





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 require trusted 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, 37, 45, 52] electronic auctions [49], or multi-party computation (MPC) [7, 8, 51] protocols. At their core, BBs are essentially *broadcast channels with memory* [33] 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* [21] while other systems require that clashing items can appear on the BB [35], 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* [33], 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, 17, 19, 21, 33, 34, 42]. However, only [21, 36, 42] 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 [21, 42], an honest majority [42], or a threshold of honest BB parties [36]. 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, 22, 26, 32, 43]) 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, 37, 45, 48, 51], 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 [18, 36].

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 reusable 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 [24]. 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 [21, 36, 42] 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* [24, 29, 44, 46] 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 [21, 34, 42], 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 [29], 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* [27] 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 [16], 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. [29]). 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.

Tool 2: Universal Composability. The universal composability (UC) paradigm (e. g., [10, 11, 47]) 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., [21] requires a BB with non-clashing items while [35] 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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ therefore is already sufficient. We further explain how $\mathcal{F}_{\text{BB}}^{\text{acc}}$ may be extended to capture all standard BB security properties from the literature. We merge these extensions into the highly customizable BB functionality $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ 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 account-

¹In 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.

²Ideally, one might want to identify *all* misbehaving parties. This is generally not possible since some types of misbehavior cannot be observed [46].

ability, $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ 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^{*}_{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 [23]. In [27], 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^{*}_{BB} by slightly extending and instantiating Fabric^{*}. This extension lifts accountability guarantees from the core components to the full protocol including clients. We then formally prove that Fabric^{*}_{BB} realizes an instantiation of $\mathcal{F}_{\text{cBB}}^{\text{acc}}$, 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 $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ and show that Fabric^{*}_{BB} also achieves this property. As we discuss in Section IV-C, this implies as a simple corollary that Fabric^{*}_{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^{*}_{BB} realizes $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{cBB}}^{\text{acc}}$. The UC composition theorem then implies that all security results for \mathcal{P}' are retained even if \mathcal{P}' is later implemented using Fabric^{*}_{BB}.

Performance Evaluation of Fabric^{*}_{BB}. As we detail in Section V, the overall performance of Fabric^{*}_{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^{*} [31]. Our

results show that Fabric^{*}_{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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$ and $\mathcal{F}_{\text{cBB}}^{\text{acc}}$. 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^{*}_{BB} by instantiating and slightly extending Fabric^{*} and prove that Fabric^{*}_{BB} UC-realizes an instantiation of $\mathcal{F}_{\text{cBB}}^{\text{acc}}$, 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^{*}_{BB} is practical.

Structure of This Paper. After recalling preliminaries in Section II, we propose $\mathcal{F}_{\text{BB}}^{\text{acc}}$ and $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ in Section III. Section IV recalls the Fabric^{*} protocol, proposes Fabric^{*}_{BB} based on Fabric^{*}, and shows that Fabric^{*}_{BB} realizes an instantiation of $\mathcal{F}_{\text{cBB}}^{\text{acc}}$. Section V presents our implementation and benchmarks of Fabric^{*} (available for download at [31]) and discusses the implications for Fabric^{*}_{BB}. We discuss related work in Section VI. Full details and proofs are given in our technical report [30].

II. PRELIMINARIES

A. Computational Model

There are many different models following the universal composability paradigm, e. g., [10, 11, 47]. Formally, here we use the iUC model, a highly general model by Camenisch et al. [10]. 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* S that controls the network of \mathcal{F} such that \mathcal{P} and \mathcal{F} (alongside 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 S on \mathcal{F} ’s I/O interface and the network interface of S it exposes to \mathcal{E} ; equivalently one can consider a *real adversary* that runs alongside \mathcal{P} , analogously to \mathcal{F} running with 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 our technical report [30].

B. Accountability in UC

Graf et al. [28] recently proposed the AUC framework, which provides a general blueprint for modeling and formally proving accountability of arbitrary security properties of protocols within any UC model. 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 [28] and our technical report [30].

