+i} does not fit the structure of a proposal, `extractProp`(msg_{n+i}) outputs $\perp$.

[h] On input a state S and a transaction commitment tx, the function `applicableTx` outputs `true` if tx is applicable to S, i.e., the readset from the endorsements in tx are equal and can be extracted from S. Otherwise, `applicableTx` outputs `false`.

Fig. 36: Specification of the Fabric* peer $\mathcal{P}_{\texttt{peer}}^{\mathsf{FAB}^*}$ (Part 2)

Description of $M_{\texttt{peer}}$ (cont.):

Main:

recv $(\texttt{Propose}, \mathit{txId}, \mathit{pid}_C, \mathit{seqNum}_C, \mathit{ch}, \mathit{chaincodeId}, \mathit{txPayload}, \sigma)$ **from** NET
  s.t. $\mathit{pid}_C \in \mathsf{clients}, \mathit{chaincodeId} \in \mathsf{chaincodes}, \mathit{ch} = \mathsf{ch}$: *{Tx proposal*
 send $(\texttt{Verify}, \langle \mathit{txId}, \mathit{pid}_C, \mathit{seqNum}_C, \mathit{ch}, \mathit{chaincodeId}, \mathit{txPayload}\rangle, \sigma)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathit{pid}_C, \mathsf{sid}_{\mathsf{cur}}, \texttt{client}), \mathcal{F}_{\text{cert}} : \texttt{verifier})$ *{Verify signature of proposal*
 wait for $(\texttt{Signature}, b)$
 if b:
  $\mathit{proposal} \leftarrow (\mathit{txId}, \mathit{seqNum}_C, \mathit{pid}_C, \mathit{ch}, \mathit{chaincodeId}, \mathit{txPayload}, \sigma)$
  $\mathit{executableProposal} \leftarrow (\mathit{txId}, \mathit{seqNum}_C, \mathit{pid}_C, \mathit{ch}, \mathit{chaincodeId}, \mathit{txPayload})$
  if $\mathsf{v}_{\mathsf{ep}}(S, \mathit{proposal}) = \texttt{true}$: *{Check whether proposal matches the endorsing policy*
   $(\mathit{resp}, \mathit{readset}, \mathit{writeset}) \leftarrow \mathsf{simulateTx}(S, \mathit{executableProposal})$ *{"Execute" transaction proposal*
   $\mathit{txEnd} \leftarrow (\mathsf{pid}_{\mathsf{cur}}, \mathit{txId}, \mathsf{seqNum}_P, \mathit{chaincodeId}, \mathit{proposal}, \mathit{resp}, \mathit{readset}, \mathit{writeset})$ *{Generate transaction endorsement*
   $\mathsf{seqNum}_P \leftarrow \mathsf{seqNum}_P + 1$
   send $(\texttt{Sign}, \mathit{txEnd})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \mathcal{F}_{\text{cert}} : \texttt{signer})$
   wait for $(\texttt{Signature}, \sigma_{\mathit{txEnd}})$
   $\mathit{msg} \leftarrow (\texttt{Endorsed}, \mathit{txEnd}, \sigma_{\mathit{txEnd}})$ *{Construct response*
  else:
   $\mathit{msg} \leftarrow (\texttt{Invalid}, \mathit{txId}, \texttt{Rejected})$
  send $(\mathit{pid}_C, \mathit{msg})$ **to** NET *{Return execution result to client*

