# A Security Framework for Distributed Ledgers

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

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

Viktoria Ronge
viktoria.ronge@fau.de
FAU Erlangen-Nürnberg
Erlangen, Germany

Christoph Egger
christoph.egger@fau.de
FAU Erlangen-Nürnberg
Erlangen, Germany

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

Dominique Schröder
dominique.schroeder@fau.de
FAU Erlangen-Nürnberg
Erlangen, Germany

## ABSTRACT

In the past few years blockchains have been a major focus for security research, resulting in significant progress in the design, formalization, and analysis of blockchain protocols. However, the more general class of distributed ledgers, which includes not just blockchains but also prominent non-blockchain protocols, such as Corda and OmniLedger, cannot be covered by the state-of-the-art in the security literature yet. These distributed ledgers often break with traditional blockchain paradigms, such as block structures to store data, system-wide consensus, or global consistency.

In this paper, we close this gap by proposing the first framework for defining and analyzing the security of general distributed ledgers, with an ideal distributed ledger functionality, called $\mathcal{F}_{\text{ledger}}$, at the core of our contribution. This functionality covers not only classical blockchains but also non-blockchain distributed ledgers in a unified way.

To illustrate $\mathcal{F}_{\text{ledger}}$, we first show that the prominent ideal blockchain functionalities $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{PL}}$ realize (suitable instantiations of) $\mathcal{F}_{\text{ledger}}$, which captures their security properties. This implies that their respective implementations, including Bitcoin, Ouroboros Genesis, and Ouroboros Crypsinous, realize $\mathcal{F}_{\text{ledger}}$ as well. Secondly, we demonstrate that $\mathcal{F}_{\text{ledger}}$ is capable of precisely modeling also non-blockchain distributed ledgers by performing the first formal security analysis of such a distributed ledger, namely the prominent Corda protocol. Due to the wide spread use of Corda in industry, in particular the financial sector, this analysis is of independent interest.

These results also illustrate that $\mathcal{F}_{\text{ledger}}$ not just generalizes the modular treatment of blockchains to distributed ledgers, but moreover helps to unify existing results.

## CCS CONCEPTS

• **Security and privacy → Formal security models**; **Cryptography**; **Distributed systems security**.


*CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea*

ACM ISBN 978-1-4503-8454-4/21/11…$15.00
https://doi.org/10.1145/3460120.3485362

## KEYWORDS

Distributed Ledgers; Blockchain; Universal Composability; Protocol Security; Corda

**ACM Reference Format:**
Mike Graf, Daniel Rausch, Viktoria Ronge, Christoph Egger, Ralf Küsters, and Dominique Schröder. 2021. A Security Framework for Distributed Ledgers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3460120.3485362

## 1 INTRODUCTION

In the past few years, researchers made significant progress in formalizing and analyzing the security of blockchain protocols [3, 5, 16, 21, 26]. Initially analyzed in the game-based setting based on trace properties [15, 20, 38], blockchain security research has moved to simulation-based security which leverages modularity and strong security guarantees offered by universal composability (UC) frameworks [11, 12, 32]. Several ideal blockchain functionalities have been proposed – most notably the functionality by Badertscher et al. [5], called $\mathcal{G}_{\text{ledger}}$, and its privacy-preserving derivative $\mathcal{G}_{\text{PL}}$ [26]. They have successfully been applied to prove the security of various, partly newly designed blockchains (cf. [3, 5, 26]). However, the more general class of distributed ledgers has been out of reach so far:

Distributed ledgers are a generalization of blockchains. A distributed ledger allows for establishing consensus on and distribution of data. While the class of distributed ledgers includes blockchains as a special case, there are several prominent non-blockchain distributed ledgers, such as Corda [9], OmniLedger [30], and Canton [44], which break with several central blockchain paradigms. For example, some of these ledgers do not establish a system-wide consensus, do not use a block structure to store data, and/or do not provide central security goals of traditional blockchains, such as global consistency, chain-growth, or chain-quality. By departing from such blockchain paradigms, these systems aim for higher transaction throughput and security properties like transaction privacy that are not easily provided by blockchains. Both of these aspects are highly desired by industry, thereby making non-blockchain distributed ledgers very attractive for practical use [14, 19, 23, 24, 39].

Due to the conceptual differences between traditional blockchains and non-blockchain distributed ledgers, existing security definitions and results for blockchains do not apply to the class

of distributed ledger protocols in general, and non-blockchain distributed ledgers in particular (cf. Section 3 and Section 4).

In this work, we close this gap by proposing the ideal distributed ledger functionality $\mathcal{F}_{\text{ledger}}$. This functionality provides a highly flexible tool set that allows for the modular security analysis of virtually arbitrary distributed ledgers, thereby, for the first time, covering not only classical blockchains but also non-blockchain distributed ledgers. It does so in a single unified framework.

***The Ideal Ledger Functionality $\mathcal{F}_{\text{ledger}}$.*** To capture and analyze security properties of arbitrary distributed ledger protocols including blockchains, the design of and the features offered by $\mathcal{F}_{\text{ledger}}$ follow these main objectives:

Firstly, $\mathcal{F}_{\text{ledger}}$ is highly flexible due to various parameters, modeled as generic subroutines. This not only allows for capturing a wide range of distributed ledgers, but also a broad spectrum of security properties without having to change the ideal functionality itself. Such security properties include both established (blockchain) security notions, such as consistency and chain-growth, but also entirely new security properties such as *partial consistency*, which we propose of the first time in this work (see below).

Secondly, the interface and core logic of $\mathcal{F}_{\text{ledger}}$ abstract from technical details of the envisioned implementations/realizations, such as purely internal roles (miners or notaries), maintenance operations such as mining, consensus mechanisms (proof-of-work, proof-of-stake, Byzantine agreement, ...), and setup assumptions (networks with bounded delay, honest majorities, trusted parties, ...). All of these details are left to realizations/implementations. Hence, $\mathcal{F}_{\text{ledger}}$ can not only be implemented using vastly different, e.g., consensus mechanisms, but $\mathcal{F}_{\text{ledger}}$ also offers a very simple, clean, and implementation-independent interface to higher-level protocols which should facilitate their specification, modeling, and analysis.

Thirdly, $\mathcal{F}_{\text{ledger}}$ is built for a very general interpretation of corruption: parties in a realization cannot only be corrupted directly, and hence controlled by the adversary, but whether or not they are considered corrupted may also depend on the security assumptions, such as an honest majority. For example, if the honest majority assumption, say in Bitcoin, is violated, one would consider *all* participants to be corrupted, even if they are not directly controlled by the adversary but still run the protocol honestly, since it is impossible for honest parties to provide any security guarantees in this case. We believe that this technique, which has already been successfully employed in the non-blockchain UC literature before (e.g., in [33]), will improve security analyses in the field of distributed ledgers. For example, and as also illustrated by our case study, the commonly used environment-restricting wrapper is typically obsolete when using this general corruption model.

We show the power and generality of $\mathcal{F}_{\text{ledger}}$ via two core results, as further explained below: Firstly, as a fundamental result, we show that existing results for the modular security of blockchains carry over to $\mathcal{F}_{\text{ledger}}$. Secondly, as a case study, we provide the first formal model and security analysis of a non-blockchain distributed ledger, namely the prominent Corda system.

***Covering Blockchains.*** To demonstrate that $\mathcal{F}_{\text{ledger}}$ generalizes the existing literature on blockchains, we first show that $\mathcal{F}_{\text{ledger}}$ is indeed able to capture blockchains as a special case. Instead of illustrating this via a classical case study, which would typically prove that, e.g., Bitcoin realizes $\mathcal{F}_{\text{ledger}}$, we choose a more general approach. We show that the so far most commonly used blockchain functionality $\mathcal{G}_{\text{ledger}}$ [5] (with some syntactical interface alignments) realizes a suitable instantiation of $\mathcal{F}_{\text{ledger}}$ which captures the security properties provided by $\mathcal{G}_{\text{ledger}}$, and demonstrate that this result also holds for its privacy-preserving variant $\mathcal{G}_{\text{PL}}$ [26]. Hence, any realization of $\mathcal{G}_{\text{ledger}}$ or $\mathcal{G}_{\text{PL}}$ (with the mentioned alignments) also realizes $\mathcal{F}_{\text{ledger}}$, which covers all published UC analyses of blockchains, including Bitcoin [5], Ouroboros Genesis [3], and Ouroboros Crypsinous [26].

We want to emphasize that, while $\mathcal{G}_{\text{ledger}}$ realizes $\mathcal{F}_{\text{ledger}}$, both functionalities differ fundamentally in several core design choices. For example, $\mathcal{G}_{\text{ledger}}$ is designed for the special case of blockchains and hence, among others, requires the security property of consistency for realizations. In contrast, $\mathcal{F}_{\text{ledger}}$ requires only the existence of a totally ordered set of transactions. Similarly, the design rationales for $\mathcal{G}_{\text{PL}}$ and $\mathcal{F}_{\text{ledger}}$ are quite different as well.

We also discuss how other published ideal blockchain functionalities are captured by $\mathcal{F}_{\text{ledger}}$ – though these functionalities have only been used as setup assumptions for higher-level protocols.

Altogether, this shows that $\mathcal{F}_{\text{ledger}}$ can cover the blockchain literature and unifies existing models and results.

***Case study: Corda.*** We demonstrate that $\mathcal{F}_{\text{ledger}}$ can capture non-blockchain distributed ledgers, making $\mathcal{F}_{\text{ledger}}$ the first such functionality. We do so via a case study. That is, we provide the first formal analysis of a non-blockchain distributed ledger: *Corda* [8, 9, 40, 41]. We emphasize that existing ideal blockchain functionalities are not suitable for capturing Corda (cf. Section 3 and Section 4).

Corda is one of the most widely used distributed ledgers. It is used by more than 60 companies, banks, and institutions, including HP Enterprise, Intel, Microsoft, and also by NASDAQ [14, 19, 23–25, 39, 42]. Leading consulting groups identify Corda as the most prominent distributed ledger technology [1, 7, 37].

Understanding the security and privacy of Corda is not only interesting due to its wide spread use in practice, but also from a scientific perspective because of its conceptual differences to other distributed ledger technologies. Compared to traditional blockchains, such as Bitcoin, three major differences strike immediately. First, Corda does not structure transactions in blocks. The second one is the lapse of a common state, i. e., no party has a full view of the state, which in turn improves *privacy* of transactions. In particular, while blockchains strive to achieve the notion of *consistency*, where every party is supposed to have the same full view of the global state, Corda aims to provide a weaker security notion, which we call *partial consistency*. For partial consistency, which we formalize for the first time in this work, parties may see only part of the state but these views put together should result in a consistent global state. The third major difference is the inclusion of a number of trusted parties in Corda, so-called *notaries*, which are used to prevent double-spending (see Section 4 for details).

In our case study, we model Corda and formalize its security and privacy properties via an instantiation of $\mathcal{F}_{\text{ledger}}$. We then show that Corda realizes $\mathcal{F}_{\text{ledger}}$. Our analysis uncovers and defines the level of privacy provided for transactions in Corda, including several meta-information leakages that Corda does not protect against. Further,

while the official specification of Corda requires security only under the assumption that *all* notaries are honest, our analysis shows that Corda achieves security even in the presence of some corrupted notaries, thereby improving on the official security claims.

***Summary of Our Contributions.*** In summary, our contributions are as follows:

- We propose, in Section 2, an ideal functionality – called $\mathcal{F}_{\text{ledger}}$ – for general distributed ledgers. It is the first functionality that can be applied to non-blockchain distributed ledgers. As demonstrated in this work, it covers both traditional blockchains and non-blockchain distributed ledgers. Our functionality offers high flexibility to support a wide variety of different implementations with various security properties while simultaneously exposing a simple and implementation independent interface to higher-level protocols. Thereby $\mathcal{F}_{\text{ledger}}$ not only generalizes but also unifies the landscape of existing functionalities for blockchains.
- We show in Section 3 that our functionality subsumes $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{PL}}$. In particular, this allows for directly transferring all published results on the modular security of blockchains, such as Bitcoin and the Ouroboros family, to our functionality. We further discuss that other published ideal blockchain functionalities, which have so far only been used to model setup assumptions, are also captured by $\mathcal{F}_{\text{ledger}}$.
- In Section 4, we provide the first formal model and security analysis of a non-blockchain distributed ledger, Corda. As part of this, we develop and formalize the novel security notion of partial consistency. Due to Corda's wide-spread use in practice, this case study is a significant contribution in its own right.

We provide further details including full formal specifications and proofs in our technical report [22].

## 2 AN IDEAL FUNCTIONALITY FOR GENERAL DISTRIBUTED LEDGERS

In this section, we present the main contribution of our paper: our ideal functionality $\mathcal{F}_{\text{ledger}}$ for distributed ledgers, which includes "common" blockchains as a special case. At a high level, $\mathcal{F}_{\text{ledger}}$ is designed around a *read* and *write* operation offered to higher-level protocols. This captures the two common operations of distributed ledgers, which allow parties from higher-level protocols to *submit* data to the ledger and *get access* to data from other parties. In what follows, we firstly explain $\mathcal{F}_{\text{ledger}}$ in detail. Afterwards, we elaborate on $\mathcal{F}_{\text{ledger}}$'s capabilities to capture different distributed ledger technologies and established distributed ledger security properties.

### 2.1 Description of $\mathcal{F}_{\text{ledger}}$:

Our functionality $\mathcal{F}_{\text{ledger}}$ is defined in the iUC framework [11], which is a recently proposed, expressive, and convenient general framework for universal composability similar in spirit to Canetti's UC model [12]. We explain our functionality in such a way that readers familiar with the UC model are able to understand it even without knowing the iUC framework.

The functionality $\mathcal{F}_{\text{ledger}}$ is a single machine containing the core logic for handling incoming read and write requests. In addition to this main machine, there are also several subroutine machines

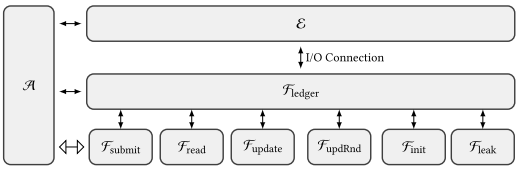


**Figure 1: Overview of $\mathcal{F}_{\text{ledger}}$ and its subroutines. The open headed arrow indicates that $\mathcal{A}$ also connects to all of $\mathcal{F}_{\text{ledger}}$'s subroutines**

that serve as parameters which must be instantiated by a protocol designer to customize the exact security guarantees provided by $\mathcal{F}_{\text{ledger}}$. Figure 1 illustrates the structure of the functionality.[1] Intuitively, $\mathcal{F}_{\text{ledger}}$'s subroutines have the following purposes: $\mathcal{F}_{\text{submit}}$ handles write requests and, e. g., ensures the validity of submitted transactions, $\mathcal{F}_{\text{read}}$ processes read requests and, e. g., models situations that not all clients are up-to-date or ensures privacy properties, $\mathcal{F}_{\text{update}}$ handles updates to $\mathcal{F}_{\text{ledger}}$'s global state, $\mathcal{F}_{\text{updRnd}}$ controls updates to $\mathcal{F}_{\text{ledger}}$'s built-in clock, $\mathcal{F}_{\text{init}}$ determines the initial state of $\mathcal{F}_{\text{ledger}}$, and $\mathcal{F}_{\text{leak}}$ defines the information that leaks upon corruption of a party in $\mathcal{F}_{\text{ledger}}$. As we exemplify in our Corda analysis in Section 4, these subroutines can, in principle, also specify and even share their own additional subroutines. For example, all of the parameterized subroutines could share and access an additional (potentially global) random oracle subroutine in order to obtain consistent hashes for transactions throughout all operations. We note, however, that only the fixed parameterized subroutines can directly access, influence, and change the state of $\mathcal{F}_{\text{ledger}}$. Any additional subroutines are transparent to $\mathcal{F}_{\text{ledger}}$ and only serve to further structure, modularize, and/or synchronize the fixed parameterized subroutines. The rest of this section describes and discusses the static subroutines in more detail.

During a run of $\mathcal{F}_{\text{ledger}}$, there can be multiple instances of the ideal functionality, each of which models a single session of a distributed ledger that can be uniquely addressed by a session ID (SID). Each of these instances/sessions handles an unbounded number of parties that can read from and write to the ledger, where a party ID identifies each party (PID). A party (in a session) can either be honest or corrupted, where only honest parties obtain any form of security guarantees. In what follows, we explain – from the point of view of honest parties – the process of submitting new transactions, adding those transactions to the global state, and then reading from that state (cf. Figure 2 for a formal definition of these operations). Dishonest parties and further details are discussed afterwards.

***Submitting transactions.*** During the run of $\mathcal{F}_{\text{ledger}}$, a higher-level protocol can instruct an honest party *pid* in session *sid* of the distributed ledger to submit a transaction *tx*. Upon receiving such a request, $\mathcal{F}_{\text{ledger}}$ forwards the request to the subroutine $\mathcal{F}_{\text{submit}}$,[2] which then decides whether the transaction is accepted, i.e., is "valid", and which exact information of *tx* should leak to the adversary. As a result, $\mathcal{F}_{\text{ledger}}$ expects to receive a boolean value from $\mathcal{F}_{\text{submit}}$ indicating whether the transaction is accepted as well as

[1] We choose machines, instead of just algorithms, as parameters since they are more flexible in terms of storing and sharing state, and since they can interact with the adversary. For example, they could all have access to a global random oracle.