Formalizing Accountability in Ideal Functionalities. Start with an ideal functionality \mathcal{F} that formalizes (preventive security of) a certain set of security properties Sec , such as consistency, non-clashing, or liveness. Intuitively, AUC modifies \mathcal{F} using the following main ideas to capture accountability of a subset $Sec^{acc} \subseteq Sec$ 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 $p \in Sec^{acc}$ 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 form $dis((pid, sid, role))$ 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 Sec^{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 $p \in Sec^{acc}$: 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} .

III. AN IDEAL INDIVIDUALLY ACCOUNTABLE BB

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

A. The Ideal Accountable BB Functionality \mathcal{F}_{BB}^{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}_{BB}^{acc} functionality that formalizes accountability w.r.t. consistency and hence also append-only. Thus, \mathcal{F}_{BB}^{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, 12–15]). These functionalities have only been used to model setup assumptions within

³While AUC also supports other types of judges, we focus on public accountability and thus public judges, sometimes just called judge in what follows.

higher-level protocols and, at their core, all work similarly to \mathcal{F}_{BB} reproduced in Figure 1. That is, clients can write a message to \mathcal{F}_{BB} which is then internally appended to a global list of messages `msglist`. Clients can also read from \mathcal{F}_{BB} to obtain the current `msglist`. Thus \mathcal{F}_{BB} formalizes (preventive security of) consistency and an append-only state.

Description of \mathcal{F}_{BB} :	
Implemented role(s): <code>{client}</code>	
Corruption model: <code>incorruptible</code>	
CheckID (<code>pid, sid, role</code>):	<code>{Each instance of this functionality is responsible for an entire session sid.}</code>
Accept all messages with the same <code>sid</code> .	
Main:	
rcv (<code>Write, msg</code>) from <code>I/O</code> :	<code>{Write request from a honest identity}</code>
<code>id ← id + 1; msglist.add(id, msg)</code>	<code>{Record message at position id}</code>
rcv <code>Read</code> from <code>I/O</code> :	<code>{Read request from an honest identity}</code>
reply (<code>Read, msglist</code>)	<code>{Provide full item list to requester}</code>

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

All ideal functionalities in the spirit of \mathcal{F}_{BB} [4, 7, 12–15] model a perfectly secure BB that does not exist in reality. For example, \mathcal{F}_{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}_{BB} of a BB that \mathcal{P}_{BB} does not realize \mathcal{F}_{BB} . As a result, \mathcal{F}_{BB} cannot be used for the security analysis of realistic BBs. Even more, security results obtained for higher-level protocols based on \mathcal{F}_{BB} , including all of the aforementioned works [1, 7, 8, 37, 45, 48, 51], might not hold true when those higher-level protocols are based on an actual implementation \mathcal{P}_{BB} .

In what follows, we therefore lift \mathcal{F}_{BB} from a setup assumption to $\mathcal{F}_{\text{BB}}^{\text{acc}}$ which can be realized by realistic BB protocols. We also incorporate accountability-based security into $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$ not only requires adding accountability to \mathcal{F}_{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}_{BB} , irrespective of accountability. We present the core logic of our proposed $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$. We provide the full formal definition of $\mathcal{F}_{\text{BB}}^{\text{acc}}$ in [30]

The ideal functionality $\mathcal{F}_{\text{BB}}^{\text{acc}}$ captures accountability w.r.t. consistency, i.e., we consider $\text{Sec}^{\text{acc}} = \{\text{consistency}\}$ and no other (preventive) security properties. Clients in $\mathcal{F}_{\text{BB}}^{\text{acc}}$ can issue `Write` and `Read` requests. However, unlike \mathcal{F}_{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