recv $(\texttt{Read}, \mathit{queryID}, \mathit{pid}_C, \mathit{seqNum}_C, \mathit{ch}, \mathit{chaincodeId}, \mathit{query}, \sigma)$ **from** NET **s.t.** $\mathit{pid}_C \in \mathsf{clients}, \mathit{chaincodeId} \in \mathsf{chaincodes}, \mathit{ch} = \mathsf{ch}$: *{Read request*
 send $(\texttt{Verify}, \langle \mathit{queryID}, \mathit{pid}_C, \mathit{seqNum}_C, \mathit{ch}, \mathit{chaincodeId}, \mathit{query}\rangle, \sigma)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathit{pid}_C, \mathsf{sid}_{\mathsf{cur}}, \texttt{client}), \mathcal{F}_{\text{cert}} : \texttt{verifier})$ *{Verify signature of query*
 wait for $(\texttt{Signature}, b)$
 if b:
  $\mathit{queryToBeProcessed} \leftarrow (\mathit{queryID}, \mathit{seqNum}_C, \mathit{pid}_C, \mathit{ch}, \mathit{chaincodeId}, \mathit{query}, \sigma)$
  if $\mathsf{v}_{\mathsf{ep}}(S, \mathit{queryToBeProcessed}) = \texttt{true}$: *{Check whether query matches the endorsing policy*
   if $\mathit{chaincodeId} = 0$:
    $\mathit{output} \leftarrow \mathit{msglist}(\mathsf{chain})$[a] *{Generate query output*
   else:
    $(\mathit{resp}, \mathit{readset}, \mathit{writeset}) \leftarrow \mathsf{simulateTx}(S, \mathit{queryToBeProcessed})$ *{"Execute" transaction proposal*
    $\mathit{output} \leftarrow (\mathsf{pid}_{\mathsf{cur}}, \mathit{queryID}, \mathsf{seqNum}_P, \mathit{chaincodeId}, \mathit{queryToBeProcessed}, \mathit{resp}, \mathit{readset}, \mathit{writeset})$ *{Generate query output*
   $\mathsf{seqNum}_P \leftarrow \mathsf{seqNum}_P + 1$
   send $(\texttt{Sign}, \mathit{output})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, (\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}}), \mathcal{F}_{\text{cert}} : \texttt{signer})$
   wait for $(\texttt{Signature}, \sigma_{\mathit{output}})$
  send $(\texttt{DeliverRead}, \mathit{pid}_C, \mathit{output}, \sigma_{\mathit{output}})$ **to** NET *{Return execution result to client*

[a] For a chain chain, we define $\mathit{msglist}(\mathsf{chain}) = \{(\mathit{id}, \mathit{msg}) \mid$ there exits an entry in chain with ID id and payload $\mathit{msg} \wedge \mathit{msg} = (S, \mathit{chaincodeId}, \mathit{txPayload})$ from the initial transaction proposal included in msg, if parsing does not succeed $\perp\}$.

Fig. 37: Specification of the Fabric* peer $\mathcal{P}^{\mathsf{FAB}^*}_{\texttt{peer}}$ (Part 3)

Description of $\mathcal{F}_{\text{sv}} = (\texttt{supervisor})$:

Participating roles: $\{\texttt{supervisor}\}$
Corruption model: *incorruptible*

Description of $M^{\mathsf{FAB}^*_{\text{BB}}}_{\texttt{supervisor}}$:

Implemented role(s): $\{\texttt{supervisor}\}$
CheckID$(\mathit{pid}, \mathit{sid}, \mathit{role})$**:**
 Accept all messages with the same sid.
Main:
 recv $(\texttt{corruptInt?}, (\mathit{pid}, \mathsf{sid}_{\mathsf{cur}}, \mathit{role}))$ **s.t.** $\mathit{role} \in \{\texttt{kafka}, \texttt{orderer}, \texttt{peer}\}$:
  $\mathit{corrRes} \leftarrow \mathbf{corr}(\mathit{pid}, \mathsf{sid}_{\mathsf{cur}}, \mathit{role})$ *{Request corruption status at dedictated machine*
  reply $(\texttt{corruptInt}, \mathit{corrRes})$ *{Return corruption status*
 recv $(\texttt{IsAssumptionBroken?}, \mathit{prop}, \mathit{id})$:
  reply $(\texttt{IsAssumptionBroken?}, \mathit{prop}, \texttt{false})$

Fig. 38: The supervisor $\mathcal{F}_{\text{sv}}$.