[2] Requests forwarded to subroutines always also contain a copy of the full internal state of $\mathcal{F}_{\text{ledger}}$ to allow subroutines to make decisions based on, e.g., the current list of corrupted parties. In what follows, we keep this implicit for better readability.

**Main** (excerpt):
**recv** (Submit, *msg*) **from** I/O: {*Submission request from an honest identity*
  **send** (Submit, *msg*, internalState) **to** ($\text{pid}_{\text{cur}}$, $\text{sid}_{\text{cur}}$, $\mathcal{F}_{\text{submit}}$ : submit)
  **wait for** (Submit, *response*, *leakage*) **s.t.** *response* ∈ {true, false}
    **if** *response* = true:
      reqCtr ← reqCtr + 1; requestQueue.add(reqCtr, round, $\text{pid}_{\text{cur}}$, *msg*)
  **send** (Submit, *response*, *leakage*) **to** NET
**recv** (Update, *msg*) **from** NET: {*Update request triggered by the adversary.*
  **send** (Update, *msg*, internalState) **to** ($\epsilon$, $\text{sid}_{\text{cur}}$, $\mathcal{F}_{\text{update}}$ : update)
  **wait for** (Update, *listAdd*, *updRequestQueue*, *leakage*)
      **s.t.** *listAdd* $\subset \mathbb{N} \times \{\text{round}\} \times \{\text{tx}, \text{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$
  *max* ← max$\{i | (i, _, _, _, _, _) \in \text{msglist}\}$
  *check* ← *listAdd* ≠ ∅ ∨ *updRequestQueue* ≠ ∅
    **for** $i$ = *max* + 1 to *max* + |*listAdd*| **do**:
      **if** ($\nexists_1 (i, \_, \_, \_, \_, \_) \in$ *listAdd*):
        *check* ← false
      **if** $\exists (i, \_, \text{meta}, \_, a, b) \in$ *listAdd* $\wedge$ ($a \neq \perp \vee b \neq \perp$):
        *check* ← false
    **if** *check*:
      msglist.add(*listAdd*)
        **for all** *item* ∈ *updRequestQueue* **do**:
          requestQueue.remove(*item*)
  **reply** (Update, *check*, *leakage*)
**recv** (Read, *msg*) **from** I/O: {*Read request from an honest identity*
  **send** (InitRead, *msg*, internalState) **to** ($\text{pid}_{\text{cur}}$, $\text{sid}_{\text{cur}}$, $\mathcal{F}_{\text{read}}$ : read)
  **wait for** (InitRead, *local*, *leakage*) **s.t.** *local* ∈ {true, false}
    **if** *local*:
      **send responsively** (InitRead, *leakage*) **to** NET (★)
      **wait for** (InitRead, *suggestedOutput*)
      **send** (FinishRead, *msg*, *suggestedOutput*, internalState)
        **to** ($\text{pid}_{\text{cur}}$, $\text{sid}_{\text{cur}}$, $\mathcal{F}_{\text{read}}$ : read)
      **wait for** (FinishRead, *output*, *leakage*′)
        **if** *output* = ⊥:
          Go back to (★) and repeat the request (local variables *suggestedOutput*, *output*, and *leakage*′ are cleared)
      **send responsively** (FinishRead, *leakage*′) **to** NET
      **wait for** ack; **reply** (Read, *output*)
    **else**:
      readCtr ← readCtr + 1; readQueue.add(*pid*, readCtr, round, *msg*)
      **send** (Read, readCtr, *leakage*) **to** NET
**recv** (DeliverRead, *readCtr*, *suggestedOutput*) **from** NET **s.t.**
      (*pid*, *readCtr*, *r*, *msg*) ∈ readQueue: {*Deliver network read*
  **send** (FinishRead, *msg*, *suggestedOutput*, internalState)
        **to**($\text{pid}_{\text{cur}}$, $\text{sid}_{\text{cur}}$, $\mathcal{F}_{\text{read}}$ : read)
  **wait for** (FinishRead, *output*, *leakage*′)
    **if** *output* ≠ ⊥:
      **send responsively** (FinishRead, *readCtr*, *leakage*′) **to** NET
      **wait for** ack; readQueue.remove(*pid*, *readCtr*, *r*, *msg*)
      **send** (Read, *output*) **to** (*pid*, $\text{sid}_{\text{cur}}$, I/O)
    **else**:
      **send** nack **to** NET

**Figure 2: Excerpt of $\mathcal{F}_{\text{ledger}}$'s handling of submit, read, and update operations. See Figure 5 to 6 in Appendix B for the full specification. $\text{pid}_{\text{cur}}$ is the current party and $\text{sid}_{\text{cur}}$ the ledger's current session . round is the current time.**

an arbitrary leakage. If the transaction *tx* is accepted, $\mathcal{F}_{\text{ledger}}$ adds *tx* together with the submitting party *pid* and a time stamp (see below) to a buffer list requestQueue that keeps track of transactions from honest parties which have not yet been added to the global transaction list. In any case, both the acceptance result as well as the leakage are then forwarded to the adversary.

As mentioned above, the specification of $\mathcal{F}_{\text{submit}}$ is a parameter that is left to the protocol designer to instantiate. This allows for customizing how the format of a "valid transaction" looks like and whether submitted transactions are supposed to remain (partially) private or fully leak to the adversary on the network. For example, most blockchains do not provide any privacy for transactions, and hence, for those blockchains the leakage generated by $\mathcal{F}_{\text{submit}}$ would be the full transaction *tx*. We provide example instantiations of $\mathcal{F}_{\text{submit}}$ as well as of all other subroutines in Sections 3 and 4.

***Adding transactions to the global transaction list.*** At the core of $\mathcal{F}_{\text{ledger}}$ is a global list of transactions msglist, representing the global state of the ledger. These transactions are ordered, i.e., they are numbered without gaps starting from 0, and form the basis for reading requests of honest parties. Furthermore, they are stored along with some additional information: the ID of the party which submitted the transaction and two time stamps indicating when the transaction was submitted, and when it was added to the global state (we discuss the modeling of time further below). In addition to transactions submitted by parties, we also allow the ledger to contain ordered meta-information represented as a special type of transaction without a submitting party and without a submitting a time stamp. This meta transaction can be useful, e.g., to store block boundaries of a blockchain in those cases where this should be captured as an explicit property of a realization. Similar to ideal functionalities for blockchains, the global transaction list of $\mathcal{F}_{\text{ledger}}$ is determined and updated by the adversary, subject to restrictions that ensure expected security properties.

More specifically, at any point in time, the adversary on the network can send an update request to $\mathcal{F}_{\text{ledger}}$. This request, which contains an arbitrary bit string, is then forwarded to the subroutine $\mathcal{F}_{\text{update}}$. The exact format of the bit string provided by the adversary is not a priori fixed and can be freely interpreted by $\mathcal{F}_{\text{update}}$. This subroutine then computes and returns to $\mathcal{F}_{\text{ledger}}$ an extension of the current global state, an update to the list requestQueue of submitted transactions that specify transactions which should be removed (as those have now become part of the global state, or they became invalid concerning the updated global state), and leakage for the network adversary. Upon receiving the response from $\mathcal{F}_{\text{update}}$, $\mathcal{F}_{\text{ledger}}$ ensures that appending the proposed extension to msglist still results in an ordered list of transactions. If this is the case, then $\mathcal{F}_{\text{ledger}}$ applies the proposed changes to both lists. In any case, $\mathcal{F}_{\text{ledger}}$ sends the leakage from $\mathcal{F}_{\text{update}}$ as well as a boolean indicating whether any changes have been applied to the adversary.

The functionality $\mathcal{F}_{\text{ledger}}$, by default, guarantees only that there exists a unique and ordered global list of transactions. Further security properties which should be enforced for the global state can be specified by appropriately instantiating $\mathcal{F}_{\text{update}}$. For example, $\mathcal{F}_{\text{update}}$ can be used to enforce the security properties of *double spending protection* and *no creation*.

We note that the default guarantee provided by $\mathcal{F}_{\text{ledger}}$ (existence of a unique and ordered global list of transactions) is somewhat weaker than the security notion of consistency for blockchains, which additionally requires that all honest parties also obtain the same (prefix of) that global state. Indeed, many distributed ledgers, such as Corda, are not designed to and do not meet this notion of consistency in its traditional sense (cf. our case study in Section 4). If desired, the property of consistency can, of course, also be captured in $\mathcal{F}_{\text{ledger}}$, namely via a suitable instantiation of $\mathcal{F}_{\text{read}}$ (see below).

***Reading from the global state.*** A higher-level protocol can instruct a party of $\mathcal{F}_{\text{ledger}}$ to read from the global state. There are two types of reading requests that we distinguish, namely, *local* and *non-local* read requests: a local read request generates an immediate output based on the current global state, whereas a non-local

read request might result in a delayed output, potentially based on an updated global transaction list, or even no output at all (as determined by the adversary on the network). Local reads capture cases where a client already has a copy of the ledger stored within a local buffer and reads from that buffer. To the best of our knowledge local reads offered by realizations have not been formalized in idealizations before in the blockchain literature. This is a very useful feature for higher-level protocols since when local reads are possible they do not have to deal with arbitrarily delayed responses, dropped responses, or intermediate state changes. In contrast, a non-local read instead models a thin client that first has to retrieve the data contained in the ledger via the network, and hence, cannot guarantee when (and if at all) the read request finishes.

More specifically, when $\mathcal{F}_{\text{ledger}}$ receives a read request, the subroutine $\mathcal{F}_{\text{read}}$ is used to decide whether the read request is performed locally or non-locally (this decision might depend on, e.g., party names or certain prefixes contained in the read-request) and which exact information leaks to the adversary by the read operation. $\mathcal{F}_{\text{ledger}}$ provides the adversary with the responses of $\mathcal{F}_{\text{read}}$. The adversary is then supposed to provide a bit string used to determine the output for the read request. This response is forwarded back to $\mathcal{F}_{\text{read}}$, which uses the bit string to generate the read request's final output. The exact format of the bit string provided by the adversary is not a priori fixed and can be freely interpreted by $\mathcal{F}_{\text{read}}$. Finally, the output is forwarded by $\mathcal{F}_{\text{ledger}}$ to the higher-level protocol.

On a technical level, for properly modeling local read requests, we use a feature of the iUC framework that allows for forcing the adversary to provide an immediate response to certain network messages (in Figure 2 the operation "send responsively" indicates such network messages with immediate responses). That is, if the adversary receives such a network message and wants to continue the protocol run at all, then in the next interaction with the protocol he has to provide the requested response; he cannot interact with any part of the overall protocol before providing the response. As shown in [10], this mechanism can, in principle, also be added to Canetti's UC model. Non-local read requests are split into two separate activations of $\mathcal{F}_{\text{read}}$, with the adversary being activated in-between: the adversary has to be able to delay a response to such requests and potentially also update the global state.

Besides local and non-local reads, any further security properties regarding reading requests can be specified by instantiating $\mathcal{F}_{\text{read}}$ appropriately. $\mathcal{F}_{\text{read}}$ can also be used to model *access* and *privacy* properties of the global state where, e.g., parties may read only those transactions from the global state where they have been involved in. We use the latter in our analysis of Corda (cf. Section 4).

Having explained the basic operations of submitting transactions, we now explain several further details and features of $\mathcal{F}_{\text{ledger}}$.

***Initialization of*** $\mathcal{F}_{\text{ledger}}$**.** Distributed ledgers often rely on some initial setup information – in blockchains often encoded in a so-called *genesis block* – that is shared between all participants. To allow for capturing such initially shared state $\mathcal{F}_{\text{ledger}}$ includes an ideal initialization subroutine $\mathcal{F}_{\text{init}}$ that can be defined by a protocol designer and is used to initialize the starting values of all internal variables of $\mathcal{F}_{\text{ledger}}$, including transactions that are already part of the global transaction list (say, due to a genesis block that is assumed to be shared by all parties).

***Built-in clock.*** Our functionality $\mathcal{F}_{\text{ledger}}$ includes a clock for capturing security properties that rely on time. More specifically, $\mathcal{F}_{\text{ledger}}$ maintains a counter starting at 0 used as a timer. One can interpret this counter as an arbitrary atomic time unit or the number of communication rounds determined by an ideal network functionality. As mentioned above, both the transactions submitted to the buffer requestQueue and transactions included in the global ordered transaction list msglist are stored with timestamps representing the time they were submitted respectively added to the global state. This allows for defining security properties, which can depend on this information.

Higher-level protocols/the environment can request the current value of the timer, which not only allows for checking that passed time was simulated correctly but also allows for building higher-level protocols that use the same (potentially global) timer for their protocol logic. The adversary on the network is responsible for increasing the timer. More specifically, he can send a request to $\mathcal{F}_{\text{ledger}}$ to increase the timer by 1. This request is forwarded to and processed by a subroutine $\mathcal{F}_{\text{updRnd}}$, which gets to decide whether the request is accepted and whether potentially some information is to be leaked to the adversary. If the request is accepted, then $\mathcal{F}_{\text{ledger}}$ increments the timer by 1. In any case, both the decision and the (potentially empty) leakage are returned to the adversary.

The subroutine $\mathcal{F}_{\text{updRnd}}$ can be instantiated to model various time-dependent security properties, such as various forms of *liveness* [20, 21, 38] (see below). Note that the timer in $\mathcal{F}_{\text{ledger}}$ is optional and can be ignored entirely if no security properties that rely on time should be modeled. In this case, $\mathcal{F}_{\text{updRnd}}$ can reject (or accept) all requests from the adversary without performing any checks.

***Corrupted parties.*** At any point in time, the adversary can corrupt an honest party in a certain session of a distributed ledger. This is done by sending a special corrupt request to the corresponding instance of $\mathcal{F}_{\text{ledger}}$. Upon receiving such a request, the ideal functionality uses a subroutine $\mathcal{F}_{\text{leak}}$ to determine the leakage upon a party's corruption. In the case of ledgers without private data where the adversary already knows all transactions' content, this leakage can be empty. However, in cases where *privacy* should be modeled and hence the adversary does not already know all transactions, this leakage typically includes those transactions that the corrupted party has access to.

As is standard for ideal functionalities, we give the adversary full control over corrupted parties. More specifically, $\mathcal{F}_{\text{ledger}}$ acts as a pure message forwarder between higher-level protocols/the environment and the network adversary for all corrupted parties. Also, the adversary may send a special request to $\mathcal{F}_{\text{ledger}}$ to perform a read operation in the name of a corrupted party; this request is then forwarded to and processed by the subroutine $\mathcal{F}_{\text{read}}$, and the response is returned to the adversary. Just as for $\mathcal{F}_{\text{leak}}$, this operation is mainly included for instantiations of $\mathcal{F}_{\text{ledger}}$ that include some form of privacy for transactions, as in all other cases, the adversary already knows the full contents of all transactions.

***Novel interpretation of corruption in realizations.*** Typically, realizations of ideal functionalities use the same corruption model as explained above. That is, a party in a realization considers itself to be corrupted if it (or one of its subroutines) is under direct control of the adversary. While realizations with this corruption

model are supported by $\mathcal{F}_{\text{ledger}}$, we also propose to use a more general interpretation of corruption in realizations (cf., e.g., [33]): parties in a realization of $\mathcal{F}_{\text{ledger}}$ should consider themselves to be corrupted – essentially by setting a corruption flag – not just if the adversary directly controls them, but also if an underlying security assumption, such as honest majority or bounded network delay, is no longer met. Importantly, even if a party sets a corruption flag due to broken assumptions it still follows the protocol honestly. Setting a corruption flag allows a simulator to tell $\mathcal{F}_{\text{ledger}}$ that a party is corrupted, and hence, $\mathcal{F}_{\text{ledger}}$ no longer has to provide security guarantees for parties that rely on broken assumptions.[3]

This interpretation of corruption, which is a novel concept in the field of universally composable security for blockchains and distributed ledgers, avoids having to encode specific security assumptions into $\mathcal{F}_{\text{ledger}}$ (and more generally ideal functionalities for blockchains and distributed ledgers), and hence, makes such functions applicable to a wider range of security assumptions and corruption settings: the corruption status of a party is sufficient to determine whether $\mathcal{F}_{\text{ledger}}$ must provide security guarantees for that party. It is not necessary to include any additional security assumptions of an intended realization in $\mathcal{F}_{\text{ledger}}$ explicitly (e.g., by providing consistency only as long as there is an honest majority of parties) or to add a wrapper on top of $\mathcal{F}_{\text{ledger}}$ that forces the environment to adhere to the security assumptions. Such security assumptions can rather be specified by and stay at the level of the realization, which in turn reduces the complexity of the ideal functionality while enabling a wide variety of realizations based on potentially vastly different security assumptions. We use this more general concept of corruption in our case study of Corda (cf. Section 4.2), where a client considers itself to be corrupted not only if she is under the direct control of the adversary but also if she relies on a corrupted notary. This models that Corda assumes (and indeed requires) notaries to be honest in order to provide security guarantees. Importantly, this is possible without explicitly incorporating notaries and their corruption status in $\mathcal{F}_{\text{ledger}}$. In fact, following the above rationale, $\mathcal{F}_{\text{ledger}}$ still only has to take care of the corruption status of clients.