⁴By 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:	
rcv (<code>Write, msg</code>) from <code>I/O</code> :	<code>{Write request from a honest identity}</code>
<code>writeCtr ← writeCtr + 1</code>	
<code>writeQueue.add(writeCtr, msg)</code>	<code>{Store msg for update}</code>
send (<code>Write, writeCtr, msg</code>) to <code>NET</code>	<code>{Leak full msg.}</code>
rcv <code>Read</code> from <code>I/O</code> :	<code>{Read request from an honest identity}</code>
<code>readCtr ← readCtr + 1</code>	
<code>readQueue.add((pid_{call}, sid_{call}, role_{call}), readCtr, false, 0)</code>	
<code>{Store for later processing, where (pid_{call}, sid_{call}, role_{call}) denotes the calling entity}</code>	
send (<code>Read, readCtr</code>) to <code>NET</code>	<code>{Leak full details}</code>
rcv (<code>DeliverRead, readCtr, (sugPtr, sugOutput)</code>) from <code>NET</code>	<code>{A triggers message}</code>
s.t. <code>((pid, sid, role), readCtr, false, 0) ∈ readQueue:</code>	<code>{delivery}</code>
if <code>brokenProps[consistency] = false:</code>	<code>{F_{BB}^{acc} provides consistency in the absence of a verdict}</code>
if <code>∃((pid, sid, role), _, true, prt) ∈ readQueue:</code>	
s.t. <code>prt > sugPtr</code>	
send <code>nack</code> to <code>NET</code>	<code>{Delivery request of A was denied}</code>
else:	
<code>readQueue.remove((pid, sid, role), readCtr, false, 0)</code>	<code>{Clean up}</code>
<code>readQueue.add((pid, sid, role), readCtr, true, sugPtr)</code>	
send (<code>Read, msglist(sugPtr)</code>) to <code>(pid, sid, role)</code>	
else:	<code>{A defines the output if consistency is broken}</code>
<code>readQueue.remove((pid, sid, role), readCtr, msg, false, 0)</code>	
<code>readQueue.add((pid, sid, role), readCtr, msg, true, 0)</code>	
send (<code>Read, sugOutput</code>) to <code>(pid, sid, role)</code>	
rcv (<code>Update, appendToMsglist, updRequestQueue</code>) from <code>NET</code>	
s.t. <code>updRequestQueue ⊂ writeQueue</code>	
<code>all entries from updRequestQueue appear in appendToMsglist:</code>	<code>{Update or maintain request triggered by the adversary.}</code>
<code>msglist.append(appendToMsglist)</code>	
<code>{append adds each entry from appendToMsglist to msglist including consecutive IDs}</code>	
for <code>all item ∈ updRequestQueue</code> do:	<code>{Remove “consumed” elements from writeQueue}</code>
<code>writeQueue.remove(item)</code>	
reply (<code>Update, msglist</code>)	<code>{Leak data}</code>
Include static code provided by AUC [29,30] here. This code adds a public judge <code>judge</code> . \mathcal{A} can send a verdict <code>v</code> to <code>judge</code> indicating that consistency should be broken. <code>judge</code> checks whether <code>v</code> is fair. If so, <code>judge</code> marks consistency as broken and sets <code>brokenProps[consistency]</code> to <code>true</code> ; otherwise it remains <code>false</code> . If the environment \mathcal{E} asks <code>judge</code> regarding verdicts, <code>judge</code> provides current verdicts (if any) to \mathcal{E} .	

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

`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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$. 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, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ enforces consistency for all clients. Therefore, \mathcal{A} provides `sugPtr` to $\mathcal{F}_{\text{BB}}^{\text{acc}}$ which determines the exact prefix of the `msglist` to be output. $\mathcal{F}_{\text{BB}}^{\text{acc}}$ accepts `sugPtr` only if it includes at least all messages that

the same client has previously already read, if any.⁵ Therefore, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ checks that all stored output pointers for the requesting party in `requestQueue` are smaller or equal to `sugPtr`. Afterwards, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ then stores that the request was processed and that $\mathcal{F}_{\text{BB}}^{\text{acc}}$ delivered the `msglist` prefix up to `sugPtr` to the requestor.