Description of the protocol $\mathcal{F}_{\text{cert}} = (\texttt{signer}, \texttt{verifier})$:

Participating roles: $\{\texttt{signer}, \texttt{verifier}\}$
Corruption model: *incorruptible* {*See text below*
Protocol parameters:
- $\mathsf{p} \in \mathbb{Z}[x]$. {*Polynomial that bounds the runtime of the algorithms provided by the adversary.*
- $\eta \in \mathbb{N}$ {*The security parameter.*
- sig {*Signing algorithm, outputs a signature σ on input (msg, sk). The generated signature has a length of η bits*
- ver {*Signature verifying algorithm, outputs verification result on input $(msg, \sigma, \mathsf{pk})$*
- gen {*Key generation algorithm, outputs $(\mathsf{pk}, \mathsf{sk})$ on input 1^η*

Description of $M_{\texttt{signer},\texttt{verifier}}$:

Implemented role(s): $\{\texttt{signer}, \texttt{verifier}\}$
Internal state:
- $(\mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^2 = (\bot, \bot)$. {*Key pair.*
- $\mathsf{pidowner} \in \{0,1\}^* \cup \{\bot\} = \bot$. {*Party ID of the key owner.*
- $\mathsf{msglist} \subset \{0,1\}^* = \emptyset$. {*Set of recorded messages.*
- $\texttt{corrupted} \in \{\texttt{true}, \texttt{false}\} = \texttt{false}$. {*Is signature key corrupted?*

CheckID$(pid, sid, role)$**:**
Check that $sid = (pid', sid')$:
If this check fails, output $\texttt{reject}$.
Otherwise, accept all entities with the same SID.
{*A single instance manages all parties and roles in a single session. A session models one signature key pair belonging to party pid'.*

Corruption behavior:
- **DetermineCorrStatus**$(pid, sid, role)$: Return $\texttt{corrupted}$.

Initialization:
$(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\eta)$ {*Generate public/secret key pair*
Parse $\mathsf{sid}_{\mathsf{cur}}$ as (pid, sid).
$\mathsf{pidowner} \leftarrow pid$.

Main:
recv $(\texttt{Sign}, msg)$ **from** $\texttt{I/O}$ **to** $(\mathsf{pidowner}, _, \texttt{signer})$**:**
$\sigma \leftarrow \mathsf{sig}^{(\mathsf{p})}(msg, \mathsf{sk})$.
add msg **to** msglist.
reply $(\texttt{Signature}, \sigma)$. {*Record msg for verification and return signature.*

recv $(\texttt{Verify}, msg, \sigma)$ **from** $\texttt{I/O}$ **to** $(_, _, \texttt{verifier})$**:**
$b \leftarrow \mathsf{ver}^{(\mathsf{p})}(msg, \sigma, \mathsf{pk})$. {*Verify signature.*
if $b = \texttt{true} \land msg \notin \mathsf{msglist} \land \texttt{corrupted} = \texttt{false}$**:**
reply $(\texttt{VerResult}, \texttt{false})$. {*Prevent forgery.*
else:
reply $(\texttt{VerResult}, b)$. {*Return verification result.*

recv $\texttt{corruptSigKey}$ **from** $\texttt{NET}$**:** {*Allow network attacker to corrupt signature keys.*
$\texttt{corrupted} \leftarrow \texttt{true}$.
reply $(\texttt{corruptSigKey}, \texttt{ok})$.

Fig. 39: The ideal signature functionality $\mathcal{F}_{\text{cert}}$.