***Further features.*** $\mathcal{F}_{\text{ledger}}$ also provides and supports many other features, including dynamic registration of clients, different client (sub-)roles with potentially different security guarantees, full support for smart contracts, and a seamless transition between modeling of public and private ledger without having to reprove any security results. We discuss these features in Appendix C.

## 2.2 Ledger Technologies and Security Properties

Having explained the technical aspects of $\mathcal{F}_{\text{ledger}}$, this section discusses that $\mathcal{F}_{\text{ledger}}$ can indeed capture various types and features of distributed ledgers as well as their security properties – including new ones – illustrating the generality and flexibility of $\mathcal{F}_{\text{ledger}}$.

[3]We note that this concept can easily be extended to capture multiple different levels of "broken" assumptions, e.g., to handle cases where the assumption for the security property of liveness is broken, but another assumption that guarantees the property of consistency still holds. The main requirement is that the environment can check that real and ideal world are consistent in their corruption levels.

*2.2.1 Ledger Technologies.* $\mathcal{F}_{\text{ledger}}$ supports a wide range of ledger types and features, including all of the following:

***Types of global state.*** At the core of $\mathcal{F}_{\text{ledger}}$ is the totally ordered msglist, which includes transactions and meta data and is interpreted by $\mathcal{F}_{\text{read}}$ and $\mathcal{F}_{\text{update}}$. By defining both subroutines in a suitable manner, it is possible to capture a wide variety of different forms of global state, including traditional blockchains (e.g., [4, 16, 20, 26, 45]), ledgers with a graph structure (e.g., [6, 8]) or ledgers that use sharding [30, 36, 47]. In Section 3, we describe how blockchains are captured and in Section 4 we capture the global graph used by Corda. To capture sharding, where participants are assigned to a shard of a ledger and are supposed to have a full view of their respective shard, $\mathcal{F}_{\text{update}}$ ensures that each transaction is assigned to a specific shard (this information is stored together with the transaction in msglist). $\mathcal{F}_{\text{read}}$ then ensures that parties have access only to transactions assigned to their respective shard(s).

***Consensus protocols.*** $\mathcal{F}_{\text{ledger}}$ itself is agnostic to the consensus protocol used in the realization. This allows for realizations using a wide variety of consensus protocols such as Byzantine fault-tolerant protocols, Proof-of-Work, Proof-of-Stake, Proof-of-Elapsed-Time, Proof-of-Authority, etc. If desired, it is also possible to customize $\mathcal{F}_{\text{update}}$ to capture properties that are specific to a certain consensus algorithm. In Section 3, we exemplify that $\mathcal{F}_{\text{ledger}}$ can indeed capture Proof-of-Work and Proof-of-Stake blockchains. In Section 4, we show that $\mathcal{F}_{\text{ledger}}$ can capture the partially centralized consensus service of Corda, i.e., Proof-of-Authority. Other consensus mechanisms can be captured using analogous techniques.

***Network models.*** $\mathcal{F}_{\text{ledger}}$ can capture various types of network models, including, e.g., *(i)* synchronous, *(ii)* partially synchronous, and *(iii)* asynchronous networks. To model these cases, $\mathcal{F}_{\text{updRnd}}$ needs to be customized appropriately. For synchronous/partially synchronous network models one typically enforces in $\mathcal{F}_{\text{updRnd}}$ that time/rounds cannot advance as long as messages are not delivered within expected boundaries, say $\delta$ rounds (cf. Section 4). Additionally, one might also define $\mathcal{F}_{\text{read}}$ to give honest parties read access to (at least) all messages in msglist that are more than $\delta$ (or $c \cdot \delta$ for some constant $c$) rounds old. To model fully asynchronous networks, $\mathcal{F}_{\text{updRnd}}$ and $\mathcal{F}_{\text{read}}$ do not impose any restrictions.

***Time models.*** $\mathcal{F}_{\text{ledger}}$ can capture different time models including, e.g., *(i)* synchronous clocks, *(ii)* clocks with bounded time drift, and *(iii)* asynchronous clocks. For synchronous clocks, we can directly use the global clock of $\mathcal{F}_{\text{ledger}}$ which then defines the time for all parties. For other types of clocks, protocol designers typically add a new type of read request (via the bit string *msg* that is part of read requests, say, by using *msg* = `getLocalTime` and interpreting this in $\mathcal{F}_{\text{read}}$) for reading the local time of a party. $\mathcal{F}_{\text{read}}$ then allows the adversary to determine the local time freely (for asynchronous clocks) or subject to the condition that it is within a certain time frame w.r.t. the global time (for clocks with bounded shift).

***Smart contracts and dynamic party (de-)registration.*** As noted above and detailed in Appendix C, $\mathcal{F}_{\text{ledger}}$ can capture both of these features.

*2.2.2 Security Properties.* $\mathcal{F}_{\text{ledger}}$ can capture a wide variety of (combinations of) security properties from the blockchain security literature, including existing properties from both game-based and universally composable settings. This includes the following

properties, which have game-based and/or universally composable formalizations:

***Consistency*** [21, 30, 38] as already explained above, states that honest parties share a prefix of the global state of a ledger. This can be enforced by properly defining $\mathcal{F}_{\text{read}}$ as we also show in Section 3. We note that the notions of agreement, persistence, and common prefix [2, 20] are closely related to consistency and can be covered in an analogous way.

***Chain-growth*** [5, 16, 20, 28, 38] ensures that a blockchain grows at least with a certain speed, i.e., a certain minimal number of blocks is created per time unit. As we show in Section 3, this can be captured in $\mathcal{F}_{\text{ledger}}$ via $\mathcal{F}_{\text{updRnd}}$. Specifically, $\mathcal{F}_{\text{updRnd}}$ rejects round/time update requests whenever there are not sufficiently many blocks yet as would be required for the next time period.

***Chain-quality*** [5, 16, 20, 28, 38] requires that honest users create a certain ratio of blocks in a blockchain in order to prevent censorship. This can be captured in $\mathcal{F}_{\text{ledger}}$, e.g, by recording the block creators as metadata in $\mathcal{F}_{\text{ledger}}$'s msglist. $\mathcal{F}_{\text{update}}$ then rejects updates if they violate chain-quality .

***Liveness*** [5, 16, 20, 28, 38] ensures that transactions submitted by honest clients enter the global state respectively the state read by other honest clients within $\rho$ rounds. As we exemplify in Section 3 and 4, protocol designers can use $\mathcal{F}_{\text{updRnd}}$ to ensure various forms of liveness. Specifically, $\mathcal{F}_{\text{updRnd}}$ forbids the adversary from advancing time as long as conditions for the next time unit are not yet met, e.g., because a transaction that is already $\rho$ rounds old is not yet in the global state.

***Privacy Properties***, such as transaction privacy [31, 35, 43, 46], ensure secrecy of transactions, e.g., that only parties involved in a transaction are aware of its contents. To capture different forms/levels of privacy in $\mathcal{F}_{\text{ledger}}$, the leakages of its subroutines are specified to keep private information hidden from the adversary as long as the adversary does not control any parties that have access to this information. Furthermore, $\mathcal{F}_{\text{read}}$ ensures that also honest parties gain read access only to information that they are allowed to see. In Section 4, we use this technique to formalize and analyze the level of privacy of Corda, including which information is leaked to the adversary for honest transactions.

***Soundness Properties***, such as transaction validity and double-spending protection, can be captured by customizing $\mathcal{F}_{\text{submit}}$ and/or $\mathcal{F}_{\text{update}}$ to reject incoming messages that violate soundness properties. This is exemplified in Sections 3 and 4 with further details provided in our technical report [22].

***New security properties*** that have not yet been formally defined in the distributed ledger security literature can potentially also be supported by $\mathcal{F}_{\text{ledger}}$. One example is our novel notion of partial consistency (cf. Section 4).

In summary, as discussed above, $\mathcal{F}_{\text{ledger}}$ is indeed able to formalize existing security notions from the game-based blockchain security literature [16, 20, 21, 28, 31, 35, 38, 43, 46]. For the universally composable blockchain security literature we show an even stronger statement in Section 3: $\mathcal{F}_{\text{ledger}}$ can not only formalize existing security properties; existing security proofs and security results obtained for concrete blockchains, such as Bitcoin, carry over to $\mathcal{F}_{\text{ledger}}$ (after lifting them to the abstraction level of $\mathcal{F}_{\text{ledger}}$).

## 3 COVERING BLOCKCHAINS WITH $\mathcal{F}_{\text{ledger}}$

In this section, we demonstrate that $\mathcal{F}_{\text{ledger}}$ is able to capture traditional blockchains as a special case. Firstly, we show that the so far most commonly used blockchain functionality $\mathcal{G}_{\text{ledger}}$ [5] (with some syntactical interface alignments) realizes a suitable instantiation of $\mathcal{F}_{\text{ledger}}$, which captures the security guarantees of $\mathcal{G}_{\text{ledger}}$, and demonstrate that this result also holds for its privacy-preserving variant $\mathcal{G}_{\text{PL}}$ [26]. Hence, any realization of $\mathcal{G}_{\text{ledger}}$ or $\mathcal{G}_{\text{PL}}$ (with interface alignments) also realizes $\mathcal{F}_{\text{ledger}}$. This in fact covers all published UC analyses of blockchains, including Bitcoin [5], Ouroboros Genesis [3], and Ouroboros Crypsinous [26]. Secondly, we discuss that $\mathcal{F}_{\text{ledger}}$ can also capture other published ideal blockchain functionalities, which so far have been used only to model setup assumptions for higher-level protocols. Altogether, this illustrates that $\mathcal{F}_{\text{ledger}}$ not only generalizes but also unifies the landscape of ideal blockchain functionalities from the literature.

***The ideal blockchain functionality $\mathcal{G}_{\text{ledger}}$.*** Let us start by briefly summarizing the ideal blockchain functionality $\mathcal{G}_{\text{ledger}}$ (further information, including a formal specification of $\mathcal{G}_{\text{ledger}}$ in the iUC framework, is available in in our technical report [22]. $\mathcal{G}_{\text{ledger}}$ offers a write and read interface for parties and is parameterized with several algorithms, namely validate, extendPolicy, Blockify, and predictTime, which have to be instantiated by a protocol designer to capture various security properties. By default, $\mathcal{G}_{\text{ledger}}$ provides only the security property of *consistency* which is standard for blockchains. An honest party can submit a transaction to $\mathcal{G}_{\text{ledger}}$. If this transaction is valid, as decided by the validate algorithm, then it is added to a buffer list. $\mathcal{G}_{\text{ledger}}$ has a global list of blocks containing transactions. This list is updated (based on a bit string that the adversary has previously provided) in a preprocessing phase of honest parties. More specifically, whenever an honest party activates $\mathcal{G}_{\text{ledger}}$, the extendPolicy algorithm is executed to decide whether new "blocks" are appended to the global list of blocks, with the Blockify algorithm defining the exact format of those new blocks. Then, the validate algorithm is called to remove all transactions from the buffer that are now, after the update of the global blockchain, considered invalid. An honest party can then read from the global blockchain. If the honest party has been registered for a sufficiently long amount of time (larger than parameter $\delta$), then it is guaranteed to obtain a prefix of the chain that contains all but the last at most windowSize blocks. This captures the property of consistency. In addition to these basic operations, $\mathcal{G}_{\text{ledger}}$ also supports dynamic (de-)registration of parties and offers a clock, modeled via a subroutine $\mathcal{G}_{\text{clock}}$, that advances depending on the output of the predictTime algorithm (and further constraints).

As becomes clear from the above short description of $\mathcal{G}_{\text{ledger}}$, $\mathcal{F}_{\text{ledger}}$ draws inspiration from $\mathcal{G}_{\text{ledger}}$. However, there are several fundamental differences:

- $\mathcal{G}_{\text{ledger}}$ is designed for capturing blockchains and therefore, e.g., requires that transactions are stored in a "block" format (via the Blockify algorithm) and always provides the security property of consistency. As already discussed in Section 2, $\mathcal{F}_{\text{ledger}}$ only requires the existence of a totally ordered list of transactions.
- Read operations in $\mathcal{G}_{\text{ledger}}$ *always* output a full prefix of $\mathcal{G}_{\text{ledger}}$'s blockchain in plain, i.e., $\mathcal{G}_{\text{ledger}}$ is built for blockchains without privacy guarantees and those that do not modify/interpret data

in any way. $\mathcal{F}_{\text{ledger}}$ includes a parameter $\mathcal{F}_{\text{read}}$ to modify and also restrict the contents of outputs for read requests, which in turn allows for capturing, e.g., privacy properties (as illustrated by our Corda case study).

- $\mathcal{G}_{\text{ledger}}$ takes a lower level of abstraction compared to $\mathcal{F}_{\text{ledger}}$. That is, $\mathcal{G}_{\text{ledger}}$ has several details of the envisioned realization built into the functionality and higher-level protocols have to take these details into account. In other words, the rationale of how higher-level protocols see and deal with blockchains is different to $\mathcal{F}_{\text{ledger}}$. $\mathcal{G}_{\text{ledger}}$ requires active participation of higher-level protocols/the environment, while $\mathcal{F}_{\text{ledger}}$ models blockchains (and distributed ledgers) essentially as black boxes that higher-level protocols use. More specifically, $\mathcal{G}_{\text{ledger}}$ includes a mining or maintenance operation `MaintainLedger` that higher-level protocols/the environment have to call regularly, modeling that higher-level protocols have to manually trigger mining or state update operations in the blockchain for security to hold true. Similarly, the clock used by $\mathcal{G}_{\text{ledger}}$ also has to be regularly and manually triggered by higher-level protocols/the environment for the run of the blockchain to proceed. In contrast, $\mathcal{F}_{\text{ledger}}$ abstracts from such details and leaves them to the realization. The motivation for this is that higher-level protocols usually do not (want to) actively participate in, e.g., mining operations and rather expect this to be handled internally by the underlying distributed ledger.
- $\mathcal{G}_{\text{ledger}}$ includes a predictTime parameter that, based on the number of past activations (but not based on the current global state/blockchain), determines whether time should advance. This parameter can be synchronized with suitable definitions of the extendPolicy, which has access to and determines the global state, to model time dependent security properties such as liveness. $\mathcal{F}_{\text{ledger}}$ instead allows the adversary to choose arbitrarily when time should advance. The single parameter $\mathcal{F}_{\text{updRnd}}$ can then directly enforce time dependent security properties without requiring synchronization with other parameters (cf. Section 2.2).
- $\mathcal{G}_{\text{ledger}}$ uses algorithms as parameters, whereas $\mathcal{F}_{\text{ledger}}$ uses subroutines, with the advantages explained in Footnote 1.

In summary, the main differences between $\mathcal{G}_{\text{ledger}}$ and $\mathcal{F}_{\text{ledger}}$ are due to *(i)* different levels of abstraction to higher-level protocols and *(ii)* the fact that $\mathcal{G}_{\text{ledger}}$ is built specifically for traditional blockchains. Both of these aspects have to be addressed to show that $\mathcal{G}_{\text{ledger}}$ is a realization of a suitable instantiation of $\mathcal{F}_{\text{ledger}}$. To address *(i)*, we use a wrapper $\mathcal{W}_{\text{ledger}}$ that we add on top of the I/O interface of $\mathcal{G}_{\text{ledger}}$ and which handles messages from/to the environment. This wrapper mainly translates the format of data output by $\mathcal{G}_{\text{ledger}}$ to the format used by $\mathcal{F}_{\text{ledger}}$ (e.g., from a blockchain to a list of transactions). It also handles the fact that $\mathcal{F}_{\text{ledger}}$ does not include certain operations on the I/O interface by instead allowing the adversary $\mathcal{A}$ to run the maintenance operation `MaintainLedger` and perform clock updates in $\mathcal{G}_{\text{clock}}$ even in the name of honest parties. That is, $\mathcal{W}_{\text{ledger}}$ models real world behavior, using $\mathcal{A}$ as a scheduler, where blockchain participants perform mining based on external events, such as incoming network messages, without first waiting to receive an explicit instruction from a higher-level protocol to do so (see also the remarks following Corollary 3.2). Issue *(ii)* is addressed via a suitable instantiation of the parameters of $\mathcal{F}_{\text{ledger}}$

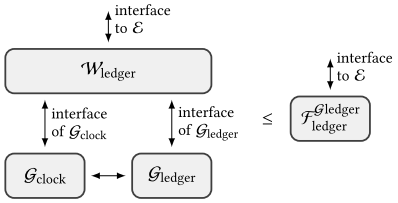


**Figure 3: Realization relation of $\mathcal{G}_{\text{ledger}}$ and $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ as stated in Theorem 3.1. The system $\mathcal{E}$ denotes the environment, modeling arbitrary higher level protocols. All machines are additionally connected to the network adversary.**

in order to capture the same (blockchain) properties provided by $\mathcal{G}_{\text{ledger}}$ respectively the parameterized algorithms of $\mathcal{G}_{\text{ledger}}$. This instantiation roughly works as follows, with full definitions and details provided in our technical report [22]:

- $\mathcal{F}_{\text{init}}$ is defined to run the extendPolicy algorithm to generate the initial transaction list (that is read from the blocks output by the algorithm). This is because extendPolicy might already generate a genesis block during the preprocessing of the first activation of the functionality before any transactions have even been submitted.
- $\mathcal{F}_{\text{submit}}$ executes the validate algorithm to check validity of incoming transactions.
- $\mathcal{F}_{\text{update}}$ executes the extendPolicy and Blockify algorithms to generate new blocks from the update proposed by the adversary. These blocks are transformed into individual transactions which are appended to the global transaction list of $\mathcal{F}_{\text{ledger}}$ together with a special meta transaction that indicates a block boundary. Additionally, the validate algorithm is used to decide which transactions are removed from the transaction buffer.
- $\mathcal{F}_{\text{read}}$ checks whether a party has already been registered for an amount of time larger than $\delta$ and then either requests the adversary to provide a pointer to a transaction within the last windowSize blocks or lets the adversary determine the full output of the party. We note that $\mathcal{F}_{\text{read}}$ has to always use non-local reads: this is because a read operation in $\mathcal{G}_{\text{ledger}}$ might change the global state during the preprocessing phase and before generating an output, i.e., read operations are generally not immediate (in the sense defined in Section 2).
- If the parameters of $\mathcal{G}_{\text{ledger}}$ are such that they guarantee the property of *liveness*, then $\mathcal{F}_{\text{updRnd}}$ can be defined to also encode this property (cf. Section 2); similarly for the time dependent security property of *chain-growth* and other time-related properties.
- $\mathcal{F}_{\text{leak}}$ does not leak (additional) information as all information is leaked during submitting and reading.