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

To add pending write requests from `writeQueue` to the $\mathcal{F}_{\text{BB}}^{\text{acc}}$'s state, \mathcal{A} can at any point in time issue a `Update` command to $\mathcal{F}_{\text{BB}}^{\text{acc}}$. 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, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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, $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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}_{BB} realizes $\mathcal{F}_{\text{BB}}^{\text{acc}}$, then \mathcal{P}_{BB} must enjoy the same security properties. Since $\mathcal{F}_{\text{BB}}^{\text{acc}}$ and \mathcal{P}_{BB} have to have the same behavior towards the environment \mathcal{E} , \mathcal{P}_{BB} has also to provide consistency as long as the judge (in the real protocol) does not

render a verdict. As soon \mathcal{P}_{BB} 's judge renders a verdict, \mathcal{P}_{BB} does not guarantee consistency any longer.

One can also build higher-level protocols \mathcal{P}' on top of $\mathcal{F}_{\text{BB}}^{\text{acc}}$, which then use that $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$ is later implemented via \mathcal{P}_{BB} .

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

As explained in the introduction, some applications require BBs that achieve additional properties beyond just (accountability w.r.t.) consistency. Our ideal functionality $\mathcal{F}_{\text{BB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{BB}}^{\text{acc}}$ is modified to obtain the customization framework $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ 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 $\mathcal{F}_{\text{cBB}}^{\text{acc}}$. In Section IV-C, we further establish a full instantiation of $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ that formalizes the precise security properties provided by Fabric*.

We provide the full formal definition of the customization framework $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ in [30]. In summary, it involves the following changes compared to $\mathcal{F}_{\text{BB}}^{\text{acc}}$:

Customizations of operations: The customization framework $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ is derived from $\mathcal{F}_{\text{BB}}^{\text{acc}}$ mainly by introducing several additional subroutines, namely $\mathcal{F}_{\text{write}}$, $\mathcal{F}_{\text{update}}$, and $\mathcal{F}_{\text{read}}$, 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 $\mathcal{F}_{\text{cBB}}^{\text{acc}}$. When called by $\mathcal{F}_{\text{cBB}}^{\text{acc}}$, a subroutine receives the full internal state allowing it make decisions given the full view. The subroutines $\mathcal{F}_{\text{write}}$, $\mathcal{F}_{\text{update}}$, and $\mathcal{F}_{\text{read}}$ 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 $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ framework additionally contains an internal clock. Formally, this is just an in-

⁵We require only a prefix, instead of the full `msglist`, to allow for realizations \mathcal{P}_{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.

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

ternal counter round which models arbitrary discrete time steps such as seconds or network rounds. The environment/higher-level protocols can query \mathcal{F}_{cBB}^{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}_{cBB}^{acc} uses a new subroutine \mathcal{F}_{updRnd} to decide whether this request is granted, where \mathcal{F}_{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 [30] we formally verify that the \mathcal{F}_{cBB}^{acc} customization framework captures \mathcal{F}_{BB}^{acc} as a special case:

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

Capturing standard security properties via \mathcal{F}_{cBB}^{acc} . Standard security properties of BBs, including the ones mentioned in the introduction, can be captured via instantiations of \mathcal{F}_{cBB}^{acc} . Here we illustrate several examples, starting with preventive security. The remaining properties are discussed in [30].

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}_{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}_{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}_{write} and \mathcal{F}_{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}_{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}_{BB}^{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 `Secacc`. By the static code of AUC, this has the effect that the adversary can now set `brokenProps[liveness] = true` if and 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}_{updRnd} and \mathcal{F}_{read} to first check whether `brokenProps[liveness] = 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}_{cBB}^{acc} .⁷

IV. FABRIC*

We show that the Fabric* distributed ledger proposed by Graf et al. in [27] can be slightly extended and instantiated to obtain a provably secure, composable, and accountable BB which we call Fabric_{BB}^* . Notably, we are able to prove this result based on standard cryptographic assumptions and without requiring trust in any of the parties running Fabric_{cBB}^* 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 $\mathcal{F}_{cBB, \text{Fabric}_{BB}^*}^{acc}$ of \mathcal{F}_{cBB}^{acc} , the security guarantees that we want our BB to achieve. In Section IV-D we present our BB protocol Fabric_{BB}^* based on Fabric*. Finally, in Section IV-E we formally prove the security of Fabric_{BB}^* . We provide the full details in [30].

