



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}_{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 Cryptosinous, 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.

KEYWORDS

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

1 INTRODUCTION

In the past few years, researchers made significant progress in formalizing and analyzing the security of blockchain protocols [3, 5, 16, 21, 25]. Initially analyzed in the game-based setting based on trace properties [15, 20, 37], blockchain security research has moved to simulation-based security which leverages modularity and strong security guarantees offered by universal composability (UC) frameworks [11, 12, 31]. 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}_{PL} [25]. They have successfully been applied to prove the security of various, partly newly designed blockchains (cf. [3, 5, 25]). 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 [29], 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, 22, 23, 38].

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 and formalize in this work for the first time (see our case study 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 [32]), 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}_{PL} [25]. Hence, any realization of $\mathcal{G}_{\text{ledger}}$ or \mathcal{G}_{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 Cryptsinous [25].

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. To give another example, $\mathcal{G}_{\text{ledger}}$ provides a lower-level interface to higher-level protocols than $\mathcal{F}_{\text{ledger}}$. Among others, $\mathcal{G}_{\text{ledger}}$ includes an explicit “mining” operation that has to be called manually by higher-level protocols. In contrast, $\mathcal{F}_{\text{ledger}}$ keeps such operations implicit and purely on the implementation side since higher-level protocols usually do not participate in mining (see Section 3 for details). Similarly, the design rationales for \mathcal{G}_{PL} and $\mathcal{F}_{\text{ledger}}$ are quite different as well.

We also discuss how other published ideal blockchain functionalities are captured by $\mathcal{F}_{\text{ledger}}$. Unlike $\mathcal{G}_{\text{ledger}}$ and \mathcal{G}_{PL} these functionalities, however, 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, 39, 40]. 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 currently used by more than 60 companies and institutions, including Hewlett Packard Enterprise, Intel, Microsoft, and also by NASDAQ [14, 19, 22, 23, 38]. The main application of Corda is within the financial industry, with many of the most important banks being part of the R3 consortium that develops Corda, including Bank of America, Barclays, Commerzbank, Credit Suisse, Deutsche Bank, HSBC, Royal Bank of Canada, Royal Bank of Scotland, and many more [24, 42]. Several consulting groups identify Corda as the most prominent distributed ledger technology [1, 7, 36].

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}_{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.

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 first 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 (for interested readers see Appendix A for a brief introduction).

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 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.

¹We 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.

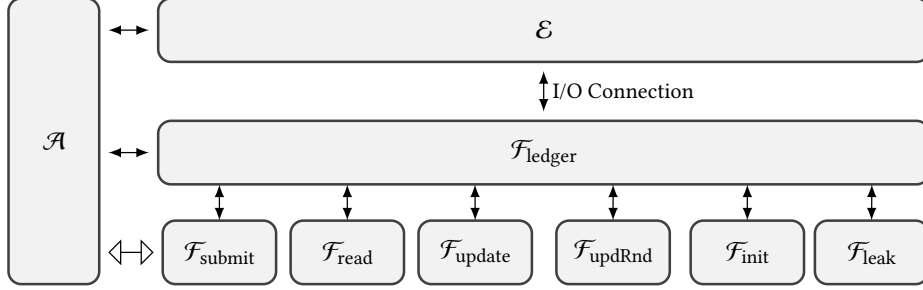


Figure 1: Overview of $\mathcal{F}_{\text{ledger}}$ and its subroutines. The open headed arrow indicates that \mathcal{A} also connects to all of $\mathcal{F}_{\text{ledger}}$'s subroutines

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 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 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*. On a technical level,

²Requests 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 (pidcur, sidcur,  $\mathcal{F}_{\text{submit}}$  : submit)
  wait for (Submit, response, leakage) s.t. response  $\in$  {true, false}
  if response = true:
    reqCtr  $\leftarrow$  reqCtr + 1; requestQueue.add(reqCtr, round, pidcur, msg)
  send (Submit, response, leakage) to NET
recv (Update, msg) from NET: {Update request triggered by the adversary.}
  send (Update, msg, internalState) to ( $\epsilon$ , sidcur,  $\mathcal{F}_{\text{update}}$  : 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  $\leftarrow \max\{i \mid (i, \_, \_, \_) \in \text{msglist}\}$ 
  check  $\leftarrow \text{listAdd} \neq \emptyset \vee \text{updRequestQueue} \neq \emptyset$ 
  for i = max + 1 to max + |listAdd| do:
    if ( $\#_1(i, \_, \_, \_) \in \text{listAdd}$ ):
      check  $\leftarrow$  false
    if  $\exists (i, \_, \text{meta}, \_, a, b) \in \text{listAdd} \wedge (a \neq \perp \vee b \neq \perp)$ :
      check  $\leftarrow$  false
  if check:
    msglist.add(listAdd)
    for all item  $\in$  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 (pidcur, sidcur,  $\mathcal{F}_{\text{read}}$  : read)
  wait for (InitRead, local, leakage) s.t. local  $\in$  {true, false}
  if local:
    send responsively (InitRead, leakage) to NET (★)
    wait for (InitRead, suggestedOutput)
    send (FinishRead, msg, suggestedOutput, internalState)
      to (pidcur, sidcur,  $\mathcal{F}_{\text{read}}$  : read)
    wait for (FinishRead, output, leakage')
    if output =  $\perp$ :
      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  $\leftarrow$  readCtr + 1; readQueue.add(pid, readCtr, round, msg)
    send (Read, readCtr, leakage) to NET
recv (DeliverRead, readCtr, suggestedOutput) from NET s.t. {Deliver network read}
  (pid, readCtr, r, msg)  $\in$  readQueue:
  send (FinishRead, msg, suggestedOutput, internalState)
    to (pidcur, sidcur,  $\mathcal{F}_{\text{read}}$  : read)
  wait for (FinishRead, output, leakage')
  if output  $\neq \perp$ :
    send responsively (FinishRead, readCtr, leakage') to NET
    wait for ack; readQueue.remove(pid, readCtr, r, msg)
    send (Read, output) to (pid, sidcur, 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. pid_{cur} is the current party and sid_{cur} the ledger's current session. round is the current time.

such an instantiation of $\mathcal{F}_{\text{update}}$ would be defined in such a way that it checks that the incoming update requests from the adversary contain a proposed extension of the global state that does not cause double-spending and does not contain transactions for honest parties that have not previously been submitted.