Let $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ be the protocol stack consisting of $\mathcal{F}_{\text{ledger}}$ with all of its subroutines instantiated as sketched above. Then we can indeed show that $\mathcal{G}_{\text{ledger}}$ (with the wrapper $\mathcal{W}_{\text{ledger}}$) realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ (cf. Figure 3).

Theorem 3.1 (informal). *Let $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ be as above and let $\mathcal{W}_{\text{ledger}}$ be the wrapper for $\mathcal{G}_{\text{ledger}}$ and its subroutine clock $\mathcal{G}_{\text{clock}}$. Then, $(\mathcal{W}_{\text{ledger}} \mid \mathcal{G}_{\text{ledger}}, \mathcal{G}_{\text{clock}}) \leq \mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$.*

We formalize this theorem and provide precise specifications of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$, $\mathcal{W}_{\text{ledger}}$, $\mathcal{G}_{\text{ledger}}$, and $\mathcal{G}_{\text{clock}}$ as well as a full proof in the

full version of this paper [22]. As explained above, the additional component $\mathcal{W}_{\text{ledger}}$ merely aligns the syntax of $\mathcal{G}_{\text{ledger}}$ and $\mathcal{F}_{\text{ledger}}$, and makes explicit that maintenance operations and clock updates are performed automatically based on external events. In fact, all existing higher-level protocols we are aware of do not trigger maintenance operations and do not update the clock themselves (see, e.g., [27]). They rather leave this to the adversary/environment, as one might expect. Hence, from the point of view of a higher-level protocol, typically it does not matter whether it uses $\mathcal{G}_{\text{ledger}}$ or $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$; there are only slight syntactical alignments necessary.

From Theorem 3.1, transitivity of the realization relation, and the composition theorem of the iUC framework we immediately obtain that existing realizations of $\mathcal{G}_{\text{ledger}}$ also apply to and can be re-used with $\mathcal{F}_{\text{ledger}}$.

Corollary 3.2 (informal). *Let $\mathcal{P}_{blockchain}$ be a realization of $\mathcal{G}_{\text{ledger}}$, e.g., Bitcoin or Ouroboros Genesis. Furthermore, let $Q^{\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}}$ be a higher-level protocol using $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ and let $Q^{\mathcal{P}}$ be the same protocol as $Q$ but using $\mathcal{P}_{blockchain}$ (plus the wrapper $\mathcal{W}_{\text{ledger}}$ and $\mathcal{G}_{\text{clock}}$) instead of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$. Then, $Q^{\mathcal{P}}$ realizes $Q^{\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}}$.*

The corollary intuitively states that if we have analyzed and proven secure a higher-level protocol $Q$ based on $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$, then $Q$ remains secure even if we run it with an actual blockchain $\mathcal{P}_{\text{blockchain}}$ that realizes $\mathcal{G}_{\text{ledger}}$.

***$\mathcal{G}_{\text{PL}}$ and other ideal blockchain functionalities.*** Similarly to the above result, we provide a proof sketch showing that $\mathcal{G}_{\text{ledger}}$'s privacy preserving variant $\mathcal{G}_{\text{PL}}$ [26] (plus a wrapper aligning syntax and mapping abstraction levels) also realizes a suitable instantiation of $\mathcal{F}_{\text{ledger}}$ in our technical report [22]. Hence, the famous privacy preserving blockchain protocol Ouroboros Crypsinous [26], which has been proven to realize $\mathcal{G}_{\text{PL}}$, also realizes $\mathcal{F}_{\text{ledger}}$ with slight adjustments to the interface as described above. Besides $\mathcal{G}_{\text{PL}}$, we also discuss further ideal ledger functionalities [17, 18, 29] in our technical report [22] which so far have only been used to model setup assumptions for higher-level protocols and which have not been realized yet. We show that $\mathcal{F}_{\text{ledger}}$ can be instantiated to model the same security properties as those ideal functionalities and hence can be used as an alternative within higher-level protocols.

# 4 CASE STUDY: SECURITY AND PRIVACY OF THE CORDA LEDGER

Corda is one of the most widely employed distributed ledgers. It is a privacy-preserving distributed ledger where parties share some information about the ledger but not the full view. It is mainly used to model business processes within the financial sector. In this section, we first give a description of Corda. We then provide a detailed security and privacy analysis by proving that Corda realizes a carefully designed instantiation of $\mathcal{F}_{\text{ledger}}$.

## 4.1 Description of the Corda Protocol

There are two types of participants/roles in Corda: *(i) Nodes* or *clients*, who can submit transactions to and read from the ledger, and *(ii) notary services* (called just *notaries* in what follows) which are trusted services that are responsible for preventing double spending. Each participant is identified via its public signing key, which is certified via one or more *certificate agencies* and then distributed via a so-called *network service provider* to all participants. All participants communicate via secure authenticated channels.

Clients own *states* (sometimes also called *facts*) in Corda. A state typically represents an asset that the party owns in reality, e.g., money, bonds, or physical goods, like a car. States can be "spent" via a transaction, which consumes a set of input states and creates a set of new output states. These transactions are validated by notaries to prevent double spending of states.

***States, transactions, attachments.*** On a technical level, a state is represented via a tuple consisting of at least one owner of the state (identified via public signature keys) and an arbitrary bit string that encodes the asset. States are stored as the outputs of transactions in Corda, similar to how Bitcoin stores ownership of currency as an output of a transaction. Transactions in Corda consist of a (potentially empty) set of pointers to *input states*, a (potentially empty) set of pointers to *reference states* (see below), a set of *output states*, a non-empty set of *participants* (clients), a *notary* that is responsible for validating this transaction and for preventing double spending of its inputs, a (potentially empty) set of pointers to *smart contracts*, an arbitrary bit string that can encode parameters for the transaction, and an ID that is computed as a hash over the transaction. The participants contain at least all owners of input states, who are expected to confirm the transaction by a signature. One of the participants takes the role of an *initiator*, who starts and processes the transaction, while the other participants, if any, act as so-called *signees* who, if they agree with the transaction, only add their signatures to confirm the transaction. The set of input states can be empty, which allows for adding new assets to Corda by creating new output states. The referenced smart contracts are stored in so-called *attachments* with a unique ID (computed via the hash of the attachment) and can be used to impose further conditions for the transaction to be performed. These conditions may in particular depend on *reference states*, which, unlike input states, are not consumed by the transaction but rather only provide some additional information for the smart contracts. For example, a smart contract might state that an initiator's car is bought by a signee only if its age is below a certain threshold. A reference state might contain the manufacturing date of the car, including a signature of the manufacturer, which can then be validated.[4]

In the following, we call the set of input states, reference states, and smart contracts the *direct dependencies* of a transaction. The set of *(full) dependencies* of a transaction is a set of all direct dependencies, their respective direct dependencies, and so on. A transaction is called *valid* if the format of the transaction is correct, the set of participants includes all owners of input states, and all smart contracts referenced by the transaction allow the transaction.

***Partial views.*** In a Corda instance, the set of all transactions and attachments used by those transactions forms a global directed graph (which is not necessarily a tree or a forest). However, clients do not obtain a full view of this graph. Instead, each client has only a partial view of the global graph consisting of those transactions it is involved in as an initiator/signee as well as the full dependencies

[4] In addition to reference states, smart contracts can also access so-called oracles, which are trusted third parties, to provide data points. Since the same can also be achieved by reference states, we did not explicitly include oracles in our analysis.

of those transactions. Generally speaking, a client forwards one of its known transactions $tx$ (or one of its known attachments) to another client only if both clients are involved in a transaction $\widehat{tx}$ that (directly or also indirectly) depends on $tx$, i.e., where both clients are allowed to and need to learn $tx$ in order to validate $\widehat{tx}$.

This decentralized graph structure, where clients are supposed to learn only those parts that they actually are involved in, facilitates privacy but makes it impossible for an individual client to detect and protect itself against double spending attacks: Assume Alice has an input state representing a car and she uses this state in a transaction with Bob. Now, Alice might use the same state again in a transaction with Carol. Both Bob and Carol would assume that they now own Alice's car, however, neither of them can detect that Alice has sold her car twice since neither of them is able to see both transactions. To solve this problem, Corda, as already briefly mentioned, introduces the concept of notaries, which are trustees that are responsible for validating transactions and preventing double spending, as discussed in more detail in what follows.

Each transaction $tx$ is assigned one notary $\mathcal{N}$ who is responsible for this transaction; $\mathcal{N}$, just as the participants, also learns the full dependencies of $tx$. To be able to detect double spending of input states, it is required that $tx$ only uses inputs for which $\mathcal{N}$ is also responsible for, i. e., the input state was produced as an output for which $\mathcal{N}$ is responsible. The notary then checks that $tx$ is valid (which entails checking that the set of participants of $tx$ contains all owners of input states), there are valid signatures of all participants, and also that no input state has already been used by another transaction. If this is the case, the notary signs $tx$, which effectively adds $tx$ to the global graph of Corda. To change the notary $\mathcal{N}$ responsible for a certain state to a different one, say $\mathcal{N}'$, Corda offers a special *notary change transaction*. This transaction takes a single input state, generates a single output state that is identical to the input, and is validated by the notary $\mathcal{N}$ who is responsible for the input state. The responsibility for the output is then transferred to $\mathcal{N}'$, i.e., future transactions need to rely on that notary instead.

***Submitting transactions.*** A new transaction is first signed by the initiator, who then forwards the transaction to all signees to collect their signatures. The initiator then sends the transaction together with the signatures to the notary, who adds his own signature to confirm validity of the transaction. The initiator finally informs all signees that the transaction was successful. The initiator is required to know the full dependencies of the transaction such that he can distribute this information to signees and the notary. To obtain this knowledge in the first place, which might include input states known only to, say, one of the signees, clients/signees can proactively send known transactions to other clients. In what follows, we say that a client *pushes* a transaction .

***Customization and security goals.*** All protocol operations in Corda, such as the process of submitting a transaction, can be customized and tailored towards the specific needs of a deployment of Corda. For example, one could decide to simply accept transactions without signatures of a notary, with all of its implications for security and double spending. Our description given above (and our analysis carried out below) of Corda follows the predefined standard behavior which captures the most typical deployment as

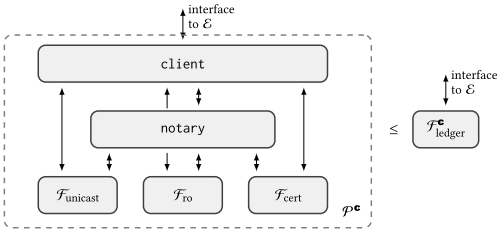


**Figure 4: Corda protocol $\mathcal{P}^{\mathbf{c}}$ and realization statement.**

specified by the documentation [41]. The white paper of Corda [9] states three major security goals:

***Partial consistency:*** Whenever parties share some transaction, they agree on the content of the transaction as well as on (contents of) all dependencies. In this work we propose and formalize the novel notion of partial consistency to capture this goal, which is stated only on an intuitive level in the white paper.

***Double spending protection:*** Transaction's output states cannot be spent twice.

***Privacy:*** A transaction between a group of parties is only visible to them and all parties that need to validate this transaction as part of validating another (dependent) transaction in the future.

According to the Corda white paper, these goals should be achieved under the assumption that all notaries behave honestly. Jumping slightly ahead, while some level of trust into notaries is clearly necessary, our analysis refines this requirement by showing that participants enjoy security guarantees as long as they do not rely on a dishonest notary (even if other notaries are dishonest).

## 4.2 Model of Corda in the iUC Framework

Our model $\mathcal{P}^{\mathbf{c}}$ of Corda in the iUC framework closely follows the above description. Formally, $\mathcal{P}^{\mathbf{c}}$ is the protocol $(\texttt{client} \mid \texttt{notary}, \mathcal{F}_{\text{unicast}}, \mathcal{F}_{\text{cert}}, \mathcal{F}_{\text{ro}})$ consisting of a `client` machine that is accessible to other (higher-level) protocols/the environment, an internal `notary` machine, and three ideal subroutines $\mathcal{F}_{\text{unicast}}$, $\mathcal{F}_{\text{cert}}$, and $\mathcal{F}_{\text{ro}}$ modeling secure authenticated channels, certificate based signatures using a EUF-CMA signature scheme, and idealized hash functions respectively (cf. Figure 4). In a run, there can be multiple instance of machines, modeling different participants of the protocol. We consider a static but unbounded number of participants, i.e., clients and notaries. We discuss technical details of our modeling in what follows.

Recall from above that signees are free to agree or decline an incoming transaction, depending on whether their higher-level protocol wants to perform that transaction. We model agreement to a transaction by letting the higher-level protocol submit the transaction (but not its dependencies) to the signee first. Upon receiving a new transaction from an initiator, the signee then checks whether it has previously received the same transaction from the higher-level protocol and accepts or declines accordingly. This modeling is realistic: in practice, the users of the initiator and signee clients would typically have to first agree on some transaction out of band, and can then input this information into the protocol. Since this modeling means that transactions are submitted to both clients in the initiator and the signee roles, we assume w.l.o.g. that transactions indicate which party is supposed to perform the initiation process (e.g., by listing this party first in the list of participants).

In addition to explicit agreement of signees, we also model the process of pushing a transaction to another client. On a technical level, this is modeled via a special submit request that instructs a client to push one of its known transactions to some client with a certain PID. Explicitly modeling agreement of signees and pushing of transactions, instead of assuming that this is somehow done out-of-band, allows for obtaining more realistic privacy results.

A notary in Corda may not just be a single machine but a service distributed across multiple machines. In our modeling, for simplicity of presentation, we model a notary as a single machine. However, the composition theorem of the iUC framework then allows for replacing this single machine with a distributed system that provides the same guarantees, thereby extending our results also to distributed notaries.

All network communication between parties of Corda is via an ideal functionality $\mathcal{F}_{\text{unicast}}$, modeling authenticated secure unicast channels between all participants. This functionality also offers a notion of time and guarantees eventual message delivery, i.e., time may not advance if there is any message that still needs to be delivered and has been sent at least $\delta$ time units ago.

We allow dynamic corruption of clients and notaries. The adversary gains full control over corrupted clients and notaries and can receive/send messages in their name from/to other parts of the protocol/higher-level protocols. While the ideal subroutines are not directly corruptible, the adversary can simply corrupt the client/notary using the subroutine to, e.g., sign messages in the name of that client/notary.

In addition to being explicitly corruptible by the adversary, clients also consider themselves to be (implicitly) corrupted – they set a corruption flag but otherwise follow the protocol honestly – if they know a transaction that relies on (signatures of) a corrupted notary.[5] More specifically, we capture the fact that Corda needs to assume honesty of notaries to be able to provide its security guarantees. Consequently, if a client relies on a corrupted notary, then it cannot obtain the intended security guarantees such as double spending protection anymore. Note that this modeling actually captures a somewhat weaker security assumption than Corda: Corda officially requires *all* notaries to be honest in order to provide security guarantees. Our modeling only assumes that those notaries that a specific client actually relies on are honest, i.e., our analysis shows that security guarantees can be given to clients even in the presence of corrupted notaries as long as these notaries are not used by the clients.

## 4.3 Corda Realizes $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$

In this section, we present our security analysis of Corda. On a high-level, we will show the following security properties for Corda:

***Partial consistency:*** All honest parties read subsets of the same global transaction graph. Hence, for every transaction ID they in particular also agree on the contents and dependencies of the corresponding transaction.