A. Recap: The Fabric* Protocol

The Fabric* protocol is derived from the prominent Hyperledger Fabric ledger [2]. Graf et al. [27] 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. [27]. 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

⁷Using techniques established by AUC, it is even possible to formalize security statements in \mathcal{F}_{cBB}^{acc} 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 [29] for more details.

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.

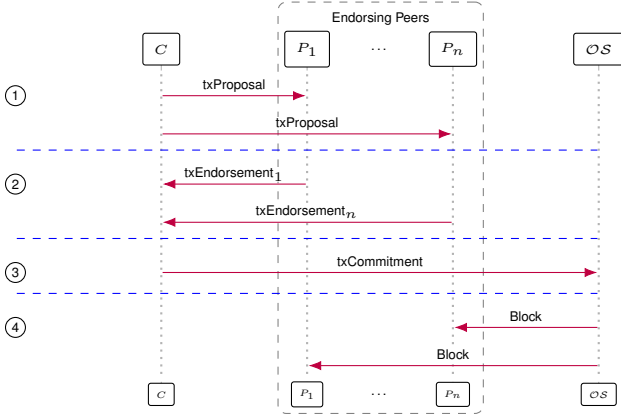


Fig. 3: Example Flow in Fabric* with Client C , (endorsing) peers P_1, \dots, P_n , and ordering service OS (cf. [27]).

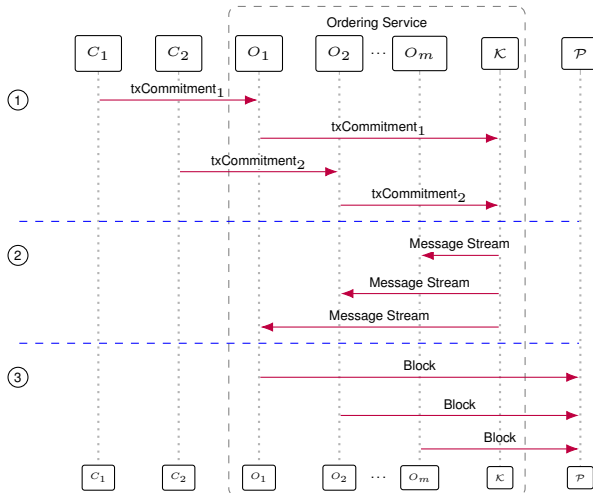


Fig. 4: Ordering and block generation with clients C_1, C_2 , orderers O_1, \dots, O_m , a Apache Kafka \mathcal{K} , and (endorsing) peers \mathcal{P} (cf. [27]).

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):

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 [39]. 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.

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, 38]. 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 [50, 55]. 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. [27] 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 $\text{Fabric}_{\text{BB}}^*$, 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 $\text{Fabric}_{\text{BB}}^*$, 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 $\text{Fabric}_{\text{BB}}^*$ 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 $\text{Fabric}_{\text{BB}}^*$. (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 $\text{Fabric}_{\text{BB}}^*$, we have implemented and provide the first benchmarks for the underlying Fabric*.

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

We want to show accountability w.r.t. two security properties for $\text{Fabric}_{\text{BB}}^*$: 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, 37, 45] 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. For space reasons, we provide details in our technical report [30].