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 resulting 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, 37] (see below). We 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., [32]): 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. The point of setting a corruption flag is to allow 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 $\mathcal{F}_{\text{ledger}}$.

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, 25, 45]), ledgers with a graph structure (e.g., [6, 8]) or ledgers that use sharding [29, 35, 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.

³We 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.

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 δ 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 δ (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, 29, 37] 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, 27, 37] 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, 27, 37] 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 (cf. Appendix D).

Liveness [5, 16, 20, 27, 37] ensures that transactions submitted by honest clients enter the global state respectively the state read by other honest clients within ρ 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 ρ rounds old is not yet in the global state.

Privacy Properties, such as transaction privacy [30, 34, 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 Appendices D and F.

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, 27, 30, 34, 37, 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 $\mathcal{F}_{\text{ledger}}$ can 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}_{PL} [25]. Hence, any realization of $\mathcal{G}_{\text{ledger}}$ or \mathcal{G}_{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 Cryptsinous [25]. 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 Appendix D). $\mathcal{G}_{\text{ledger}}$ 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 δ), then it is guaranteed to obtain a prefix of the chain that contains all but the last at most `windowSize` blocks. This captures the security 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 some additional 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 $\mathcal{F}_{\text{ledger}}$ 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 Appendix D:

- $\mathcal{F}_{\text{init}}$ 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.

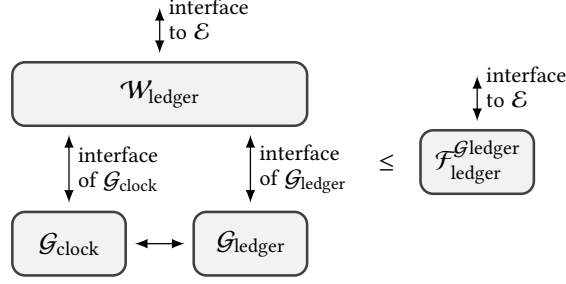


Figure 3: Realization relation of $\mathcal{G}_{\text{ledger}}$ and $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ as stated in Theorem 3.1. The system \mathcal{E} denotes the environment, modeling, as usual in UC setting, arbitrary higher level protocols. All machines are additionally connected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

- $\mathcal{F}_{\text{update}}$ 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 δ 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 security 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 Appendix D. 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., [26]). 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 $\mathcal{P}_{\text{blockchain}}$ 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 $\mathcal{P}_{\text{blockchain}}$ (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}_{PL} and other ideal blockchain functionalities. Similarly to the above result, in Appendix E we provide a proof sketch showing that $\mathcal{G}_{\text{ledger}}$'s privacy preserving variant \mathcal{G}_{PL} [25] (plus a wrapper aligning syntax and mapping abstraction levels) also realizes a suitable instantiation of $\mathcal{F}_{\text{ledger}}$. Hence, the famous privacy preserving blockchain protocol Ouroboros Cryptsinous [25], which has been proven to realize \mathcal{G}_{PL} , also realizes $\mathcal{F}_{\text{ledger}}$ with slight adjustments to the interface as described above. Besides \mathcal{G}_{PL} , Appendix E also discusses further ideal ledger functionalities [17, 18, 28] 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*,⁴ 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 by the smart contract.⁵

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 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 \hat{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 \hat{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

⁴Technically, such a pointer includes the ID of the transaction that created the state as an output as well as a counter that determines which output state of that transaction is to be used.

⁵In 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.

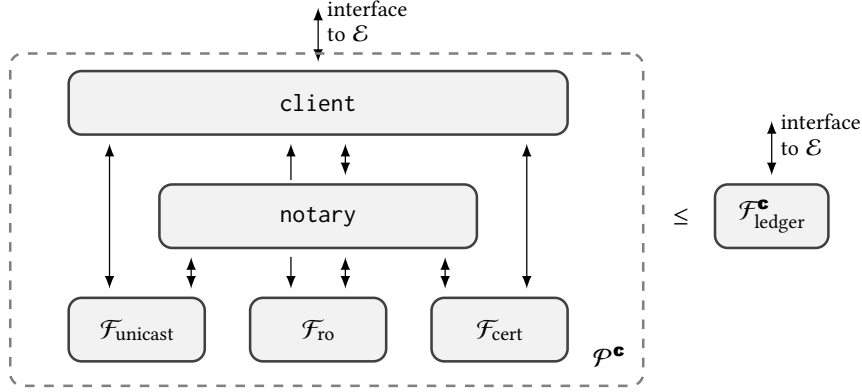


Figure 4: Corda protocol \mathcal{P}^c and realization statement.

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 (see Appendix F.1 for details).

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 specified by the documentation [40]. 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}^c of Corda in the iUC framework closely follows the above description. Formally, \mathcal{P}^c is the protocol (`client` | `notary`, $\mathcal{F}_{unicast}$, \mathcal{F}_{cert} , \mathcal{F}_{ro}) consisting of a `client` machine that is accessible to other (higher-level) protocols/the environment, an internal notary machine, and three ideal subroutines $\mathcal{F}_{unicast}$, \mathcal{F}_{cert} , and \mathcal{F}_{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 δ 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.⁶ 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^7 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.