***Double spending protection:*** The global graph, which honest parties read from, does not contain double spending.

***Liveness:*** If a transaction involves honest clients only, then, once it has been approved by all clients, it will end up in the global graph within a bounded time frame. Further, after another bounded time frame, all participating clients will consider this transaction to be part of their own partial view of the local state, i. e., this transaction will be part of the output of read requests from those participants.

***Privacy:*** A dishonest party (or an outside attacker) does not learn the body of a transaction $tx$[6] unless he is involved in $tx$ (e.g., *(i)* because he is an initiator, signee, or the notary of $tx$, or *(ii)* because one of the honest clients who has access to $tx$ pushes $tx$ or a transaction that depends on $tx$ to the dishonest party).

Formally, we first define $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$, an instantiation of $\mathcal{F}_{\text{ledger}}$, which formalizes and enforces the above security properties. This is the first formalization of the novel notion of *partial consistency*. As part of defining this instantiation, we also identify the precise privacy level provided by Corda, including several (partly unexpected) privacy leakages. That is, we define $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ to leak only the information that an attacker on Corda can indeed obtain but not anything else, as discussed at the end of this section. We then show that Corda indeed realizes $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ and discuss why this result implies that Corda itself in fact enjoys the above mentioned properties.

Technically, we define the subroutines of $\mathcal{F}_{\text{ledger}}$ to obtain the instantiation $\mathcal{F}_{\text{ledger}}^{\mathbf{c}} = (\mathcal{F}_{\text{ledger}} \mid \mathcal{F}_{\text{submit}}^{\mathbf{c}}, \mathcal{F}_{\text{read}}^{\mathbf{c}}, \mathcal{F}_{\text{update}}^{\mathbf{c}}, \mathcal{F}_{\text{updRnd}}^{\mathbf{c}}, \mathcal{F}_{\text{init}}^{\mathbf{c}}, \mathcal{F}_{\text{leak}}^{\mathbf{c}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}})$ as described next (cf. Figure 1, the additional subroutine $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ is explained below). We provide formal specifications of the subroutines in $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ in the technical report [22].

In what follows, we call the set of transaction and attachment IDs a party *pid* may have access to in plain its *potential knowledge*. More specifically, the potential knowledge of *pid* includes all transactions from the buffer and global graph that involve only honest clients and which either directly involve *pid*, or which have been pushed to *pid* by another honest party that knows the transaction. In addition, it also contains arbitrary transactions that involve at least one corrupted client, with the exact set of transactions determined by $\mathcal{A}$. We use the term *current knowledge* to describe the set of transactions that a party *pid* currently knows, where we allow $\mathcal{A}$ to determine this set as a growing subset of the potential knowledge.

- $\mathcal{F}_{\text{init}}^{\mathbf{c}}$ is parameterized by a set of participants. It provides this set to $\mathcal{F}_{\text{ledger}}$.
- $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ handles *(i)* transaction and attachment submission, and *(ii)* pushing transactions from one party to another party. In Case *(i)*, $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ ensures that incoming transactions and attachments are valid according to a validation algorithm, a parameter of $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$. If *pid* is the initiator of the transaction, $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ also checks that *pid* can execute the validation, i. e., whether all dependent objects of the transaction are in *pid*'s current knowledge. For valid transactions and attachments, $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ generates an object ID and leaks all meta-information (e.g., involved parties, IDs of dependent objects, ...) to $\mathcal{A}$ plus the length of the transaction/attachment body. If a corrupted party is involved,

[5] Here we use the more general corruption model we proposed in Section 2 to capture the security assumption of honest notaries in Corda. Using this modeling, we do not have to hardwire this assumption explicitly into $\mathcal{F}_{\text{ledger}}$.

[6] We consider the "transaction body" to consist of the bit string contained in the transaction (and which might contain, e.g., inputs for the smart contracts) as well as the bit strings contained in output states (encoding, e.g., assets modeled by those states). We consider everything else to be meta-information of the transaction, including its ID, references to input states and smart contracts, and the set of participants.

then $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ also leaks the body. If a party $pid_a$ (tries to) push a transaction identified by *txID* to a party $pid_b$ (Case *(ii)*), $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ first ensures that all dependencies of *tx* are in the current knowledge of $pid_a$ and, if so, then leaks to $\mathcal{A}$ that $pid_a$ shared *txID* with $pid_b$. From then on, *tx* and all of its dependencies are considered to be part of the potential knowledge of $pid_b$.

- $\mathcal{F}_{\text{update}}^{\mathbf{c}}$ mainly handles updates to the state (proposed by $\mathcal{A}$). The adversary $\mathcal{A}$ can specify a set of IDs of transactions/attachments that have previously been submitted by honest parties and submit a set of transactions/attachments from dishonest parties to extend the transaction graph. $\mathcal{F}_{\text{update}}^{\mathbf{c}}$ ensures that *(i)* all (honest) participants agreed to a transaction, *(ii)* all dependencies are included in the global graph, *(iii)* dishonest transactions are valid, and *(iv)* there is no double spending. If any of the checks fails, the graph update is rejected.
- $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ always enforces local read operations. Upon receiving such a read request for an honest party *pid*, the adversary is expected to provide a subgraph *g* of the global graph. This graph *g* must also be a subset of *pid*'s current knowledge, must be self-consistent, i. e., it must contain at least the previous outputs to *pid*'s read requests, and it must be complete, i. e., the graph *g* contains all dependencies of objects in *g*. Furthermore, if there is a transaction *tx* in the global graph which has an honest initiator, *pid* is a participant, and which has been submitted at least $2\delta$ time units ago, where $\delta$ is a parameter which specifies the network delay, then *tx* must be included in *g*. The graph *g* is then returned as response to the read request. For read requests from corrupted parties $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ returns an empty response. Intuitively, this is because $\mathcal{F}_{\text{submit}}$ and $\mathcal{F}_{\text{leak}}$ already leak all information known to corrupted parties.
- Whenever $\mathcal{A}$ requests to advance time, $\mathcal{F}_{\text{updRnd}}^{\mathbf{c}}$ checks whether a transaction *tx* exists in the buffer where all participants are honest, agreed on the transaction, and the last acknowledgment respectively the initiation (if no signees are involved) was received more than $\omega(tx)$ time units ago.[7] If such a transaction exists, then the time increment request is denied. Otherwise, it is accepted.
- As explained in Section 2, the subroutines of $\mathcal{F}_{\text{ledger}}$ can themselves share other subroutines, e.g, to exchange shared state. We use this feature by adding an additional subroutine $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ which provides an interface for all other $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ subroutines *(i)* to query the potential knowledge of a party, *(ii)* to generate unique IDs, to store them, and to distribute them, and *(iii)* to access transactions/attachments by ID. $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ simplifies the specification as it allows to easily synchronize internal state used for bookkeeping purposes across the subroutines of $\mathcal{F}_{\text{ledger}}$.
- Upon corruption of a client, $\mathcal{F}_{\text{leak}}^{\mathbf{c}}$ computes its potential knowledge and forwards this information to $\mathcal{A}$.
- To capture $\mathcal{P}^{\mathbf{c}}$'s random oracle, the adversary $\mathcal{A}$ is also allowed to query $\mathcal{F}_{\text{update}}^{\mathbf{c}}$ for (new) transaction and attachment IDs.
- To capture that Corda might leak the validity of a transaction, $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ allows the adversary to query the validity of transactions regarding a parties *pid* current state.

Using this instantiation of $\mathcal{F}_{\text{ledger}}$, we can state our main theorem.

Theorem 4.1. *Let $\mathcal{P}^{\mathbf{c}}$ and $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ be as described above. Then, $\mathcal{P}^{\mathbf{c}} \leq \mathcal{F}_{\text{ledger}}^{\mathbf{c}}$.*

Here we provide a proof sketch with the core intuition. We provide the full proof in our technical report [22].

*Sketch.* We show that $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ leaks just enough details for a simulator to internally simulate a *blinded* version of the Corda protocol. As mentioned and discussed at the end of this section, all leakages defined by $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ are indeed necessary for a successful simulation since the same information is also leaked by Corda. Hence, $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ precisely captures the actual privacy level of Corda. As explained above, all meta-information of transactions leak, only transaction bodies stay private. The meta data information already allows to execute all checks in the Corda protocol except for the validity check of the transaction body. For honest participants, we can directly derive the validity of the transaction body from the leakage during transaction submission of the transaction's initiator and use this during the simulation.

Our simulator $\mathcal{S}$ internally simulates a blinded instance of $\mathcal{P}^{\mathbf{c}}$, in the following called $\overline{\mathcal{P}^{\mathbf{c}}}$. During the simulation, $\mathcal{S}$ uses dummy transactions generated from the submission leakage. The dummy transaction is identified by the original transaction ID, contains all leaked data and pads the transaction body such that the dummy version has the same length as the original transaction. As $\mathcal{S}$ can extract the knowledge of honest parties, the transaction graph structure, and the validity of transactions, $\mathcal{S}$ can derive all steps in $\overline{\mathcal{P}^{\mathbf{c}}}$ without having access to the full data. In particular, $\mathcal{S}$ knows for all honest parties which transaction/attachment IDs are in the parties knowledge. This allows it to perfectly simulate all network interaction of $\overline{\mathcal{P}^{\mathbf{c}}}$ as $\mathcal{S}$ knows when a party needs to trigger, e. g., the `SendTransactionFlow` subprotocol instead of directly simulating the approval to a transaction. Further, $\mathcal{S}$ can keep states of honest parties in $\overline{\mathcal{P}^{\mathbf{c}}}$ and $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ synchronous such that read requests lead to the same output in real and ideal world. We observe that the output from $\mathcal{S}$ to $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ never fails. $\overline{\mathcal{P}^{\mathbf{c}}}$ ensures that knowledge does not violate the boundaries of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$, e. g., $\overline{\mathcal{P}^{\mathbf{c}}}$'s build-in network $\mathcal{F}_{\text{unicast}}$ ensures delivery boundaries.

Regarding $\mathcal{S}$ interaction with the network. As corrupted parties send transactions and attachments in plain to $\mathcal{S}$ and $\mathcal{S}$ can evaluate the validity of transactions (according to a parties knowledge), $\mathcal{S}$ has access to all relevant information to answer request/handle operations indistinguishably between $\mathcal{P}^{\mathbf{c}}$ and $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$. This is due to the fact that $\mathcal{S}$ replaces the dummy transaction by the original transaction as soon as they leak (and regenerate dependent data, especially signatures, to make both worlds indistinguishable).

We highlight two edge cases: Firstly, an attacker may try to break privacy of transactions by brute forcing the hashes. As $\mathcal{S}$ queries $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ for IDs, this attack would be successful in both real and ideal world. Secondly, when corrupted parties push arbitrary transactions to honest parties, $\mathcal{S}$ might not know whether the validity check

[7] $\omega(tx)$ is a function that linearly depends on the network delay $\delta$ and the size of the subgraph defined by the transaction *tx* and all of its inputs (including their respective inputs, etc.). Such a function is necessary due to the way parties in Corda retrieve unknown dependencies for transactions.

succeeds (since this transaction might reference input states that the corrupted party and hence $\mathcal{S}$ does not know). In this case (and only in this case), $\mathcal{S}$ directly queries $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ for the validity of the transaction according to the honest party's knowledge. We will discuss both cases in more detail in the following discussion. □

We now discuss the implications of Theorem 4.1 for the security properties of Corda.

***Partial consistency.*** By definition of $\mathcal{F}_{\text{read}}^{\mathbf{c}}$, the responses to read requests of honest parties are subsets of the global graph. This directly implies that honest clients (i. e., clients that are neither controlled by the adversary nor rely on a malicious notary) of Corda obtain consistent partial views of the same global state.

***Double spending protection.*** By definition of $\mathcal{F}_{\text{update}}^{\mathbf{c}}$, the global graph does not contain any double spending. Since this global graph is a superset of read outputs of honest parties (as per $\mathcal{F}_{\text{read}}^{\mathbf{c}}$), this implies that Corda protects honest clients from double spending.

***Liveness.*** $\mathcal{F}_{\text{updRnd}}^{\mathbf{c}}$ guarantees that transactions which involve only honest clients end up in the global graph after an upper bounded delay (once all clients have acknowledged the transaction). Furthermore, $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ ensures that transactions with honest initiators end up in the local state of all honest signees after another bounded time delay. By Theorem 4.1, these properties directly translate to Corda. A stronger liveness statement is not possible for Corda: if a notary is corrupted (and by extension all clients that rely on this notary also consider themselves to be corrupted), then a transaction might never be signed by that notary and hence not enter the global graph. Further, since the initiator is solely responsible for forwarding responses from the notary, such a response might not end up in the local state of a signee if the initiator misbehaves.

***Privacy.*** Privacy needs a bit more explanation than the other properties. Firstly, observe that $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ ensures that honest parties can only read transactions that are part of their potential knowledge, i. e., those they are directly involved in or that have been forwarded to them by someone that already knew the transaction. Furthermore, by definition of $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$, if no dishonest client is involved in a new transaction, only the length of the body is leaked. For Corda, this implies that the body of a transaction that involves only honest clients (and in extension an honest notary) stays secret from everyone, unless one of those clients intentionally forwards the transaction to another party.

We can also derive what a dishonest client or dishonest notary in Corda can learn at most, thereby determining the level of privacy that Corda provides: By definition of $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ and $\mathcal{F}_{\text{leak}}$, all of the metadata of transactions is leaked. In contrast, the message bodies of transactions leak only if they involve a dishonest client. Hence, an adversary on Corda learns at most the metadata of transactions, all transaction bodies that use a dishonest notary, and all transaction bodies that involve a dishonest client. An adversary cannot learn anything else since otherwise the simulation of dishonest clients/notaries would fail, i. e., Theorem 4.1 could not be shown.

We note that Corda indeed leaks (some) meta-information of transactions. This is because an outside adversary can observe the network communication, which in itself strongly depends and changes based on the meta-information of a transaction. For example, the initiator of an honest transaction collects the approvals of all signees, which makes it trivial to derive the set of participating clients. Similarly, the notary is obvious from watching where a transaction is sent by the initiator after collecting approvals from signees. Even the set of inputs to a transaction is partially visible as, e.g., the signees and the notary request missing inputs from the initiator. While we slightly over approximate this information leakage by leaking the full meta-information in $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$, it is not possible to obtain a reasonably stronger privacy statement for meta-information in Corda.

Furthermore, observe that the adversary on $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ is allowed to obtain IDs for arbitrary transactions. This captures that the IDs of transactions in Corda are computed as hashes over the full transaction, including the body of the transaction in plain. Hence, if an attacker gets hold of such an ID, then he can use it to try and brute force the content of the transaction.

Finally, observe that an adversary on $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ is also allowed to validate arbitrary transactions with respect to the current partial view of some honest client, which might in particular leak information about input states. This captures the following attack on Corda: If an adversary is in control of a notary and he knows an ID of a (currently secret) transaction $tx$ from an honest client, then he can create (and let the notary sign) a new transaction $tx'$ that uses one or more output states from the secret transaction $tx$ as input. Now, the adversary can push this transaction via a corrupted client to the honest client, which then verifies the transaction and, depending on whether verification succeeds, adds $tx'$ to his partial view of the global state. Since this is generally observable, the adversary learns the result of the verification, which, depending on the smart contracts involved, might leak parts of $tx$.

We emphasize that both of the above leakages, respectively attacks, on Corda are possible only if an ID of a transaction is leaked by a higher-level protocol, illustrating the importance of the IDs for secrecy. Since we consider arbitrary higher-level protocols (simulated by the environment) in our proof, we cannot circumvent these leakages. However, if we were to consider a specific higher-level protocol, say, $Q$ using Corda/the ideal ledger such that $Q$ keeps the transaction IDs secret (at least for honest parties), then one can actually prove that Corda in this specific context realizes a variant of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ that does not leak transaction IDs, does not give access to a hash oracle, and does not leak verification results. But, again, our results show that this is not true in general.

## ACKNOWLEDGMENTS

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program "Services Computing".[8] This work was also supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through grant KU 1434/13-1, 442893093, and as part of the Research and Training Group 2475 "Cybercrime and Forensic Computing" under grant number 393541319/GRK2475/1-2019 and by the state of Bavaria at the Nuremberg Campus of Technology (NCT).

[8] http://www.services-computing.de/

# REFERENCES


[1] Accenture. 2019. Accenture and SAP Build Prototype that Uses Distributed Ledger Technology to Enable More Efficient, Secure and Reliable Payments Between Banks and Customers. https://newsroom.accenture.com/news/accenture-and-sap-build-prototype-that-uses-distributed-ledger-technology-to-enable-more-efficient-secure-and-reliable-payments-between-banks-and-customers.htm. (Accessed on 05/26/2020).

[2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. 2018. 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, 30:1–30:15.

[3] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2018. 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, 913–930.

[4] Christian Badertscher, Peter Gazi, Aggelos Kiayias, Alexander Russell, and Vassilis Zikas. 2018. Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability. *IACR Cryptology ePrint Archive* 2018 (2018), 378.

[5] Christian Badertscher, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. 2017. 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 (Lecture Notes in Computer Science)*, Vol. 10401. Springer, 324–356.

[6] Leemon Baird. 2016. Hashgraph Consensus: Fair, Fast, Byzantine Fault Tolerance. http://www.swirlds.com/wp-content/uploads/2016/06/2016-05-31-Swirlds-Consensus-Algorithm-TR-2016-01.pdf. (Accessed on 03/06/2020).