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

The functionality $\mathcal{F}_{\text{cBB}, \text{FAB}_{\text{BB}}^*}^{\text{acc}}$ sets $\text{Sec}^{\text{acc}} = \{\text{consistency}, \text{smartRead}\}$. It is parameterized with an arbitrary but fixed set of algorithms that are identified via IDs $1, \dots, n$. 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}_{cBB}^{acc} : $msg = 0$ indicates a full read whereas $msg = (id, clientInput)$, $id \in \{1, \dots, n\}$ 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:

- $\mathcal{F}_{write, FAB_{BB}^*}$ discards inputs that do not follow the input format of $Fabric_{BB}^*$.
- $\mathcal{F}_{read, FAB_{BB}^*}$ works differently depending on the type of read request, i.e., the auxiliary input msg . For a full read with $msg = 0$, $\mathcal{F}_{read, FAB_{BB}^*}$ captures accountability w.r.t. consistency by using the same logic as \mathcal{F}_{BB}^{acc} in Figure 2. For smart reads with $msg = (id, clientInput)$, $\mathcal{F}_{read, FAB_{BB}^*}$ first checks whether the property has already been broken, i.e., $brokenProps[smartRead] = true$, 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 $msglist$ that is used as input for the smart read. $\mathcal{F}_{read, FAB_{BB}^*}$ 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 $\mathcal{F}_{read, FAB_{BB}^*}$ runs the algorithm with ID id on BB state p and additional input $clientInput$ and instructs $\mathcal{F}_{cBB, FAB_{BB}^*}^{acc}$ to return the result to the client.
- $\mathcal{F}_{update, FAB_{BB}^*}$ uses the same logic as the Update procedure in \mathcal{F}_{BB}^{acc} (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 $Fabric^*$.
- $\mathcal{F}_{updRnd, FAB_{BB}^*}$ always allows time updates since we do not formalize any time-based security properties.

D. Deriving the Accountable BB $Fabric_{BB}^*$ from $Fabric^*$

In what follows, we define our $Fabric_{BB}^*$ based on $Fabric^*$. To obtain $Fabric_{BB}^*$, 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 $\mathcal{F}_{cBB, FAB_{BB}^*}^{acc}$, 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 .

$Fabric_{BB}^*$ 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 $Fabric_{BB}^*$, 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 endorsement⁸ 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 $Fabric_{BB}^*$ 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 $Fabric_{BB}^*$, 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 $Fabric_{BB}^*$ that the smart read was performed on a growing prefix of the state even if different peers are used.

Note that smart reads in $Fabric_{BB}^*$ 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 $Fabric_{BB}^*$ 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 $Fabric_{BB}^*$ which in turn leverages a feature of $Fabric^*$.

Modeling $Fabric_{BB}^*$ as a (real) UC protocol. We model $Fabric_{BB}^*$ as a protocol $\mathcal{P}_{FAB_{BB}^*}^{acc}$ in the iUC model [10], with the structure of the resulting protocol depicted in Figure 5. We provide a formal specification of all machines in [30]. Here, we provide a high-level overview of $\mathcal{P}_{FAB_{BB}^*}^{acc}$.

One session of $\mathcal{P}_{FAB_{BB}^*}^{acc}$ models one $Fabric_{BB}^*$ instance. There can be several sessions of $\mathcal{P}_{FAB_{BB}^*}^{acc}$ running in parallel. $\mathcal{P}_{FAB_{BB}^*}^{acc}$ 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 $Fabric_{BB}^*$ protocol. In each $\mathcal{P}_{FAB_{BB}^*}^{acc}$ 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}_{init} provides the channel setup to the participants, we use a random oracle \mathcal{F}_{ro} to model hash functions, and the ideal certificate functionality \mathcal{F}_{cert} (see, e.g., [28]) captures secure digital signatures with a PKI.

$\mathcal{P}_{FAB_{BB}^*}^{acc}$ 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.

⁸We expect that clients query *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.

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

Showing security of $\text{Fabric}_{\text{BB}}^*$, 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 $\text{Fabric}_{\text{BB}}^*$. Secondly, we then have to show that $\text{Fabric}_{\text{BB}}^*$ together with this specific judge is a secure realization of $\mathcal{F}_{\text{cBB}, \text{FAB}_{\text{BB}}^*}^{\text{acc}}$ and hence achieves all desired accountability-based properties. In this sense, the judge can also be seen as part of the specification of $\text{Fabric}_{\text{BB}}^*$ 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 $\text{Fabric}_{\text{BB}}^*$ and all signatures contained therein are valid. If a peer provides an invalid block as evidence, then he violates the $\text{Fabric}_{\text{BB}}^*$ 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 $\text{Fabric}_{\text{BB}}^*$ 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 reruns 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 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 $\mathcal{F}_{\text{cBB}, \text{FAB}_{\text{BB}}^*}^{\text{acc}}$ be as defined Section IV-C with an arbitrary set of deterministic smart read algorithms. Let $\mathcal{P}_{\text{FAB}_{\text{BB}}^*}^{\text{acc}}$ with the public judge as defined above.*