On a technical level, 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 the formal specification of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ in Appendix F.

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

⁶Here 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 $\mathcal{F}_{\text{ledger}}$.

⁷We 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.

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, 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 2δ time units ago, where δ 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.⁸ 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. The full proof is given in Appendix F.

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} (cf. proof of Theorem F.1 in Appendix F for a full definition) internally simulates a blinded instance of $\mathcal{P}^{\mathbf{c}}$, in the following called $\mathcal{P}^{\mathbf{c}}$. During the simulation, \mathcal{S} uses dummy transactions generated from the submission leakage. The dummy

⁸ $\omega(tx)$ is a function that linearly depends on the network delay δ 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.

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 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. \square

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.

Other security properties. Appendix F.3 discusses why other standard blockchain security properties, including *chain-quality* and *chain-growth*, are not applicable to Corda.

ACKNOWLEDGMENTS

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program “Services Computing”. This work was also funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-IDs 459731562; 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).

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.swirls.com/wp-content/uploads/2016/06/2016-05-31-Swirls-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, Vol. 10032)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). 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, Vol. 4392)*, S. P. Vadhan (Ed.). 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 Ecker, 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-nasdaq-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] 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).
- [23] 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).
- [24] 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).
- [25] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. 2019. Ouroboros Crispinus: 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.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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 [33] for a full and revised version..
- [32] 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.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] 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).
- [37] 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.
- [38] 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).
- [39] R3. 2020. Corda Source Code. <https://github.com/corda/corda>. (Accessed on 04/24/2020).
- [40] R3. 2020. R3 Corda Master documentation. <https://docs.corda.net/docs/corda-os/4.4.html>. (Accessed on 04/24/2020).
- [41] R3. 2023. Reference States - R3 Documentation. <https://docs.r3.com/en/tools/cdl/ledger-evolution-view/reference-states.html>. (Accessed on 11/08/2023).
- [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 $Q = \{M_1, \dots, M_k\}$ 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 $\{M_1, \dots, M_l\}$; 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}_{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 $(\text{pubrole}_1, \dots, \text{pubrole}_n \mid \text{privrole}_1, \dots, \text{privrole}_n)$ 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 $\text{Comb}(\mathcal{Q}, \mathcal{P})$.

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.

As explained, the iUC framework offers a convenient template for specifying protocols (which can then also be combined with each other). This template includes many optional parts with sensible defaults such that protocol designers can customize exactly those parts that they need. The specifications using the iUC template that we give in this paper are mostly self explanatory, except for a few aspects:

- 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 $(pid_{\text{cur}}, sid_{\text{cur}}, role_{\text{cur}})$ 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 $(pid_{\text{call}}, sid_{\text{call}}, role_{\text{call}})$ refers to that entity.
- The special macro $\text{corr}(pid_{\text{sub}}, sid_{\text{sub}}, role_{\text{sub}})$ 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.
- 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” instead of just “send”.

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 $\mathcal{R} \in \text{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \text{Comb}(\mathcal{Q}, \mathcal{F})$ 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 readability 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** $\langle msg \rangle$ **from** $\langle sender \rangle$ **to** $\langle receiver \rangle$, **s.t.** $\langle condition \rangle$: $\langle code \rangle$ where $\langle msg \rangle$ is an input pattern, $\langle sender \rangle$ is the receiving interface (I/O or NET), $\langle receiver \rangle$ is the dedicated receiver of the message and $\langle condition \rangle$ is a condition on the input. $\langle code \rangle$ 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, $\langle msg \rangle$ 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 $\langle condition \rangle$ of that block, then uninitialized local variables are initialized with the part of the message that they matched to and $\langle code \rangle$ is executed in the context of $\langle receiver \rangle$; no other blocks are executed in this case. If m does not match p or $\langle condition \rangle$ 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** $\langle msg \rangle$ **to** $\langle sender \rangle$ where \overline{msg} is a message that is send via output interface $sender$.

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** $\langle msg \rangle$ **from** $\langle sender \rangle$, **s.t.** $\langle condition \rangle$ 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 $[n] = \{1, \dots, n\}$.
- For tuples, lists, etc. we start index counting at 0.

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

Participating roles: $\{\text{client}\}$
 Corruption model: *dynamic corruption*

Description of M_{client} :

Implemented role(s): $\{\text{client}\}$

Subroutines: $\mathcal{F}_{\text{submit}} : \text{submit}, \mathcal{F}_{\text{update}} : \text{update}, \mathcal{F}_{\text{read}} : \text{read}, \mathcal{F}_{\text{updRnd}} : \text{updRnd}, \mathcal{F}_{\text{init}} : \text{init}, \mathcal{F}_{\text{leak}} : \text{leak}$

Internal state:

- $\text{identities} \subset \{0, 1\}^* \times \mathbb{N}, \text{identities} = \emptyset$ {The set of participants and the round when they occurred first}
- $\text{round} \in \mathbb{N}_{\geq 0}, \text{round} = 0$ {Current (network) round in the protocol execution}
- $\text{msglist} \subset \mathbb{N} \times \mathbb{N} \times \{\text{tx}, \text{meta}\} \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^*,$
 $\text{msglist} = \emptyset.$ {Sequence of recorded messages considered as immutable state of the form $(id, \text{commitRound}, \text{type}, \text{msg}, \text{submitRound}, pid)$. If $\text{type} = \text{meta}$, $pid = \text{submitRound} = \perp$ }
- $\text{requestQueue} \subset \mathbb{N} \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^*,$
 $\text{requestQueue} = \emptyset$ {(Honest) messages queued for ordering, format: $(\text{tmpCtr}, \text{tx}, \text{submittingRound}, \text{submittingParty})$ }
- $\text{readQueue} \subset \{0, 1\}^* \times \mathbb{N} \times \mathbb{N} \times \{0, 1\}^*, \text{readQueue} = \emptyset$ {The queue of read responses that need to be delivered $(pid, \text{responseld}, \text{round}, \text{msg})$ }
- $\text{readCtr} \in \mathbb{N}, \text{readCtr} = 0,$ {readCtr is a temporary ID for transactions in the readQueue}
- $\text{reqCtr} \in \mathbb{N}, \text{reqCtr} = 0,$ {reqCtr are temporary IDs for transactions in the requestQueue}

Conventions:

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`, `δ` , `CorruptionSet`, `transcript`)

We often use the `CorruptionSet` as specified in [11]. We often write $pid \in \text{CorruptionSet}$ instead of $(pid, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}) \in \text{CorruptionSet}$ for brevity.