[7] BCG. 2019. Digital Ecosystems in Trade Finance. https://image-src.bcg.com/Images/BCG_Digital_Ecosystems_in_Trade_Finance_tcm38-229964.pdf. (Accessed on 05/26/2020).

[8] Mike Brown and Richard Gendal Brown. 2019. Corda: A distributed ledger. https://www.r3.com/reports/corda-technical-whitepaper/. (Accessed on 11/11/2019).

[9] Richard Gendal Brown. 2020. The Corda Platform: An Introduction. https://www.r3.com/wp-content/uploads/2019/06/corda-platform-whitepaper.pdf. (Accessed on 28/05/2020).

[10] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2016. Universal Composition with Responsive Environments. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security (Lecture Notes in Computer Science)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.), Vol. 10032. Springer, 807–840. A full version is available at https://eprint.iacr.org/2016/034.

[11] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2019. 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 (Lecture Notes in Computer Science)*, Vol. 11923. Springer, 191–221. The full version is available at http://eprint.iacr.org/2019/1073.

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

[13] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. 2007. Universally Composable Security with Global Setup. In *Theory of Cryptography, Proceedings of TCC 2007 (Lecture Notes in Computer Science)*, S. P. Vadhan (Ed.), Vol. 4392. Springer, 61–85.

[14] coindesk. 2019. Over 50 Banks, Firms Trial Trade Finance App Built With R3's Corda Blockchain. https://www.coindesk.com/over-50-banks-firms-trial-trade-finance-app-built-with-r3s-corda-blockchain. (Accessed on 06/02/2020).

[15] Phil Daian, Rafael Pass, and Elaine Shi. 2019. Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake. In *Financial Cryptography and Data Security 2019 (LNCS)*, Vol. 11598. Springer, 23–41.

[16] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. 2018. 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 (Lecture Notes in Computer Science)*, Vol. 10821. Springer, 66–98.

[17] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, Julia Hesse, and Kristina Hostáková. 2019. Multi-party Virtual State Channels. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I (Lecture Notes in Computer Science)*, Vol. 11476. Springer, 625–656.

[18] Christoph Egger, Pedro Moreno-Sanchez, and Matteo Maffei. 2019. Atomic Multi-Channel Updates with Constant Collateral in Bitcoin-Compatible Payment-Channel Networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 801–815.

[19] Forbes. 2020. NASDAQ Partnership With Blockchain Firm R3 Is Great For Crypto. https://www.forbes.com/sites/benjessel/2020/05/22/why-nasdaqs-partnership-with-r3-is-great-for-digital-asset-adoption/. (Accessed on 05/26/2020).

[20] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. 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 (Lecture Notes in Computer Science)*, Vol. 9057. Springer, 281–310.

[21] Mike Graf, Ralf Küsters, and Daniel Rausch. 2020. 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*. IEEE, 236–255.

[22] Mike Graf, Daniel Rausch, Christoph Egger, Viktoria Ronge, Ralf Küsters, and Dominique Schröder. 2021. A Security Framework for Distributed Ledgers. *IACR Cryptol. ePrint Arch.* 2021 (2021), 145.

[23] Hewlett Packard Enterprise. 2018. Blockchain unchained. https://www.hpe.com/us/en/newsroom/blog-post/2018/07/blockchain-unchained.html. (Accessed on 05/26/2020).

[24] HM Land Registry. 2018. HM Land Registry to explore the benefits of blockchain. https://www.gov.uk/government/news/hm-land-registry-to-explore-the-benefits-of-blockchain. (Accessed on 05/26/2020).

[25] International Business Times. 2015. Blockchain expert Tim Swanson talks about R3 partnership of Goldman Sachs, JP Morgan, UBS, Barclays et al. https://www.ibtimes.co.uk/blockchain-expert-tim-swanson-talks-about-r3-partnership-goldman-sachs-jp-morgan-ubs-barclays-1519905. (Accessed on 05/26/2020).

[26] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. 2019. Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 157–174.

[27] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. 2020. A Composable Security Treatment of the Lightning Network. In *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 334–349.

[28] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. 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 (Lecture Notes in Computer Science)*, Vol. 10401. Springer, 357–388.

[29] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. 2016. Fair and Robust Multi-party Computation Using a Global Transaction Ledger. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 9666. Springer, 705–734.

[30] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 583–598.

[31] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 839–858.

[32] R. Küsters. 2006. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*. IEEE Computer Society, 309–320. See [34] for a full and revised version.

[33] Ralf Küsters and Daniel Rausch. 2017. A Framework for Universally Composable Diffie-Hellman Key Exchange. In *IEEE 38th Symposium on Security and Privacy (S&P 2017)*. IEEE Computer Society, 881–900.

[34] Ralf Küsters, Max Tuengerthal, and Daniel Rausch. 2020. The IITM model: a simple and expressive model for universal composability. *Journal of Cryptology* 33, 4 (2020), 1461–1584.

[35] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Aravinda Krishnan Thyagarajan, and Jiafan Wang. 2019. Omniring: Scaling Private Payments Without Trusted Setup. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 31–48.

[36] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications*

*Security, Vienna, Austria, October 24-28, 2016*. ACM, 17–30.
[37] McKinsey Digital. 2018. The strategic business value of the blockchain market. https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/blockchain-beyond-the-hype-what-is-the-strategic-business-value. (Accessed on 05/26/2020).
[38] Rafael Pass, Lior Seeman, and Abhi Shelat. 2017. Analysis of the Blockchain Protocol in Asynchronous Networks. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 10211. 643–673.
[39] R3. 2017. R3's Corda Partner Network Grows to Over 60 Companies Including Hewlett Packard Enterprise, Intel and Microsoft. https://www.r3.com/press-media/r3s-corda-partner-network-grows-to-over-60-companies-including-hewlett-packard-enterprise-intel-and-microsoft/. (Accessed on 06/02/2020).
[40] R3. 2020. Corda Source Code. https://github.com/corda/corda. (Accessed on 04/24/2020).
[41] R3. 2020. R3 Corda Master documentation. https://docs.corda.net/docs/corda-os/4.4.html. (Accessed on 04/24/2020).
[42] Reuters. 2015. Nine of world's biggest banks join to form blockchain partnership. https://www.reuters.com/article/us-banks-blockchain/nine-of-worlds-biggest-banks-join-to-form-blockchain-partnership-idUSKCN0RF24M20150915. (Accessed on 05/26/2020).
[43] Shifeng Sun, Man Ho Au, Joseph K. Liu, and Tsz Hon Yuen. 2017. RingCT 2.0: A Compact Accumulator-Based (Linkable Ring Signature) Protocol for Blockchain Cryptocurrency Monero. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 10493. Springer, 456–474.
[44] Digital Asset Canton Team. 2019. Canton: A Private, Scalable, and Composable Smart Contract Platform. https://www.canton.io/publications/canton-whitepaper.pdf. (Accessed on 11/27/2019).
[45] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf. (Accessed on 01/18/2019).
[46] Tsz Hon Yuen, Shifeng Sun, Joseph K. Liu, Man Ho Au, Muhammed F. Esgin, Qingzhao Zhang, and Dawu Gu. 2020. RingCT 3.0 for Blockchain Confidential Transaction: Shorter Size and Stronger Security. In *Financial Cryptography and Data Security - 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10-14, 2020 Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 12059. Springer, 464–483.
[47] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. RapidChain: Scaling Blockchain via Full Sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 931–948.

# APPENDIX:

## A A BRIEF INTRO TO THE IUC FRAMEWORK

This section provides a brief introduction to the iUC framework, which underlies all results in this paper. The iUC framework [11] is a highly expressive and user friendly model for universal composability. It allows for the modular analysis of different types of protocols in various security settings.

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

To define the universal composability security experiment (cf. Camenisch et al. [11]), one distinguishes between three types of systems: protocols, environments, and adversaries. As is standard in universal composability models, all of these types of systems have to meet a polynomial runtime notion . Intuitively, the security experiment in any universal composability model compares a protocol $\mathcal{P}$ with another protocol $\mathcal{F}$, where $\mathcal{F}$ is typically an ideal specification of some task, called *ideal protocol* or *ideal functionality*. The idea is that if one cannot distinguish $\mathcal{P}$ from $\mathcal{F}$, then $\mathcal{P}$ must be "as good as" $\mathcal{F}$. More specifically, the protocol $\mathcal{P}$ is considered secure (written $\mathcal{P} \leq \mathcal{F}$) if for all adversaries $\mathcal{A}$ controlling the network of $\mathcal{P}$ there exists an (ideal) adversary $\mathcal{S}$, called *simulator*, controlling the network of $\mathcal{F}$ such that $\{\mathcal{A}, \mathcal{P}\}$ and $\{\mathcal{S}, \mathcal{F}\}$ are indistinguishable for all environments $\mathcal{E}$. Indistinguishability means that the probability of the environment outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$ is negligibly close to the probability of outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ (written $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$). The environment can also subsume the role of the network attacker $\mathcal{A}$, which yields an equivalent definition in the iUC framework. We usually show this equivalent but simpler statement in our proofs, i.e., that there exists a simulator $\mathcal{S}$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all environments.

A protocol $\mathcal{P}$ in the iUC framework is specified via a system of machines $\{M_1, \ldots, M_l\}$; the framework offers a convenient template for the specification of such systems. Each machine $M_i$ implements one or more roles of the protocol, where a role describes a piece of code that performs a specific task. For example, a (real) protocol $\mathcal{P}_{\text{sig}}$ for digital signatures might contain a `signer` role for signing messages and a `verifier` role for verifying signatures. In a run of a protocol, there can be several instances of every machine, interacting with each other (and the environment) via I/O interfaces and interacting with the adversary (and possibly the environment subsuming a network attacker) via network interfaces. An instance of a machine $M_i$ manages one or more so-called *entities*. An entity is identified by a tuple (*pid*, *sid*, *role*) and describes a specific party with party ID (PID) *pid* running in a session with session ID (SID) *sid* and executing some code defined by the role *role* where this role has to be (one of) the role(s) of $M_i$ according to the specification of $M_i$. Entities can send messages to and receive messages from other entities and the adversary using the I/O and network interfaces of their respective machine instances. More specifically, the I/O interfaces of both machines need to be connected to each other (because one machine specifies the other as a subroutine) to enable communication between entities of those machines.

Roles of a protocol can be either public or private. The I/O interfaces of private roles are only accessible by other (entities belonging to) roles of the same protocol, whereas I/O interfaces of public roles can also be accessed by other (potentially unknown) protocols/the environment. Hence, a private role models some internal subroutine that is protected from access outside of the protocol, whereas a public role models some publicly accessible operation that can be used by other protocols. One uses the syntax "($\texttt{pubrole}_1, \ldots, \texttt{pubrole}_n$ | $\texttt{privrole}_1, \ldots, \texttt{privrole}_n$)" to

uniquely determine public and private roles of a protocol. Two protocols $\mathcal{P}$ and $\mathcal{Q}$ can be combined to form a new more complex protocol as long as their I/O interfaces connect only via their public roles. In the context of the new combined protocol, previously private roles remain private while previously public roles may either remain public or be considered private, as determined by the protocol designer. The set of all possible combinations of $\mathcal{P}$ and $\mathcal{Q}$, which differ only in the set of public roles, is denoted by $\mathsf{Comb}(\mathcal{Q}, \mathcal{P})$.

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

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

Corollary A.1 (Concurrent composition in iUC; informal). *Let $\mathcal{P}$ and $\mathcal{F}$ be two protocols such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another protocol such that $\mathcal{Q}$ and $\mathcal{F}$ can be connected. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ agree on their public roles. Then $\mathcal{R} \leq \mathcal{I}$.*

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

***Notation in Pseudo Code.*** ITMs in our paper are specified in pseudo code. Most of our pseudo code notation follows the notation of the iUC framework as introduced by Camenisch et al. [11]. To ease readably of our figures, we provide a brief overview over the used notation here.

The description in the main part of the ITMs consists of blocks of the form **Recv** ⟨*msg*⟩ **from** ⟨*sender*⟩ **to** ⟨*receiver*⟩, **s.t.** ⟨condition⟩: ⟨code⟩ where ⟨*msg*⟩ is an input pattern, ⟨*sender*⟩ is the receiving interface (I/O or NET), ⟨*receiver*⟩ is the dedicated receiver of the message and ⟨condition⟩ is a condition on the input. ⟨code⟩ is the (pseudo) code of this block. The block is executed if an incoming message matches the pattern and the condition is satisfied. More specifically, ⟨*msg*⟩ defines the format of the message $m$ that invokes this code block. Messages contain local variables, state variables, strings, and maybe special characters. To compare a message $m$ to a message pattern *msg*, the values of all global and local variables (if defined) are inserted into the pattern. The resulting pattern $p$ is then compared to $m$, where uninitialized local variables match with arbitrary parts of the message. If the message matches the pattern $p$ and meets ⟨condition⟩ of that block, then uninitialized local variables are initialized with the part of the message that they matched to and ⟨code⟩ is executed in the context of ⟨*receiver*⟩; no other blocks are executed in this case. If $m$ does not match $p$ or ⟨condition⟩ is not met, then $m$ is compared with the next block. Usually a **recv from** block ends with a **send to** clause of form **send** $\langle \overline{msg} \rangle$ **to** $\langle \overline{sender} \rangle$ where $\overline{msg}$ is a message that is send via output interface $\overline{sender}$.

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

To clarify the presentation and distinguish different types of variables, constants, strings, etc. we follow the naming conventions of Camenisch et al. [11]:

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

***Additional Notation.*** To increase readability, we use the following non-standard notation during the specifications of machines in the iUC template:

- For a set of tuples $K$, $K$.`add`(_) adds the tuple to $K$.
- For a string $S$, $S$.`add`(_) concatenates the given string to $S$.
- $K$.`remove`(_) removes always the first appearance of the given element/string from the list/tuple/set/string $K$.
- $K$.`contains`(_) checks whether the requested element/string is contained in the list/tuple/set/string $K$ and returns either `true` oder `false`.
- We further assume that each element as a tuple in a list or set can be addressed by each element in that tuple if it is a unique key.
- Elements in a tuple are ordered can be addressed by index, starting from 0. We write $[n] = \{1, \ldots, n\}$.
- For tuples, lists, etc. we start index counting at 0.

# B THE IDEAL LEDGER FUNCTIONALITY

In this section, we present the full specification of the ideal ledger functionality $\mathcal{F}_{\text{ledger}}$ in Figure 5.

In what follows, we first briefly explain the iUC notation used in the figures:

- The **CheckID** algorithm is used to determine which machine instance is responsible for and hence manages which entities. Whenever a new message is sent to some entity $e$ whose role is implemented by a machine $M$, the **CheckID** algorithm is run with input $e$ by each instance of $M$ (in order of their creation) to determine whether $e$ is managed by the current instance. The first instance that accepts $e$ then gets to process the incoming message. By default, **CheckID** accepts entities of a single party in a single session, which captures a traditional formulation of a real protocol. Other common definitions include accepting all entities from the same session, which captures a traditional formulation of an ideal functionality.
- The special variable $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ refers to the currently active entity of the current machine instance (that was previously accepted by **CheckID**). If the current activation is due to a message received from another entity, then $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ refers to that entity.
- The special macro $\mathbf{corr}(pid_{sub}, sid_{sub}, role_{sub})$ can be used to obtain the current corruption status (i.e., whether this entity is still honest or considers itself to be implicitly/explicitly corrupted) of an entity belonging to a subroutine.
- Each machine instance in iUC includes the variable CorruptionSet. The set contains all corrupted entities $(pid, sid, role)$ in this instance.
- The iUC framework supports so-called responsive environments and responsive adversaries [10]. Such environments and adversaries can be forced to respond to certain messages on the network, called *restricting messages*, immediately and without first activating the protocol in any other way. This is a useful mechanism for modeling purposes, e.g., to leak some information to the attacker or to let the attacker decide upon the corruption status of a new entity but without disrupting the intended execution of the protocol. Such network messages are marked by writing **send responsively to** instead of just **send to**.
- After sending a message, a machine instance may wait in its current state for an expected answer. The **wait for** command is used to express this. In this "waiting" state, the machine instance does not accept any other input message.
- The symbol "_" is used as a wildcard symbol.

**Handling corruption:** As already indicated, iUC includes a specification for the behavior during corruption of parties and the behavior of corrupted parties (similar to the UC model). Essentially, corrupted parties act as forwarders, i. e., they forward messages dedicated to them to $\mathcal{A}$ and $\mathcal{A}$ may act in behalf of them, i. e., can use the connections of the corrupted party to send messages. In particular, iUC allows to restrict the behavior of $\mathcal{A}$ when impersonating a corrupted party via several parameters. In $\mathcal{F}_{\text{ledger}}$, we use **LeakedData** to specify which data is leaked on corruption of a party. Further, we restrict $\mathcal{A}$ via **AllowAdvMessage** to use any of the subroutines of $\mathcal{F}_{\text{ledger}}$ on behalf of a corrupted party.

More technical details regarding iUC are available in the full version of the paper [22] and the iUC paper [11].