Then,

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

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 [30]. As part of Theorem 2, we show that $\text{Fabric}_{\text{BB}}$ provides smart read. As detailed in Section IV-C, it hence directly follows that $\text{Fabric}_{\text{BB}}^*$ also provides several additional security properties.

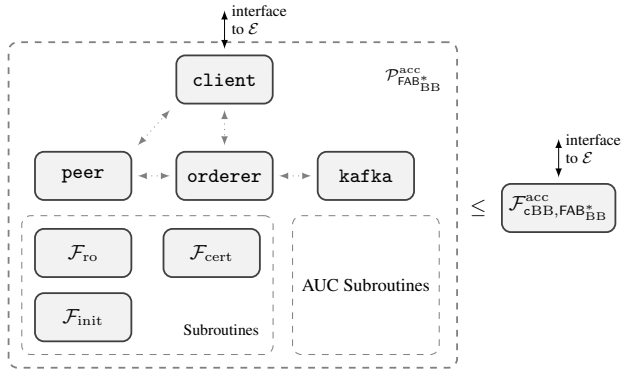


Fig. 5: Figure of Theorem 2. All machines have access to all (AUC) subroutines. \mathcal{A} is connected to all machines. Dotted arrows indicate intended message flows.

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 $\mathcal{F}_{\text{ledger}}^{\text{acc}}$ with certain properties realizes $\mathcal{F}_{\text{cBB}, \text{FAB}^*_{\text{BB}}}^{\text{acc}}$. By this, we not only establish the first formal definition of an accountable ideal ledger functionality $\mathcal{F}_{\text{ledger}}^{\text{acc}}$ 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 $\mathcal{P}_{\text{FAB}^*_{\text{BB}}}^{\text{acc}}$ is such an accountable ledger, i. e., realizes $\mathcal{F}_{\text{ledger}}^{\text{acc}}$. By transitivity of UC security, this gives the overall result.

F. Discussion

In this section, we discuss important aspects of 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 [14]. We use a random oracle \mathcal{F}_{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, 26, 32] 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,

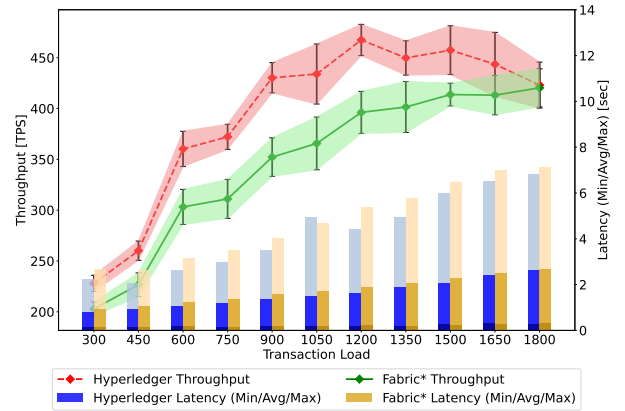


Fig. 6: Performance comparison of Hyperledger Fabric and Fabric* with an transaction size of 2000 bytes.

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. [29]).

V. PERFORMANCE OF FABRIC* RESP. FABRIC*_{BB}

$\text{Fabric}^*_{\text{BB}}$ is essentially an instantiation of Fabric^* with a specific set of smart contracts as well as some added client

⁹This is also why Theorem 2 does not contradict the well-known FLP result [25], 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.

logic that is negligible in terms of overhead. Thus performance of Fabric^{*_{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 Fabric^{*_{BB}} and their implications for practical deployments of our BB. We provide our Fabric^{*} implementation and raw benchmark data at [31].