CheckID($pid, \text{sid}, \text{role}$):

Accept all messages with the same `sid`.

Corruption behavior:

- **LeakedData**($pid, \text{sid}, \text{role}$):
 if $\exists (pid, \text{registrationRound}) \in \text{identities}, \text{registrationRound} \in \mathbb{N}$:
`identities.remove(pid, registrationRound)`
send (`corrupt`, pid, sid , `internalState`) **to** ($pid_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{leak}} : \text{leak}$)
wait for (`corrupt`, `leakage`)
return (`leakage`) {Leakage during corruption (specified in subroutine $\mathcal{F}_{\text{leak}}$)}
- **AllowAdvMessage**($pid, \text{sid}, \text{role}, pid_{\text{receiver}}, \text{sid}_{\text{receiver}}, \text{role}_{\text{receiver}}, m$):
 \mathcal{A} is not allowed to call subroutines on behalf of a corrupted party.

Initialization:

- send** `InitMe` **to** ($pid_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{init}} : \text{init}$) {Initialization via $\mathcal{F}_{\text{init}}$ }
- wait for** (`Init`, `identities`, `msglist`, `corrupted`, `leakage`) **s.t.**
1. $\text{identities} \subset \{0, 1\}^* \times \{0\}, \text{round} \in \mathbb{N}, \text{msglist} \subset \mathbb{N} \times \mathbb{N} \times \{\text{meta}\} \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^*,$
 2. $\text{corrupted} \subset \{0, 1\}^* \times \{\text{sid}_{\text{cur}}\} \times \{\text{client}\},$
 3. $\text{msg} \in \text{msglist}$ are numbered consecutively, starting with 0, {We enforce correct formats and that msglist is a total ordered sequence}
 4. $\exists (_ _ _ _ a, b) \in \text{msglist}, \text{s.t. } a \neq \perp \vee b \neq \perp$
- `identities` \leftarrow `identities`; `msglist` \leftarrow `msglist`, `CorruptionSet` \leftarrow `corrupted`
- send responsively** (`Init`, `leakage`) **to** NET {Send leaked information to \mathcal{A} }
- wait for** `ack` **from** NET

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

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 to 7. For technical details of and notation specific to the iUC framework, please see our brief summary in Section A.

Note that, in addition to what is described in Section 2, $\mathcal{F}_{\text{ledger}}$ as defined in Figure 5 to 7 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 the 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

Description of M_{Client} (cont.):

MessagePreprocessing:	
rcv ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}, \text{msg}$) from I/O:	$\left\{ \begin{array}{l} \text{Register unknown party before its first sub-} \\ \text{mit/read operation} \end{array} \right.$
if ($\text{pid}_{\text{cur}}, _ \notin \text{identities} \wedge \text{msg}$ starts with Submit or Read):	
$\text{identities.add}(\text{pid}_{\text{cur}}, \text{round})$	
Main:	
rcv (Submit , msg) from I/O:	$\left\{ \begin{array}{l} \text{Submission request from a honest identity} \\ \text{Forward request to } \mathcal{F}_{\text{submit}} \end{array} \right.$
send (Submit , msg , internalState) to ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{submit}} : \text{submit}$)	$\left\{ \begin{array}{l} \text{requestQueue.add}(_) \text{ equals } \text{requestQueue} \leftarrow \\ \text{requestQueue} \cup \{_ \}. \text{ Records message, round, identity,} \\ \text{and its state for "consensus"} \end{array} \right.$
wait for (Submit , response , leakage) s.t. $\text{response} \in \{\text{true}, \text{false}\}$	
if $\text{response} = \text{true}$:	
$\text{reqCtr} \leftarrow \text{reqCtr} + 1$	
$\text{requestQueue.add}(\text{reqCtr}, \text{round}, \text{pid}_{\text{cur}}, \text{msg})$	$\left\{ \begin{array}{l} \text{If } \mathcal{F}_{\text{submit}} \text{ leaks data regarding the submitted transaction, this is} \\ \text{forwarded to } \mathcal{A} \end{array} \right.$
send (Submit , response , leakage) to NET	
rcv (Read , msg) from I/O:	$\left\{ \begin{array}{l} \text{Read request from an honest identity} \\ \text{Forward the request to } \mathcal{F}_{\text{read}} \end{array} \right.$
send (InitRead , msg , internalState) to ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \text{read}$)	$\left\{ \begin{array}{l} \text{local} = \text{true} \text{ models a "local" read, clients get} \\ \text{immediate response, otherwise it is a network} \\ \text{read} \end{array} \right.$
wait for (InitRead , local , leakage) s.t. $\text{local} \in \{\text{true}, \text{false}\}$	
if local :	
send responsively (InitRead , leakage) to NET (\star)	$\left\{ \begin{array}{l} \mathcal{F}_{\text{read}} \text{ leaks data, this is forwarded to } \mathcal{A} \\ \mathcal{A} \text{ may influence the read processing} \end{array} \right.$
wait for (InitRead , suggestedOutput)	
send (FinishRead , msg , suggestedOutput , internalState) to ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \text{read}$)	
wait for (FinishRead , output , $\text{leakage}'$)	
if $\text{output} = \perp$:	$\left\{ \begin{array}{l} \text{If } \mathcal{A}'\text{s input for } \mathcal{F}_{\text{read}} \text{ is not accepted, it is triggered again} \\ \text{Go back to } (\star) \text{ and repeat the request (local variables } \text{suggestedOutput}, \text{output, and } \text{leakage}' \text{ are cleared)} \end{array} \right.$
send responsively (FinishRead , $\text{leakage}'$) to NET	$\left\{ \begin{array}{l} \mathcal{F}_{\text{read}} \text{ leaks data, this is forwarded to } \mathcal{A} \end{array} \right.$
wait for ack	
reply (Read , output)	
else:	
$\text{readCtr} \leftarrow \text{readCtr} + 1; \text{readQueue.add}(\text{pid}, \text{readCtr}, \text{round}, \text{msg})$	$\left\{ \begin{array}{l} \text{In case of network read,} \\ \text{store request} \end{array} \right.$
send (Read , readCtr , leakage) to NET	$\left\{ \begin{array}{l} \text{If } \mathcal{F}_{\text{read}} \text{ leaks data, this is forwarded to } \mathcal{A} \end{array} \right.$
rcv (DeliverRead , readCtr , suggestedOutput) from NET s.t. $(\text{pid}, \text{readCtr}, r, \text{msg}) \in \text{readQueue}$:	$\left\{ \begin{array}{l} \mathcal{A} \text{ triggers message delivery per message (this may include reordering of messages, non-delivery of messages, and} \\ \text{manipulation of delivered data, if not enforced by } \mathcal{F}_{\text{updRnd}} \end{array} \right.$
send (FinishRead , msg , suggestedOutput , internalState) to ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \text{read}$)	
wait for (FinishRead , output , $\text{leakage}'$)	
if $\text{output} \neq \perp$:	
send responsively (FinishRead , readCtr , $\text{leakage}'$) to NET	
wait for ack	
$\text{readQueue.remove}(\text{pid}, \text{readCtr}, r, \text{msg})$	$\left\{ \begin{array}{l} \text{Clean up readQueue} \end{array} \right.$
send (Read , output) to ($\text{pid}, \text{sid}_{\text{cur}}, \text{I/O}$)	
else:	
send nack to NET	$\left\{ \begin{array}{l} \text{Delivery request of } \mathcal{A} \text{ was denied} \end{array} \right.$