Description of the protocol $\mathcal{F}_{\text{ro}} = (\texttt{randomOracle})$:

Participating roles: $\{\texttt{randomOracle}\}$
Corruption model: *incorruptible*
Protocol parameters:
- $\eta \in \mathbb{N}$ {*Security parameter, length of the hash*

Description of $M_{\texttt{randomOracle}}$:

Implemented role(s): $\{\texttt{randomOracle}\}$
Internal state:
- $\mathsf{hashHistory} \subseteq \{0,1\}^* \times \{0,1\}^\eta$, initially $\mathsf{hashHistory} = \emptyset$ {*The set of recorded value/hash pairs*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

Main:
recv (pid, x)**:** {*Requesting the $\mathcal{F}_{\text{ro}}$ for "hashes"*
if $\exists h \in \{0,1\}^\eta$ **s.t.** $(x, h) \in \mathsf{hashHistory}$**:** {*Extract existing value from* hashHistory
reply (pid, h)
else:
$h \xleftarrow{\$} \{0,1\}^\eta$ {*Generate "hash value" uniformly at random*
$\mathsf{hashHistory} \leftarrow \mathsf{hashHistory.add}((\mathsf{x}, \mathsf{h}))$ {*Store generated key, value pair in* hashHistory
reply (pid, h)

Fig. 40: The random oracle $\mathcal{F}_{\text{ro}}$ (cf. [15])

Description of the subroutine $\mathcal{F}_{\text{read},\mathsf{FAB}^*_{\text{BB}}} = (\texttt{read})$:

Participating roles: $\{\texttt{read}\}$
Corruption model: *incorruptible*
Protocol parameters:
- simulateTx — {*Algorithms check whether endorsement policies are fulfilled and simulate all chaincodes*
- $\mathsf{ch} \in \mathbb{N}$ — {*The channel ID to match Fabric**

Description of $M_{\texttt{read}}$:

Implemented role(s): $\{\texttt{read}\}$
Subroutines: $\mathcal{F}_{\text{init}} : \texttt{init}$
Internal state:
- $\mathsf{recentState} : \{0,1\}^* \to \in \mathbb{N}$ — {*The most recent eact client saw during a read requests*

CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

Main:

```
recv (Read, msg, suggestedOutput, internalState^a) from I/O s.t. msg ≠ (peers, id, query):
    if brokenProps[consistency, public] = true ∧ id = 0:        {F^acc_ledger does not provide consistency any longer
        reply (FinishRead, suggestedOutput, ε)                   {A fully defines the output if accountability w.r.t. consistency is broken
    if id = 0:                                                  {Handle case where the client queries for the prefix of the state/transactions
        if recentState[pid_cur] > suggestedOutput:
            reply (FinishRead, ⊥, ε)                             {Delivery request of A was denied
        else:
            recentState[pid_cur] ← suggestedOutput
            reply (FinishRead, msglist(suggestedOutput)^b)
    if brokenProps[smartRead, public] = true:                    {F^acc_ledger does not provide smart read any longer
        reply (FinishRead, suggestedOutput, ε)                   {A fully defines the output if accountability w.r.t. consistency is broken
    if recentState[pid_cur] > suggestedOutput:                   {The state provided by A is not a prefix of pid's known state
        reply (FinishRead, ⊥, ε)                                 {Delivery request of A was denied
    else:
        Extract queryID from F^acc_ledger's readQueue by counting all read requests of pid_cur up to the current one.
        Extract seqNum from F^acc_ledger's transcript by counting all read and write request of pid_cur up to the current one.
        queryToBeProcessed ← (queryID, seqNum, pid_cur, ch, id, query)
        Let S be Fabric* ledger state defined by prefix of msglist up to entry suggestedOutput
        (resp, readset, writeset) ← simulateTx(S, queryToBeProcessed)    {"Execute" transaction proposal
        reply (FinishRead, resp)                                 {Deliver output of the executed smart contract to requestor. Leak full information to A
```

[a] For brevity we use data from $internalState$ with the local variant of the variable name from $\mathcal{F}_{\text{ledger}}$. This includes local variables such as $msglist$, $requestQueue$, $readQueue$, and $round$.