Note that, in addition to what is described in Section 2, $\mathcal{F}_{\text{ledger}}$ as defined in Figure 5 also provides a read interface for the adversary (`CorruptedRead`) on behalf of corrupted parties. This may allow $\mathcal{A}$ to query $\mathcal{F}_{\text{ledger}}$ on behalf of a corrupted party, e. g., to access private data of the party which has not been leaked so far.

## C FURTHER FEATURES OF $\mathcal{F}_{\text{ledger}}$

Here we explain and discuss some features of $\mathcal{F}_{\text{ledger}}$ that were only briefly mentioned in Section 2.

***Roles in*** $\mathcal{F}_{\text{ledger}}$**.** By default, $\mathcal{F}_{\text{ledger}}$ does not distinguish between different roles of participants. Every party is a client with the same read and write access to the ledger, while any additional internal non-client roles, such as miners and notaries, only exist in the realization. If one needs to further differentiate clients into different client roles, e.g., to capture that in a realization certain clients can read only part of the global transaction list while others can read the full list, then this can be done via a suitable instantiation of the subroutines of $\mathcal{F}_{\text{ledger}}$ – such client-roles can easily be added as prefixes within PIDs. The subroutines that specify security properties, such as $\mathcal{F}_{\text{read}}$, can then depend on this prefix and, e.g., offer a more or less restricted access to the global transaction list.

***Dynamic party registration.*** The ideal functionality $\mathcal{F}_{\text{ledger}}$ keeps track of all currently registered honest parties, including the time when they registered. An honest party is considered registered once it issues its first read or write request, modeling that participants in a distributed ledger first register themselves before interacting with the ledger. A higher-level protocol can also deregister a party by sending a deregister command. Such a party is removed from the set of registered parties (and will be added again with a new registration time if it ever issues another read or write request).

This mechanism allows for capturing security properties that depend on the (time of) registration. For example, an honest party might only obtain consistency guarantees after it has been registered for a certain amount of time (due to network delays in the realization). We note that, just like a clock, party registration is an entirely optional concept that can be ignored by not letting any subroutines depend on this information. This is useful to capture realizations that, e.g., do not model an explicit registration phase but rather assume this information to be static and fixed at the start of the protocol run.

***Public and private ledgers.*** Existing functionalities for blockchains have so far been modeled as so-called global functionalities using the GUC extension [13] of the UC model. The difference between a global and a normal/local ideal functionality is that, when a global functionality is used as a subroutine of a higher-level protocol, then also the environment/arbitrary other (unknown) protocols running in parallel can access and use the same subroutine. This is often the most reasonable modeling for public blockchains: here, the same blockchain can be accessed by arbitrarily many higher-level protocols running in parallel. However, such global functionalities do not allow for capturing the case of, e.g., a permissioned blockchain that is used only within a restricted context. This situation rather corresponds to a local ideal blockchain functionality.

The iUC framework that we use here provides seamless support for both local and global functionalities, and in particular allows for arbitrarily changing one to the other. Hence, our functionality $\mathcal{F}_{\text{ledger}}$ can be used both as a global or as a local subroutine for higher-level protocols, allowing for faithfully capturing both public

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

**Participating roles:** {client}
**Corruption model:** *dynamic corruption*

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

**Implemented role(s):** {client}
**Subroutines:** $\mathcal{F}_{\text{submit}}$ : submit, $\mathcal{F}_{\text{update}}$ : update, $\mathcal{F}_{\text{read}}$ : read, $\mathcal{F}_{\text{updRnd}}$ : updRnd, $\mathcal{F}_{\text{init}}$ : init, $\mathcal{F}_{\text{leak}}$ : leak
**Internal state:**

- identities $\subset \{0,1\}^* \times \mathbb{N}$, identities $= \emptyset$ {*The set of participants and the round when they occured first.*
- round $\in \mathbb{N}_{\geq 0}$, round $= 0$ {*Current (network) round in the protocol execution.*
- msglist $\subset \mathbb{N} \times \mathbb{N} \times \{\texttt{tx}, \texttt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, msglist $= \emptyset$. {*(Totally ordered) sequence of recorded messages that is considered as stable/immutable of the form* (*id*, *commitRound*, *type*, *msg*, *submitRound*, *pid*)*. If type* = meta, *pid* = *submitRound* = $\perp$.
- requestQueue $\subset \mathbb{N} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, requestQueue $= \emptyset$ {*The list of so far not ordered, honest, incoming "transactions". Format* (*tmpCtr*, *tx*, *submittingRound*, *submittingParty*).
- readQueue $\subset \{0,1\}^* \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^*$, readQueue $= \emptyset$, {*The queue of read responses that need to be delivered* (*pid*, *responseId*, *round*, *msg*)
- readCtr $\in \mathbb{N}$, readCtr $= 0$, {readCtr *is temporary ID for transactions in the* readQueue.
- reqCtr $\in \mathbb{N}$, reqCtr $= 0$, {reqCtr *are temporary IDs for transactions in the* requestQueue.

In the following, we pass through the complete internal state of $\mathcal{F}_{\text{ledger}}$ to its subroutines. Thus, we use the variable internalState as follows: internalState $\leftarrow$ (identities, round, msglist, requestQueue, readQueue, $\delta$, CorruptionSet, transcript)
We often use the CorruptionSet as specified in [11]. We often write *pid* $\in$ CorruptionSet instead of (*pid*, $\text{sid}_{\text{cur}}$, $\text{role}_{\text{cur}}$) $\in$ CorruptionSet for brevity.

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

**Corruption behavior:**

- **LeakedData**(*pid*, *sid*, *role*):
  **if** $\exists$(*pid*, *registrationRound*) $\in$ identities, *registrationRound* $\in \mathbb{N}$:
  identities.remove(*pid*, *registrationRound*)
  **send** (corrupt, *pid*, *sid*, internalState) **to** ($\text{pid}_{\text{cur}}$, $\text{sid}_{\text{cur}}$, $\mathcal{F}_{\text{leak}}$ : leak)
  **wait for** (corrupt, *leakage*)
  **return**(*leakage*) {*Depending on the desired properties of* $\mathcal{F}_{\text{ledger}}$*, output after corruption needs to be specified*
- **AllowAdvMessage**(*pid*, *sid*, *role*, $pid_{\texttt{receiver}}$, $sid_{\texttt{receiver}}$, $role_{\texttt{receiver}}$, *m*): $\mathcal{A}$ is not allowed to call subroutines on behalf of a corrupted party.

**Initialization:**

**send** InitMe **to** ($\text{pid}_{\text{cur}}$, $\text{sid}_{\text{cur}}$, $\mathcal{F}_{\text{init}}$ : init) {$\mathcal{F}_{\text{init}}$ *handels initilaization if necessary.*
**wait for** (Init, *identities*, *msglist*, *corrupted*, *leakage*) **s.t.**
1. *identities* $\subset \{0,1\}^* \times \{0\}$, *round* $\in \mathbb{N}$, *msglist* $\subset \mathbb{N} \times \mathbb{N} \times \{\texttt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*\}$,
2. *corrupted* $\subset \{0,1\}^* \times \{\text{sid}_{\text{cur}}\} \times \{\texttt{client}\}$,
3. *msg* $\in$ *msglist* are consecutivley enumarted started at 0,
4. $\exists(\_,\_,\_,\_,a,b) \in$ *msglist*, **s.t.** $a \neq \perp \vee b \neq \perp$

identities $\leftarrow$ *identities*, msglist $\leftarrow$ *msglist*, CorruptionSet $\leftarrow$ *corrupted* {*We enforce correct formats and that* msglist *is a total ordered sequence.*
**send responsively** (Init, *leakage*) **to** NET {*Send leaked information from initilaization to* $\mathcal{A}$.
**wait for** ack **from** NET

**MessagePreprocessing:**

**recv** ($\text{pid}_{\text{cur}}$, $\text{sid}_{\text{cur}}$, $\text{role}_{\text{cur}}$, *msg*) **from** I/O:
**if** ($\text{pid}_{\text{cur}}$, _) $\notin$ identities $\wedge$ *msg* starts with Submit or Read:
{*Register unknown party before its first submit/read operation*
identities.add($\text{pid}_{\text{cur}}$, round)

**Figure 5: The ideal ledger functionality $\mathcal{F}_{\text{ledger}}$ (Part 1).**

and private subroutine ledgers. This is possible without proving any of the realizations again, i.e., once security of a specific realization has been shown, this can be used in both a public and private context. As already explained at the beginning of this section, it is also possible to instantiate subroutines of $\mathcal{F}_{\text{ledger}}$ in such a way that they also are (partially) globally accessible, e.g., to provide a global random oracle to other protocols. This can be done even in cases where $\mathcal{F}_{\text{ledger}}$ itself is used as a private subroutine.

***Modelling smart contracts.*** We also note that $\mathcal{F}_{\text{ledger}}$ fully supports capturing smart contracts, if needed. Typically, smart contracts are modeled by fixing some arbitrary programming language for specifying those smart contracts as a parameter of $\mathcal{F}_{\text{ledger}}$ (the security analysis is then performed for an arbitrary but fixed parameter which makes the security result independent of a specific smart contract language). Smart contracts are then simply bit strings which are interpreted by the subroutines $\mathcal{F}_{\text{submit}}$, $\mathcal{F}_{\text{update}}$, $\mathcal{F}_{\text{read}}$, etc.

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

**Main:**

**recv** (Submit, *msg*) **from** I/O: {*Submission request from a honest identity, see Figure 2 for details.*

**recv** (Update, *msg*) **from** NET: {*Update triggered by the adversary, see Figure 2 for details.*

**recv** (Read, *msg*) **from** I/O: {*Read request from an honest identity, see Figure 2 for details.*

**recv** (DeliverRead, *readCtr*, *suggestedOutput*) **from** NET **s.t.** (*pid*, *readCtr*, *r*, *msg*) ∈ readQueue: {*See Figure 2 for details.*

**recv** (CorruptedRead, *pid*, *msg*) **from** NET **s.t.** *pid* ∈ CorruptionSet: {*Read request from a corrupted identity.*
  **send** (CorruptedRead, *pid*, *msg*, internalState) **to** (*pid*, $\mathsf{sid}_{\mathsf{cur}}$, $\mathcal{F}_{\mathsf{read}}$ : read) {*Forward the request to* $\mathcal{F}_{\mathsf{read}}$
  **wait for** (FinishRead, *leakage*)
  **send** (Read, *pid*, *leakage*) **to** NET {*Forwarded data to* $\mathcal{A}$.

**recv** UpdateRound **from** NET: {$\mathcal{A}$ *triggers round update if current round satisfies rules of* $\mathcal{F}_{\mathsf{updRnd}}$.
  **send** (UpdateRound, internalState) **to** ($\mathsf{pid}_{\mathsf{cur}}$, $\mathsf{sid}_{\mathsf{cur}}$, $\mathcal{F}_{\mathsf{updRnd}}$ : updRnd)
  **wait for** (UpdateRound, *response*, *leakage*)
  **if** *response* = true:
    round ← round + 1
  **reply** (UpdateRound, *response*, *leakage*)

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

**recv** DeRegister **from** I/O: {*De-register honest party*
  Remove the unique tuple ($\mathsf{pid}_{\mathsf{cur}}$, *r*) from identities
  **send responsively** DeRegister **to** NET {*Inform* $\mathcal{A}$ *on the deregistration*
  **wait for** ack
  **reply** DeRegister

**Figure 6: The ideal ledger functionality $\mathcal{F}_{\mathsf{ledger}}$ (Part 2).**

according to the fixed smart contract programming language. While interpreting a smart contract, these subroutines can then enforce additional security properties as desired, e.g., they might ensure that all smart contracts added to the global state are indeed well defined (according to the fixed programming language) and/or that running the smart contracts yields the correct results as specified in some transaction.

We use this concept to model smart contracts in our Corda case study. Here, our subroutines of $\mathcal{F}_{\mathsf{ledger}}$ (as part of transaction validation) guarantee the property of correct execution of smart contracts, i.e., the output states of transactions were indeed computed by running the referenced smart contracts correctly. As stated above, our security analysis of Corda treats the programming language as an arbitrary parameter and hence our results show that Corda provides correct execution of smart contracts independently of the chosen smart contract programming language as long as all participants agree on the same language.

We note that, if the algorithm used by smart contracts can be provided externally by the adversary/environment, then the execution of smart contracts in $\mathcal{F}_{\mathsf{ledger}}$ needs to be upper bounded by some polynomial in order to preserve the polynomial runtime of the ideal functionality as required for composition. Observe, however, that most if not all distributed ledgers in reality, including Corda, already hard code such a polynomial upper bound into their protocol to prevent malicious clients form creating smart contracts with exponential (or worse) runtime. The same bound can be used for $\mathcal{F}_{\mathsf{ledger}}$.

## D FURTHER DETAILS $\mathcal{G}_{\mathsf{ledger}}$ REALIZES $\mathcal{F}_{\mathsf{ledger}}$

In this section, we provide a detailed explanation the instantiation of $\mathcal{F}_{\mathsf{ledger}}$ to cover the ideal blockchain functionality $\mathcal{G}_{\mathsf{ledger}}$. Further, we provide an additional comparison between $\mathcal{G}_{\mathsf{ledger}}$ and $\mathcal{F}_{\mathsf{ledger}}$. In our technical report [22], we provide the full formal specifications of all machines and a formal proof of Theorem 3.1.

Our ideal functionality $\mathcal{F}_{\mathsf{ledger}}$ is in the spirit of and adopts some of the underlying ideas from the existing ideal blockchain functionality $\mathcal{G}_{\mathsf{ledger}}$. As a result, both functionalities share similarities at a high level. More specifically, $\mathcal{G}_{\mathsf{ledger}}$ also offers a writing and reading interface for parties. It is parameterized with several algorithms validate, extendPolicy, Blockify, and predictTime that have to be instantiated by a protocol designer to capture various security properties. By default, $\mathcal{G}_{\mathsf{ledger}}$ provides only the security property of consistency. An honest party can submit a transaction to $\mathcal{G}_{\mathsf{ledger}}$. If this transaction is valid, as decided by the validate algorithm, then it is added to a buffer list. $\mathcal{G}_{\mathsf{ledger}}$ has a global list of blocks containing transactions. This list is updated (based on a bit string that the adversary has previously provided) in a preprocessing phase of honest parties. More specifically, whenever an honest party activates $\mathcal{G}_{\mathsf{ledger}}$, the extendPolicy algorithm is executed to decide whether new blocks are appended to the global list of blocks, with the Blockify algorithm defining the exact format of those new blocks. Then the validate algorithm is called to remove all transactions from the buffer that are now, after the update of the global blockchain, considered invalid. An honest party can then read from the global blockchain. More specifically, if the honest party has been registered for a sufficiently long amount of time (larger than parameter $\delta$), then it obtains a prefix of the chain that contains all but the last at most windowSize $\in \mathbb{N}$ blocks. This captures the security property of consistency. In addition to these basic operations, $\mathcal{G}_{\mathsf{ledger}}$ also supports dynamic (de-)registration of parties and offers a clock, modeled via a subroutine $\mathcal{G}_{\mathsf{clock}}$, that is advanced by $\mathcal{G}_{\mathsf{ledger}}$ depending on the output of the predictTime algorithm (and some additional constraints).

While there are many similarities, there are also several key differences between $\mathcal{G}_{\text{ledger}}$ and our functionality $\mathcal{F}_{\text{ledger}}$:

- $\mathcal{G}_{\text{ledger}}$ requires all transactions to be arranged in "blocks" (generated via the Blockify algorithm) and then always provides the security property of consistency for those blocks. As already explained in Section 2, these are strictly stronger requirements than the ones from $\mathcal{F}_{\text{ledger}}$, which only require the existence of a global ordered list of transactions. In particular, many distributed ledgers, such as Corda, are not designed to generate blocks or to provide consistency, and hence, cannot realize $\mathcal{G}_{\text{ledger}}$.
- While $\mathcal{G}_{\text{ledger}}$ already includes several parameters to customize security properties, there are no parameters for customizing the reading operation. Hence, $\mathcal{G}_{\text{ledger}}$ cannot capture access and privacy security properties for transactions in a blockchain (as all honest participants can always read a full prefix of the chain).[9]
- The view $\mathcal{G}_{\text{ledger}}$ provides to higher-level protocols is lower level and closer to the envisioned realization than the one of $\mathcal{F}_{\text{ledger}}$. In particular, $\mathcal{G}_{\text{ledger}}$ includes an additional operation `MaintainLedger` which has to be called by a higher-level protocol in order to allow time to advance, modeling that a higher-level protocol has to regularly and manually trigger mining operations (or some similar security relevant tasks) for security to hold true. Similarly, the clock used by $\mathcal{G}_{\text{ledger}}$ prevents any time advances unless all parties have notified the clock to allow for time to advance, again forcing a higher-level protocol to manually deal with this aspect.
- While $\mathcal{G}_{\text{ledger}}$ includes a predictTime parameter to customize advancing time, this parameter is actually more restricted than the one from $\mathcal{F}_{\text{ledger}}$: the predictTime can depend only on the set of activations from honest parties but not, e.g., the global state or buffer list of transactions.