Figure 6: The ideal ledger functionality $\mathcal{F}_{\text{ledger}}$ (Pt. 2).

Main:	
recv (CorruptedRead, pid, msg) from NET s.t. $pid \in \text{CorruptionSet}$:	$\{\text{Read interface for } \mathcal{A}\}$
send (CorruptedRead, $pid, msg, \text{internalState}$) to ($pid, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \text{read}$)	$\{\text{Forward request to } \mathcal{F}_{\text{read}}\}$
wait for (FinishRead, $leakage$)	
send (Read, $pid, leakage$) to NET	$\{\text{Forwarded data to } \mathcal{A}\}$
recv (Update, msg) from NET:	$\{\text{Update or maintain request triggered by the adversary}\}$
send (Update, $msg, \text{internalState}$) to ($\epsilon, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{update}} : \text{update}$)	$\{\mathcal{F}_{\text{update}}$ outputs which data to ap-
wait for (Update, $msglist, \text{updRequestQueue}, leakage$)	pend to $msglist$ and an updated
s.t. $msglist \subset \mathbb{N} \times \{\text{round}\} \times \{\text{tx}, \text{meta}\} \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^*$	requestQueue
$max \leftarrow \max\{i \mid (i, _, _, _) \in msglist\}$	$\{\text{Check that } msglist \text{ is a totally ordered sequence, extending}$
$check \leftarrow msglist \neq \emptyset \vee \text{updRequestQueue} \neq \emptyset$	$\{\text{the existing } msglist. \text{ If } msglist = \emptyset \text{ then } max \text{ defaults to } -1\}$
for $i = max + 1$ to $max + msglist $ do :	
if $\nexists_1(i, _, _, _) \in msglist$:	$\{\text{Check that there exists exactly one entry for every ID } i \text{ in a continous}$
$check \leftarrow \text{false}$	$\{\text{sequence (no gaps)}\}$
if $\exists(i, _, \text{meta}, _, a, b) \in msglist \wedge (a \neq \perp \vee b \neq \perp)$:	$\{\text{Check that meta data has correct format}\}$
$check \leftarrow \text{false}$	
if $check$:	$\{\text{If the update is totally ordered and no new messages were added to}$
$msglist.add(msglist)$	$\{\text{requestQueue, we accept the update}\}$
for all $item \in \text{updRequestQueue}$ do :	$\{\text{Remove "consumed" elements from requestQueue}\}$
$requestQueue.remove(item)$	
reply (Update, $check, leakage$)	$\{\text{Inform } \mathcal{A} \text{ if update was successful and leak data}\}$
recv UpdateRound from NET:	$\{\mathcal{A} \text{ triggers round update if current round satisfies rules of } \mathcal{F}_{\text{updRnd}}\}$
send (UpdateRound, internalState) to ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{updRnd}} : \text{updRnd}$)	
wait for (UpdateRound, $response, leakage$)	
if $response = \text{true}$:	
$round \leftarrow round + 1$	
reply (UpdateRound, $response, leakage$)	
recv GetCurRound from I/O or NET:	$\{\mathcal{A} \text{ and } \mathcal{E} \text{ are allowed to query the current round}\}$
reply (GetCurRound, $round$)	
recv DeRegister from I/O:	$\{\text{De-register honest party}\}$
Remove the unique tuple ($\text{pid}_{\text{cur}}, r$) from identities	
send responsively DeRegister to NET	$\{\text{Inform } \mathcal{A} \text{ on the deregistration}\}$
wait for ack	
reply DeRegister	

Figure 7: The ideal ledger functionality $\mathcal{F}_{\text{ledger}}$ (Pt. 3)

in the realization. If one needs to 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 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. 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 $\mathcal{F}_{\text{ledger}}$ (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 $\mathcal{F}_{\text{ledger}}$ 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 from creating smart contracts with exponential (or worse) runtime. The same bound can be used for $\mathcal{F}_{\text{ledger}}$.