[b] For $n \in \mathbb{N}$, we define $msglist(n), n \in \mathbb{N} = \{(id, msg) \mid (id, _, msg', _, _) \in msglist \wedge id \leq n \wedge msg = (S, chaincodeId, txPayload)$from the initial transaction proposal included in msg, if parsing does not succeed $\perp\}$.

Fig. 41: $\mathcal{F}^{\text{acc}}_{\text{ledger}}$'s read functionality $\mathcal{F}_{\text{read},\mathsf{FAB}^*_{\text{BB}}}$

Description of the subroutine $\mathcal{F}_{\text{submit},\mathsf{FAB}^*_{\text{BB}}} = (\texttt{submit})$:

Participating roles: $\{\texttt{submit}\}$
Corruption model: *incorruptible*
Protocol parameters:
- $\mathsf{ch} \in \mathbb{N}$ — {*The channel ID to match Fabric**

Description of $M^{\text{BB}}_{\texttt{submit}}$:

Implemented role(s): $\{\texttt{submit}\}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.

Main:

```
recv (Submit, msg, internalState) from I/O:        {See Figure 21 for definition of internalState and the local variables it includes
    if msg ≠ (ch, chaincodeId, txPayload):
        reply (Submit, false, ϵ)                     {Submitted transactions need to have the expected data format
    else:
        reply (Submit, true, (reqCtr + 1, (ch, chaincodeId, txPayload)))
                                                     {If requests have the correct form, they are accepted in the
                                                     first place. Full request leaks including request pointer.
```

Fig. 42: $\mathcal{F}^{\text{acc}}_{\text{ledger}}$'s submit functionality $\mathcal{F}_{\text{submit},\mathsf{FAB}^*_{\text{BB}}}$

Description of the protocol $\mathcal{F}_{\text{init},\mathsf{FAB}^*_{\text{BB}}} = (\texttt{Init})$:

Participating roles: $\{\texttt{init}\}$
Corruption model: incorruptible

Description of $M^{\mathsf{FAB}^*_{\text{BB}}}_{\text{init}}$:

Implemented role(s): $\{\texttt{init}\}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:
recv `Init`**:** *{Empty initialization.*
reply $(\texttt{Init}, \epsilon, \epsilon, \epsilon, \epsilon)$

Fig. 43: The initialization functionality $\mathcal{F}_{\text{init},\mathsf{FAB}^*_{\text{BB}}}$.

Description of $\mathcal{F}_{\text{judgeParams},\mathsf{FAB}^*_{\text{BB}}} = (\texttt{judgeParams})$:

Participating roles: $\{\texttt{judgeParams}\}$
Corruption model: *incorruptible*

Description of $M^{\text{BB}}_{\texttt{judgeParams}}$:

Implemented role(s): $\{\texttt{judgeParams}\}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:
recv $(\texttt{BreakAccProp}, verdict, toBreak, internalState)$ **from** `I/O`**:**
if $verdict[\texttt{public}]$ ensures individual accountability**:** *{Ensure public individual accountability*
if $toBreak = \{\texttt{consistency}, \texttt{public}\} \vee \{\texttt{smartRead}, \texttt{public}\} \vee$ their union**:**
{Handle violation of accountability w.r.t. consistency/smart read
reply $(\texttt{BreakAccProp}, \texttt{true}, \varepsilon)$
else:
reply $(\texttt{BreakAssumption}, \texttt{false}, \varepsilon)$
recv $(\texttt{GetJudicialReport}, msg, internalState)$ **from** `I/O`**:** *{Generate judicial report*
if verdicts $\neq \varepsilon$**:**
reply $(\texttt{GetJudicialReport}, \varepsilon)$ *{Cannot provide a judicial report in this case*
else:
Extract $maxPtr$ from $\mathcal{F}^{\text{acc}}_{\text{cBB}}$'s transcript (from $internalState$) as the highest pointer that was used to answer a read request.
send responsively `GetState` **to** `NET` $(*)$ *{$\mathcal{A}$ may define the prefix to be delivered as judicial report*
wait for $(\texttt{GetState}, ptr)$
if $ptr < maxPtr \vee ptr > |msglist|$**:**
Go to $(*)$.
$list \leftarrow \{(id, msg') \mid (id, msg', _) \in \mathsf{msglist}[maxPtr]\}$
reply $(\texttt{GetJudicialReport}, list)$ *{Return state as report*
recv $(\texttt{Corrupt}, (\texttt{public}, sid, \texttt{judge}), internalState)$ **from** `I/O`**:** *{$\mathcal{F}_{\text{judgeParams},\mathsf{FAB}^*_{\text{BB}}}$ declines corruption requests for the public judge*
reply `false` *{The public judge is incorruptible*
recv $(\texttt{CorruptionStatus?}(\texttt{public}, sid, \texttt{judge}), internalState)$ **from** `I/O`**:** *{$\mathcal{F}_{\text{judgeParams},\mathsf{FAB}^*_{\text{BB}}}$ asks for the public judge's corruption status*
reply `false` *{The public judge is incorruptible*