As can be seen from the above list, the main differences between $\mathcal{G}_{\text{ledger}}$ and $\mathcal{F}_{\text{ledger}}$ are due to *(i)* different levels of abstraction on the I/O interface to higher-level protocols and *(ii)* the fact that $\mathcal{G}_{\text{ledger}}$ is tailored towards publicly accessible blockchains. Hence, intuitively, it should be possible to show that $\mathcal{F}_{\text{ledger}}$ is a generalization of $\mathcal{G}_{\text{ledger}}$. Indeed, one can instantiate $\mathcal{F}_{\text{ledger}}$ appropriately to transfer security properties provided by $\mathcal{G}_{\text{ledger}}$ to the level of $\mathcal{F}_{\text{ledger}}$.

Formally, we define the instantiation $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ as the protocol $(\mathcal{F}_{\text{ledger}} \mid \mathcal{F}_{\text{init}}^{\mathcal{G}_{\text{ledger}}}, \mathcal{F}_{\text{submit}}^{\mathcal{G}_{\text{ledger}}}, \mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}, \mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}, \mathcal{F}_{\text{updRnd}}^{\mathcal{G}_{\text{ledger}}}, \mathcal{F}_{\text{leak}}^{\mathcal{G}_{\text{ledger}}})$. The general idea for the instantiated subroutines is to run the same operations as $\mathcal{G}_{\text{ledger}}$, including the parameterized algorithms of $\mathcal{G}_{\text{ledger}}$ that determine the precise security properties provided by the global transaction list. By this, the instantiation $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$, just as $\mathcal{G}_{\text{ledger}}$, enforces the security property of consistency for all participants while also inheriting all further security properties provided for the global state, if any, from the parameterized algorithms. More specifically:

- $\mathcal{F}_{\text{init}}^{\mathcal{G}_{\text{ledger}}}$ is defined to run the extendPolicy algorithm to generate the initial transaction list (that is read from the blocks output by the algorithm). This is because extendPolicy might already generate a genesis block during the preprocessing of the first activation of the functionality, before any transactions have even been submitted.
- $\mathcal{F}_{\text{submit}}^{\mathcal{G}_{\text{ledger}}}$ executes the validate algorithm to check validity of incoming transactions.
- $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ executes the extendPolicy and Blockify algorithms to generate new blocks from the update proposed by the adversary. These blocks are transformed into individual transactions, which are appended to the global transaction list of $\mathcal{F}_{\text{ledger}}$ together with a special meta transaction that indicates a block boundary. Additionally, the validate algorithm is used to decide which transactions are removed from the transaction buffer.
- $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}$ checks whether a party has already been registered for an amount of time larger than $\delta$ and then either requests the adversary to provide a pointer to a transaction within the last windowSize blocks or lets the adversary determine the full output of the party. We note that $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}$ has to always use non-local reads: this is because a read operation in $\mathcal{G}_{\text{ledger}}$ might change the global state during the preprocessing phase and before generating an output, i.e., read operations are generally not immediate (in the sense defined in Section 2).
- If the parameters of $\mathcal{G}_{\text{ledger}}$ are such that they guarantee the property of *liveness*, then $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}_{\text{ledger}}}$ can be defined to also encode this property (cf. Section 2); similarly for the time dependent property of *chain-growth* and other time-related properties.
- $\mathcal{F}_{\text{leak}}^{\mathcal{G}_{\text{ledger}}}$ does not leak (additional) information as all information is leaked during submitting and reading.

There are, however, some technical details one has to take care of in order to implement this high-level idea, mostly due to some conceptual differences in and the higher abstraction level of $\mathcal{F}_{\text{ledger}}$. More specifically:

- A key technical difference between $\mathcal{F}_{\text{ledger}}$ and $\mathcal{G}_{\text{ledger}}$ is that updates to the global state in $\mathcal{F}_{\text{ledger}}$ are explicitly triggered by the adversary, whereas $\mathcal{G}_{\text{ledger}}$ performs those updates automatically during a preprocessing phase whenever an honest party activates the functionality, before then processing the incoming request of that party.
  As a result of this formulation, both read and submit requests might change the global transaction list in $\mathcal{G}_{\text{ledger}}$ before the request is answered. In the case of $\mathcal{F}_{\text{ledger}}$, this means the simulator has to be given the option to update the global state *before* a read/submit request is performed. In the case of read requests, this directly matches the properties of non-local read requests, i.e., we simply have to define $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}$ in such a way that it uses non-local reads only. Such non-local reads then enable the simulator to first update the global state of $\mathcal{F}_{\text{ledger}}$ before then finishing the read request, which directly matches the behavior of $\mathcal{G}_{\text{ledger}}$.
  In the case of submit requests, $\mathcal{F}_{\text{ledger}}$ does not directly include a mechanism for updating the state before processing the request. This is because, for realistic distributed ledger protocols, an incoming submit request that has not even been processed and shared with the network yet will not cause any changes to the

[9] This aspect is actually one of the key differences between $\mathcal{G}_{\text{ledger}}$ and its variant $\mathcal{G}_{\text{PL}}$ for privacy in blockchains: the latter also introduces a parameter for read operations.

global state. This, however, might technically occur in $\mathcal{G}_{\text{ledger}}$ depending on how its parameters, such as the extendPolicy and validate algorithms, are instantiated. We could address this by limiting the set of parameters of $\mathcal{G}_{\text{ledger}}$ to those that update the global state independently of (the content of) future submit requests, which matches the behavior of realistic ledger protocols from practice. Nevertheless, since we want to illustrate the generality of $\mathcal{F}_{\text{ledger}}$, we choose a different approach.

To model that the global transaction list might change depending on and before processing a new submit request, we define $\mathcal{F}_{\text{submit}}^{\mathcal{G}\text{ledger}}$ such that it internally first performs an update of the global state, based on some information requested from the simulator via a restricting message, before then validating the incoming transaction. Since $\mathcal{F}_{\text{submit}}^{\mathcal{G}\text{ledger}}$ cannot actually apply this update itself (as this operation is limited to $\mathcal{F}_{\text{update}}^{\mathcal{G}\text{ledger}}$ when it is triggered by update requests from the adversary), the update is then cached in the subroutine $\mathcal{F}_{\text{update}}$. The adversary is forced to apply this cached update first whenever he wants to further update the global transaction list, advance time, or perform a read request. This formulation provides the simulator with the necessary means to update the global state before an incoming submit request, if necessary, while not weakening the security guarantees provided by $\mathcal{F}_{\text{ledger}}$ compared to $\mathcal{G}_{\text{ledger}}$. In particular, read requests will always be answered based on the most recent update of the state, including any potentially cached updates.

- Due to a lower level of abstraction, the parameterized algorithms used in $\mathcal{G}_{\text{ledger}}$ take some inputs that are not directly included in $\mathcal{F}_{\text{ledger}}$, such as a list of all honest activations and a future block candidate (which is an arbitrary message provided by the adversary at some point in the past). We could in principle add the same parameters to subroutines in $\mathcal{F}_{\text{ledger}}$, i.e., essentially encode the full state and logic of $\mathcal{G}_{\text{ledger}}$ within our instantiations of subroutines. Observe, however, that a higher-level protocol generally does not care about (security guarantees provided for) technical details such as cached future block candidates or lists of honest activations. A higher-level protocol only cares about the security properties that are provided by the global transaction list, such as consistency, double spending protection, and liveness.[10] Such security properties can already be defined based on the information that is included in $\mathcal{F}_{\text{ledger}}$ by performing suitable checks on the global transaction list, buffer list, and current time. In particular, it is not actually necessary to include further technical details such as a list of honest activations. This is true even if a security property within a realization (of $\mathcal{G}_{\text{ledger}}$ or $\mathcal{F}_{\text{ledger}}$) actually also depends on, say, the number of honest activations. Such a realization can still realize an ideal functionality that requires, e.g., consistency to always hold true independently of the number of honest activations: one can force the environment to always activate a sufficient number of honest parties within each time frame, modeling a setup assumption that is required for security to holds. This is a common technique that has already been used, e.g., for analyzing Bitcoin [5, 20], including an analysis based on $\mathcal{G}_{\text{ledger}}$. Alternatively to limiting the environment, parties can simply consider themselves to be corrupted if the environment did not activate a sufficient number of honest parties, modeling that they cannot provide any security guarantees such as consistency once the environment violates the setup assumptions. This modeling technique is novel in the field of distributed ledgers and blockchains. We use this technique in our modeling of Corda (cf. Section 4.2).

  Hence, in the spirit of abstraction and simplification, we choose not to include further technical details of $\mathcal{G}_{\text{ledger}}$ in $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$ but rather use the following mechanism to deal with any additional inputs to parameterized algorithms such as the algorithm extendPolicy: Whenever one of the parameterized algorithms from $\mathcal{G}_{\text{ledger}}$ is run within $\mathcal{F}_{\text{ledger}}$, the adversary provides any missing inputs that are not defined in $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$, such as the next block candidate variable for the extendPolicy algorithm. By this definition, the adversary can freely determine technical details that are present only in $\mathcal{G}_{\text{ledger}}$ while $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$ still inherits all properties that are enforced for the global transaction list, buffer list, and/or are related to time.
- The functionality $\mathcal{G}_{\text{ledger}}$ is also parameterized with an algorithm predictTime which determines, based on the set of activations by honest parties, whether time advances. While we could also add this algorithm into $\mathcal{F}_{\text{ledger}}$, more specifically into $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}\text{ledger}}$, by the same reasoning as above a higher-level protocol is typically not interested in this property: it has not implications for the security properties of the global transaction list. Hence, we chose not to include this additional restriction of the adversary via the predictTime in $\mathcal{F}_{\text{ledger}}$.

  However, if the parameters of $\mathcal{G}_{\text{ledger}}$ are such that a certain time-related security property of the global transaction/block list is met, then $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}\text{ledger}}$ enforces the same properties, i.e., prevents the adversary from advancing time unless all properties are met. We exemplify this for the common security properties of *liveness* and *chain-growth*. That is, we include parameters into $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}\text{ledger}}$ that, when they are set, enforce one or both of these security properties, and then show that this can be realized as long as $\mathcal{G}_{\text{ledger}}$ is instantiated in such a way that it also provides these security properties. Clearly the same mechanism can also be used for capturing arbitrary other time-related security properties.
- There are some slight differences in the format of transactions and the global state between $\mathcal{F}_{\text{ledger}}$ and $\mathcal{G}_{\text{ledger}}$, with the key difference being that the global state of $\mathcal{G}_{\text{ledger}}$ is a list of blocks, whereas the global state in $\mathcal{F}_{\text{ledger}}$ is a list of individual transactions. We therefore require the existence of an efficient invertible function toMsglist that maps the output of the Blockify algorithm to a list of transactions contained in that algorithm. Note that such an algorithm always exists: for natural definitions of Blockify that are used by reasonable blockchains, there will always be a list of well-formed transactions encoded into each block. For artificial definitions of Blockify that do not provide outputs which can be mapped to a reasonable definition of

[10] We consider the standard definition of liveness in distributed ledger, resp. blockchain, context [20, 38]: A transaction casted by an honest client should become part of the ledger after some (known) upper time boundary.

a list of transactions, one can always interpret the full block as a single transaction. In addition, we store the end of each block as a special meta transaction in the global transaction list of $\mathcal{F}_{\text{ledger}}$, so one can define still identify the boundaries of individual blocks. This is necessary for lifting the security properties of consistency from $\mathcal{G}_{\text{ledger}}$ to $\mathcal{F}_{\text{ledger}}$, namely, honest users (that have already been registered for a sufficiently long time) are guaranteed obtain a prefix of the global transaction list except for at most the last $\mathsf{windowSize} \in \mathbb{N}$ blocks.

As already explained, we want to show that $\mathcal{G}_{\text{ledger}}$ realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$. Since $\mathcal{G}_{\text{ledger}}$ has a slightly different interface and works on a lower abstraction level than $\mathcal{F}_{\text{ledger}}$, we also have to add a wrapper $\mathcal{W}_{\text{ledger}}$ on top of $\mathcal{G}_{\text{ledger}}$ that transforms the interface and lifts the abstraction level to the one of $\mathcal{F}_{\text{ledger}}$. On a technical level, $\mathcal{W}_{\text{ledger}}$ acts as a message forwarder between the environment and $\mathcal{G}_{\text{ledger}}/\mathcal{G}_{\text{clock}}$ that translates message formats between those of $\mathcal{F}_{\text{ledger}}$ and those of $\mathcal{G}_{\text{ledger}}$ while also taking care of some low level operations that are not present on $\mathcal{F}_{\text{ledger}}$. More specifically:

- Incoming submit and read requests are simply forwarded by the wrapper.
- The output to read requests provided by $\mathcal{G}_{\text{ledger}}$ is in the form of a list of blocks. $\mathcal{W}_{\text{ledger}}$ uses the toMsglist function mentioned above to translate these blocks to a list of transactions to match the format of outputs for read requests from $\mathcal{F}_{\text{ledger}}$.
- Time in $\mathcal{G}_{\text{ledger}}$ is modeled via a separate subroutine $\mathcal{G}_{\text{clock}}$, whereas $\mathcal{F}_{\text{ledger}}$ includes all time management operations in the same functionality. Hence, the wrapper is also responsible to answering requests for the current time, which it does by forwarding those requests to the subroutine $\mathcal{G}_{\text{clock}}$ of $\mathcal{G}_{\text{ledger}}$ and then returning the response.
- As mentioned, the functionality $\mathcal{G}_{\text{ledger}}$ includes a maintenance operation `MaintainLedger` that can be performed by higher-level protocol and which models, e.g., a mining operation that must be performed in a realization. In contrast, $\mathcal{F}_{\text{ledger}}$ does not include such an operation as higher-level protocols typically do not want to explicitly perform mining, but rather expect such operations to be performed automatically "under the hood" of the protocol. This also matches how ideal blockchain functionalities have been used in the literature so far: we are not aware of a higher-level protocol that uses an ideal blockchain functionality and which manually takes care of, e.g., triggering mining operations. This is true even for [27], where a higher-level protocol was built directly on top of $\mathcal{G}_{\text{ledger}}$. That protocol simply assumes that the environment takes care of triggering `MaintainLedger` via a direct connection from the environment to $\mathcal{G}_{\text{ledger}}$.
  The wrapper resolves this mismatch by allowing the adversary on the network to freely perform `MaintainLedger` operations, also for honest parties, modeling that parties might or might not execute a mining operation. This models that parties automatically perform mining without first waiting to receive an explicit instruction from a higher-level protocol to do so. Since the exact set of parties which performing mining operations is determined by the network adversary, this safely over approximates all possible cases that can occur in reality.
  Note that this change actually does not alter or weaken the security statement of $\mathcal{G}_{\text{ledger}}$. Without a wrapper, $\mathcal{G}_{\text{ledger}}$ already allows the environment to perform (or not perform at all) arbitrary `MaintainLedger` operations for both honest and dishonest parties. Hence switching this power from the environment to the adversary on the network provides the same overall security statement. The only difference is that now the operation is indeed performed "under the hood" of the protocol, i.e., a higher-level protocol need not care about manually performing this operation anymore. This also matches how $\mathcal{G}_{\text{ledger}}$ was used by a higher-level protocol in [27] (see above).
- Registration of both honest and corrupted parties in $\mathcal{G}_{\text{ledger}}$ (and the clock $\mathcal{G}_{\text{clock}}$) must be handled manually by higher-level protocols. In contrast, $\mathcal{F}_{\text{ledger}}$ considers an honest party to be registered once it performs the first operation, modeling that a party automatically registers itself before interacting with the ledger, while not including a registration mechanism for dishonest parties. The former is because higher-level protocols typically expect registration, if even required, to be handled "under the hood", while the latter is because a list of registered dishonest parties generally is not necessary to define expected security properties for the global transaction list (this follows the same reasoning given above on why we did not include certain technical details from $\mathcal{G}_{\text{ledger}}$ in $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$).
  To match this behavior, $\mathcal{W}_{\text{ledger}}$ also automatically registers honest parties in both $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{clock}}$ when they receive their first request from a higher-level protocol. For dishonest parties, $\mathcal{W}_{\text{ledger}}$ keeps the original behavior of $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{clock}}$, i.e., the network adversary can freely register dishonest parties.
- The subroutine $\mathcal{G}_{\text{clock}}$ requires all registered parties to notify the clock during each time unit before time can advance, modeling that every party must have been able to perform some computations during each time unit. Following the same reasoning as for the `MaintainLedger` operation, this is a detail that higher-level protocols typically expect to be managed "under the hood" of the protocol and generally do not want to manually take care of. For this reason, this restriction is not included in $\mathcal{F}_{\text{ledger}}$.[11] The wrapper uses the same mechanism as for `MaintainLedger` operations to map between both abstraction levels, i.e., the adversary on the network can freely instruct parties to notify the clock $\mathcal{G}_{\text{clock}}$ that time may advance. Again, this safely over approximates all possible cases in reality while not giving the environment any more power than it already has.

[11] We note that, if desired, this restriction could easily be added to $\mathcal{F}_{\text{ledger}}$ via a suitable instantiation of the $\mathcal{F}_{\text{updRnd}}$ subroutine. Our realization proof would still work for this case. However, as explained, we expect that this is generally not needed/desired.