D FULL DETAILS: $\mathcal{G}_{\text{ledger}}$ REALIZES $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$

In this section, we provide full details for Theorem 3.1, including formal specifications of all machines and a formal proof of the theorem. We start by explaining the ideal blockchain functionality $\mathcal{G}_{\text{ledger}}$ on an intuitive level (we recall the formal specification of this functionality formulated in the iUC framework in Figure 8 to 10).

Our ideal functionality $\mathcal{F}_{\text{ledger}}$ is in the spirit of and adopts some of the underlying ideas from the existing ideal blockchain functionality $\mathcal{G}_{\text{ledger}}$. As a result, both functionalities share similarities at a high level. More specifically, $\mathcal{G}_{\text{ledger}}$ 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, $\mathcal{G}_{\text{ledger}}$ provides only the security property of consistency. 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. More specifically, if the honest party has been registered for a sufficiently long amount of time (larger than parameter δ), 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, $\mathcal{G}_{\text{ledger}}$ also supports dynamic (de-)registration of parties and offers a clock, modeled via a subroutine $\mathcal{G}_{\text{clock}}$ (see Figure 11 for the formal specification formulated in the iUC framework), that is advanced by $\mathcal{G}_{\text{ledger}}$ 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 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 (which we formally define in Figures 14 to 20 at the end of this section) 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 δ 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 security property of *chain-growth* and other time-related security 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 $\mathcal{G}_{\text{ledger}}$.

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

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 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 hold. 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 and Appendix F).

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.

¹⁰We consider the standard definition of liveness in distributed ledger, resp. blockchain, context [20, 37]: A transaction casted by an honest client should become part of the ledger after some (known) upper time boundary.

- 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 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 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 $\text{windowSize} \in \mathbb{N}$ 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 $\mathcal{F}_{\text{ledger}}$ (we provide a formal definition of $\mathcal{W}_{\text{ledger}}$ in Figure 12 and Figure 13 at the end of this section. See also Figure 3 for an illustration of the static structure of the combined protocol). 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 [26], 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 [26] (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 our instantiation of $\mathcal{F}_{\text{ledger}}$).

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.

Having explained both the ideal protocol $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ and its intended realization, we can now formally state the main theorem of this section (cf. Figure 3 for an illustration of this theorem):

THEOREM D.1. *Let $\mathcal{G}_{\text{ledger}}$ be the ideal blockchain functionality with arbitrary parameters such that all parameterized algorithms are deterministic. Let $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ be the instantiation of $\mathcal{F}_{\text{ledger}}$ as described above, where the internal subroutines use the same parameters as $\mathcal{G}_{\text{ledger}}$. Furthermore, if $\mathcal{G}_{\text{ledger}}$ is parameterized such that it provides liveness and/or chain-growth, then let the parameters of the subroutine $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}_{\text{ledger}}}$ in $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ be set such that it also enforces the same properties. Then:*

$$(\mathcal{W}_{\text{ledger}} \mid \mathcal{G}_{\text{ledger}}, \mathcal{G}_{\text{clock}}) \leq \mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$$

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

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

Network communication from/to the environment

- Messages that \mathcal{S} receives on the network connected to the environment (and which are hence meant for \mathcal{R}) are forwarded to internal simulation \mathcal{R}' .
- Any messages sent by \mathcal{R}' on its network interface (that are hence meant for the environment) are forwarded to the environment \mathcal{E} .
- If the global blockchain in \mathcal{R}' is updated (as a result of a request that is not already handled separately below), then the simulator performs the same update in $\mathcal{F}_{\text{ledger}}$ before continuing the simulation.

Corruption handling

- The simulator \mathcal{S} keeps the corruption status of entities in \mathcal{R}' and \mathcal{I} synchronized. That is, whenever an entity of $\mathcal{W}_{\text{ledger}}$ in \mathcal{R}' starts to consider itself corrupted, the simulator first corrupts the corresponding entity of $\mathcal{F}_{\text{ledger}}$ in \mathcal{I} before continuing its simulation.
- Incoming Messages from corrupted entities of $\mathcal{F}_{\text{ledger}}$ in \mathcal{I} are forwarded on the network to the environment in the name of the corresponding entity of $\mathcal{W}_{\text{ledger}}$ in \mathcal{R}' . Conversely, whenever a corrupted entity of $\mathcal{W}_{\text{ledger}}$ wants to output a message to a higher-level protocol, \mathcal{S} instructs the corresponding entity of $\mathcal{F}_{\text{ledger}}$ to output the same message to the higher-level protocol.

Transaction submission

- Whenever an honest entity $entity = (pid, sid, role)$ receives a request (`Submit, msg`) to submit a new transaction msg , $\mathcal{F}_{\text{ledger}}$ respectively the subroutine $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ first pre-processes an update to the global transaction list. During this operation, the simulator is asked to provide some missing information (such as a list of honest activations) via a message `Preprocess` that also includes the transaction msg and $\mathcal{F}_{\text{ledger}}$'s current state. Upon receiving this message, \mathcal{S} extracts all required information from the internal simulation \mathcal{R}' and returns this information, except for also extending the set of honest activations extracted from \mathcal{R}' by an additional submit request submitted by the currently active honest party of $\mathcal{F}_{\text{ledger}}$.
- After the submit request has been processed, $\mathcal{F}_{\text{ledger}}$ sends the validation result of the transaction as well as a leakage to \mathcal{S} . Upon receiving this information, \mathcal{S} first sends the message `finalizePending` to $\mathcal{F}_{\text{ledger}}$ in order to apply the pre-processed and cached update to the global transaction list of $\mathcal{F}_{\text{ledger}}$. Afterwards, \mathcal{S} simulates a submit request (`Submit, msg`) to the honest entity $entity$ of $\mathcal{W}_{\text{ledger}}$ in \mathcal{R}' , including the resulting output to the environment.