Fig. 44: The judge parameter functionality $\mathcal{F}_{\text{judgeParams},\mathsf{FAB}^*_{\text{BB}}}$.

Description of the subroutine $\mathcal{F}_{\text{update},\mathsf{FAB}^*_{\text{BB}}} = (\texttt{update})$:

Participating roles: $\{\texttt{update}\}$
Corruption model: *incorruptible*

Description of $M^{\mathsf{FAB}^*_{\text{BB}}}_{\texttt{update}}$:

Implemented role(s): $\{\texttt{update}\}$
Subroutines: $\mathcal{F}_{\text{init}} : \texttt{init}$
CheckID$(pid, sid, role)$**:**
Accept all messages with the same sid.
Main:

```
recv (Update, msg, internalState) from I/O:    {See Figure 21 for definition of internalState and the local variables it includes
    Let ctr be the highest ID in msglist
    if msg ≠ ((ctr + 1, reqCtr_1, msg_1), ..., (ctr + m, reqCtr_m, msg_m)):
        reply (Update, ε, ε, ε)                 {Update rejected, empty msglist extension
    msgListAppend ← ε; updRequestQueue ← ε
    for i = 1 to m do:
        if ∃(reqCtr_i, _, _) ∈ requestQueue with (reqCtr_i, (ch', chaincodeId', txPayload'), r, pid) ∈ requestQueue:
            updRequestQueue.add((reqCtr_i, (ch', chaincodeId', txPayload'), r, pid))
            msg_i = (ch, chaincodeId', txPayload')
        msgListAppend.add((ctr + i, round, tx, msg_i, round, A))
    reply (Update, msgListAppend, updRequestQueue, msgListAppend)    {Return list extension and updated queue.
```