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 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 [40, 41], 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.

¹⁰The used bwCloud data centers are located in Karlsruhe, Mannheim, and Ulm.

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 [30]. 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 Fabric^{*_{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 Fabric^{*_{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 [37, 45]. Hence, the TPS measured for Fabric^{*} also show the throughput of Fabric^{*_{BB}} in terms of how many write requests/items per second can be added to the BB.

This indicates that the performance of Fabric^{*_{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 Fabric^{*_{BB}} can add more than 220,000 incoming ballots to its state in 10 minutes.

Regarding the performance of smart read operations in Fabric^{*_{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.

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., [19, 21, 33].

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

^aAll 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.

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, 12–15]. 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 anyway.

With $\mathcal{F}_{\text{BB}}^{\text{acc}}$ and $\mathcal{F}_{\text{cBB}}^{\text{acc}}$ 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., [17, 20, 21]. 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 [34, 53] 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 [21, 36, 42]. Table I summarizes and compares key properties of these BBs with our Fabric_{BB}^{*}. Culnane et al. [21] 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. [42] build on and improve the construction of [21] 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. [36] 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

¹¹For 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.

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. Fabric*_{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 [18, 36]. In other words, while many works including ours focus on constructing secure BBs, these works 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, 22, 26, 32, 43, 54], 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* [27] and the relatively recent DLT construction presented by Shamis et al. [54] (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”. This work was also funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), 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).
- [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. Herrevehgen, 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] 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>.
- [11] R. Canetti, “Universally Composable Security,” *J. ACM*, vol. 67, no. 5, pp. 28:1–28:94, 2020.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] I. Cascardo 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.
- [16] M. Castro and B. Liskov, “Practical Byzantine Fault Tolerance and Proactive Recovery,” *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.
- [17] 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.
- [18] 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.

- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] M. del Castillo, "Blockchain 50: Billion Dollar Babies," <https://www.forbes.com/sites/michaelcastillo/2019/04/16/blockchain-50-billion-dollar-babies/>, 2019, (Accessed on 05/01/2019).
- [24] 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.
- [25] 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.
- [26] 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.
- [27] 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.
- [28] M. Graf, R. Küsters, and D. Rausch, "AUC: Accountable Universal Composability," Cryptology ePrint Archive, Tech. Rep. 1606, 2022.
- [29] 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.
- [30] M. Graf, R. Küsters, D. Rausch, S. Egger, M. Bechtold, and M. Flinspach, "Accountable Bulletin Boards: Definition and Provably Secure Implementation," Cryptology ePrint Archive, Tech. Rep. 2023/1869, 2023.
- [31] M. Graf, R. Küsters, D. Rausch, S. Egger, M. Bechtold, and M. Flinspach, "Fabric* - Impedimentation and Performance Testing," <https://github.com/7Y61tY6W9vgScr65UKqm>, 2023.
- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] 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.
- [37] N. Huber, R. Küsters, T. Kripi, J. Liedtke, J. Müller, D. Rausch, P. Reiser, 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.
- [38] Hyperledger Project, "fabricdocs: The Ordering Service," https://hyperledger-fabric.readthedocs.io/en/latest/orderer/ordering_service.html, 2018, (Accessed on 07/24/2019).
- [39] —, "fabricdocs: Chaincode for Developers," <https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html>, 2019, (Accessed on 06/16/2023).
- [40] —, "Github - hyperledger/caliper-benchmarks," <https://github.com/hyperledger/caliper-benchmarks>, 2022, (Accessed on 01/19/2023).
- [41] —, "Hyperledger caliper," <https://www.hyperledger.org/use/caliper>, 2022, (Accessed on 12/30/2022).
- [42] 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.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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>.
- [47] 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.
- [48] 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.
- [49] 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.
- [50] 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).
- [51] M. Rivinius, P. Reiser, 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.
- [52] 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>.
- [53] 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.
- [54] 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.
- [55] 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.