Read requests

Whenever an honest entity $entity$ receives a request (`Read, msg`) to read from the global state, $\mathcal{F}_{\text{ledger}}$ forwards this (non-local) read request to \mathcal{S} and waits to receive a suggested output. Upon receiving this request, \mathcal{S} first triggers a global state update in $\mathcal{F}_{\text{ledger}}$ using the information from the internally simulated \mathcal{R}' (except for the set of honest activations, which is extended by

¹¹We 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.

one additional read request from party pid). After this update, \mathcal{S} simulates a read (Read, msg) for entity of $\mathcal{W}_{\text{ledger}}$ in \mathcal{R}' . The resulting output out of the simulated read request from $\mathcal{W}_{\text{ledger}}$ is used to compute a response for $\mathcal{F}_{\text{ledger}}$:

- If the entity entity is de-synchronized, i. e., entity is registered for less than δ time units at $\mathcal{F}_{\text{ledger}}$, then out is forwarded to $\mathcal{F}_{\text{ledger}}$ as a response to the read request of entity .
- If the entity entity is synchronized, i. e., entity is registered for at least δ time units at $\mathcal{F}_{\text{ledger}}$ then \mathcal{S} computes a pointer ptr to the last transaction in out . The pointer ptr is returned to $\mathcal{F}_{\text{ledger}}$ as a response to the read request of entity .

Further details

- \mathcal{S} keeps the clocks/rounds of \mathcal{R}' and $\mathcal{F}_{\text{ledger}}$ synchronous. That is, \mathcal{S} sends UpdateRound to $\mathcal{F}_{\text{ledger}}$ whenever a round update in the simulated $\mathcal{G}_{\text{clock}}$ is performed and before continuing the simulation.
- Whenever \mathcal{S} is notified about the de-registration of an entity, \mathcal{S} simulates the de-registration of the corresponding entity in \mathcal{R} . Afterwards \mathcal{S} returns control to $\mathcal{F}_{\text{ledger}}$.

This concludes the description of the simulator. It is easy to see that (i) $\{\mathcal{S}, \mathcal{I}\}$ is environmentally bounded¹² and (ii) \mathcal{S} is a responsive simulator for \mathcal{I} , i. e., restricting messages from \mathcal{I} are answered immediately as long as $\{\mathcal{S}, \mathcal{I}\}$ runs with a responsive environment. We now argue that \mathcal{R} and $\{\mathcal{S}, \mathcal{I}\}$ are indeed indistinguishable for any (responsive) environment $\mathcal{E} \in \text{Env}(\mathcal{R})$.

Now, let $\mathcal{E} \in \text{Env}(\mathcal{R})$ be an arbitrary but fixed environment. First, observe that $\mathcal{F}_{\text{ledger}}$ provides \mathcal{S} with full information about all requests performed by higher-level protocols, such as the actual transactions submitted to the ledger, and including entity (de-)registration in particular. Hence, the simulated protocol \mathcal{R}' within \mathcal{S} obtains the same inputs and thus performs identical to the real world \mathcal{R} . As a result, the network behavior simulated by \mathcal{S} towards the environment is indistinguishable from the network behavior of \mathcal{R} . Observe that state changes triggered via network interface are synchronized between \mathcal{R}' and \mathcal{I} . Together with the state synchronization during I/O interaction (see below), the simulator can keep the states of \mathcal{R}' and \mathcal{I} in synchronization. Furthermore, it also follows that the corruption status of entities in the real and ideal world is always identical. Since the simulator has full control over corrupted entities, which are handled via the internal simulation \mathcal{R}' , this implies that the I/O behavior of corrupted entities of \mathcal{R}/\mathcal{I} towards higher level protocols/the environment is also identical in the real and ideal world. The only way to potentially distinguish the real and ideal world is the I/O behavior of honest entities of \mathcal{R}/\mathcal{I} towards higher-level protocols.

We will now go over all possible interactions with honest entities on the I/O interface and argue, by induction, that all of those interactions result in identical behavior towards the environment, i. e., are also indistinguishable. At the start of a run, there were no interactions on the I/O interface with honest parties yet. In the following, assume that all I/O interactions to far have resulted in the same behavior visible towards the environment in both the real and ideal world.

Submission requests: Submission requests do not directly result into an input to the environment, however, they might affect the output of future read requests by changing the internal buffer and global transaction lists of $\mathcal{F}_{\text{ledger}}$, respectively the internal buffer and global blockchain of \mathcal{R} (and by this also \mathcal{R}' ; we keep this implicit in what follows). For read requests to behave identical, we now have to argue that these changes are “synchronized”, i. e., (i) the buffered set of transactions in $\mathcal{F}_{\text{ledger}}$ is a subset of the buffered set of transactions in \mathcal{R} , and those transactions that are in the buffer of \mathcal{R} but not $\mathcal{F}_{\text{ledger}}$ are by dishonest parties, and (ii) the blockchain of \mathcal{R} , when transformed via the toMsglist function as executed by the wrapper, matches the global list of transactions in $\mathcal{F}_{\text{ledger}}$.