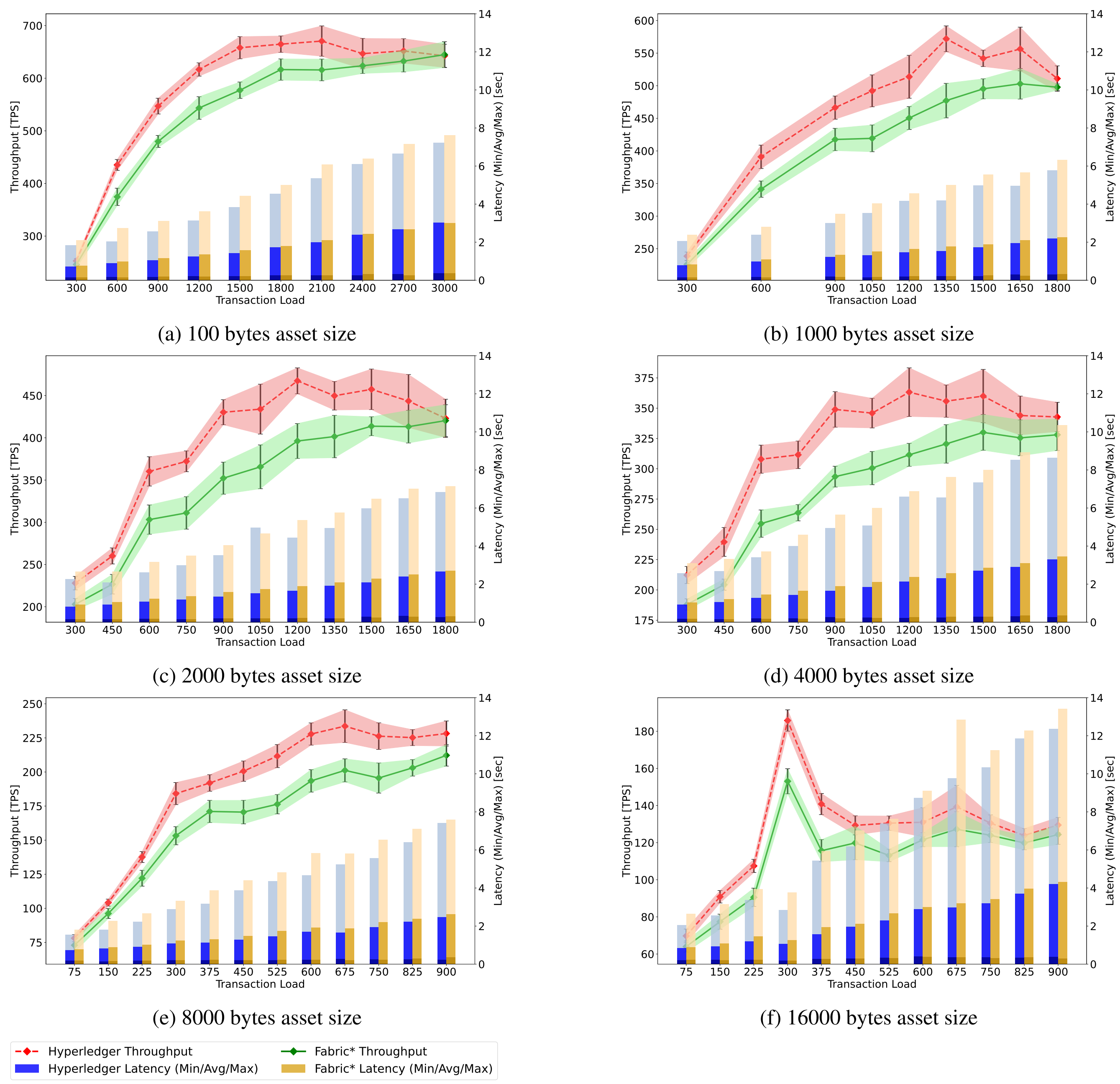
Fig. 45: The update functionality $\mathcal{F}_{\text{update},\mathsf{FAB}^*_{\text{BB}}}$

G. Performance Tests

In Figure 46, we provide further plots of our Fabric* performance tests as discussed in Section V. We provide a summary of the performance decrease for Fabric* with some more details in Table III.

Transaction Size (bytes)	100	1000	2000	4000	8000	16000
Average Throughput Decrease (%)	7.4	10.3	12.0	11.8	12.0	10.2
Maximum Throughput Decrease (%)	13.9	16.5	18.1	17.2	16.7	17.8
Average Latency Increase (%)	5.7	9.8	11.4	11.2	12.0	11.5
Maximum Latency Increase (%)	10.9	16.3	16.7	15.0	18.1	24.5

TABLE III: Throughput decrease and latency increase for Fabric*



(a) 100 bytes asset size

(b) 1000 bytes asset size

(c) 2000 bytes asset size

(d) 4000 bytes asset size

(e) 8000 bytes asset size

(f) 16000 bytes asset size

Fig. 46: Performance Comparison of Hyperledger Fabric and Fabric*