Observe that, upon receiving a submission request, $\mathcal{F}_{\text{ledger}}$ behaves just as $\mathcal{G}_{\text{ledger}}$: it first updates its global transaction list. This update is performed based on the internal buffer and global transaction list variables, which (by induction assumption) are synchronized with those of $\mathcal{G}_{\text{ledger}}$. Any missing information, such as transactions of dishonest parties that are not part of the internal buffer of $\mathcal{F}_{\text{ledger}}$, are provided by the simulator from the internal simulation \mathcal{R}' , i. e., they are the same as for \mathcal{R} . Those inputs are then used by $\mathcal{F}_{\text{ledger}}$ to run the same deterministic algorithms as $\mathcal{G}_{\text{ledger}}$ in \mathcal{R} , resulting in the same block extensions. Hence, the cached global state stored in $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ is synchronized with \mathcal{R} . Afterwards, $\mathcal{F}_{\text{ledger}}$ uses the same deterministic algorithm as $\mathcal{G}_{\text{ledger}}$ to validate the incoming transaction using the cached internal state from $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$. Again, inputs to this function are synchronized or provided by the simulator from \mathcal{R}' , the validation result is identical in both worlds the in particular the buffer sets remain synchronized. Finally, observe that the simulator immediately instructs $\mathcal{F}_{\text{ledger}}$ to apply the cached global state from $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ to the global state in $\mathcal{F}_{\text{ledger}}$. By this, at the end of a submit request both the global state and the buffer of $\mathcal{F}_{\text{ledger}}$ remain synchronized with \mathcal{R} .

Read requests: Observe that, upon receiving a read request, $\mathcal{F}_{\text{ledger}}$ gives control to the simulator who then triggers a state update for the global transaction list in $\mathcal{F}_{\text{ledger}}$. This matches the behavior of $\mathcal{G}_{\text{ledger}}$, i. e., the global transaction and global block lists that are used for the following read request (and any later requests) remain synchronized.

After the state update, there are two cases: if the honest entity receiving the read request has not been registered for less than δ time units, then the simulator is allowed to determine the exact output. Since this is done by forwarding the output of \mathcal{R}' , the output is identical in both worlds. For parties that are already registered for at least δ time units, then the simulator may only

¹²This is the polynomial runtime notion employed by the iUC framework.

provide a pointer that determines the prefix of the global transaction list that is output as a result of the read request. That pointer must be at least as large as the previous pointer (if such a pointer exists) and must be within the last `windowSize` blocks of the global transaction list (when transformed to $\mathcal{G}_{\text{ledger}}$'s state format via `toState`). Since this pointer is determined by the block that is returned in \mathcal{R}' , which by definition of $\mathcal{G}_{\text{ledger}}$ is such that it meets all the requirements enforced by \mathcal{I} , we have that $\mathcal{F}_{\text{ledger}}$ accepts the pointer of \mathcal{S} and indeed outputs the same transaction list as in the real world.

Current time requests: As the simulator updates the internal clock of $\mathcal{F}_{\text{ledger}}$ every time an update to $\mathcal{G}_{\text{clock}}$ in \mathcal{R}' occurs, both worlds always output the same value for the current time. Note that, even if $\mathcal{F}_{\text{ledger}}^{\text{updRnd}}$ enforces *liveness* and/or *chain-growth*, any round update requests of \mathcal{S} will indeed be accepted: by assumption, \mathcal{R}' guarantees that liveness and/or chain-growth are still preserved whenever the clock in $\mathcal{G}_{\text{clock}}$ advances. As the global state is synchronized, the same thus also holds true for $\mathcal{F}_{\text{ledger}}$ whenever \mathcal{S} updates the clock.

(De-)Registration: In the ideal world, $\mathcal{F}_{\text{ledger}}$ registers honest entities whenever they first perform a submit or read request. This is identical to the behavior of the wrapper $\mathcal{W}_{\text{ledger}}$ in the real world. Similarly for de-registration of honest entities. Hence, the sets of honest registered entities are also synchronized in the real and ideal world.

Updates to the global state caused by requests on the network: Observe that these updates are also applied to $\mathcal{F}_{\text{ledger}}$ by the simulator. Note in particular that these updates will be accepted by $\mathcal{F}_{\text{ledger}}$ since $\mathcal{G}_{\text{ledger}}$ has already accepted them based on a synchronized state and using deterministic algorithms. Hence, the global state stays synchronized even when the adversary, e.g., instructs the wrapper to send a `MaintainLedger` command in the name of an honest party and, by this, causes an update to the global state.

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

Remark: Though, we have not explicitly discussed the common property of chain-quality above, we emphasize that $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ also provides, resp. ensures chain-quality if $\mathcal{G}_{\text{ledger}}$ ensures the property. If one wants to capture chain-quality explicitly, one needs to adapt $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ (cf. Figure 16 and 17) such that it captures chain-quality. The handling of liveness and chain-growth in $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ (cf. Figure 18) can be used as blueprint for incorporating chain-quality explicitly into $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$.

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

Participating roles: {client}	
Corruption model: <i>dynamic</i>	
Protocol parameters:	
- validate	{Decides on the validity of a transaction with respect to the current state. Used to clean the buffer of transactions.
- extendPolicy	{The function that specifies the ledger's guarantees in extending the ledger state (e.g., speed, content etc.).
- Blockify	{The function to format the ledger state output.
- predictTime	{The function to predict the real-world time advancement.
- delay $\in \mathbb{N}$	{A delay parameter for the time it takes for a newly joining miner to become synchronized.
- windowSize $\in \mathbb{N}$	{The window size (number of blocks) of the sliding window.

Description of M_{client} :

Implemented role(s): {client}	
Subroutines: $\mathcal{G}_{\text{clock}} : \text{clock}$	
Internal state:	
- $P \subset \{0, 1\}^*$, identities = \emptyset	{The (dynamic) set of registered participants.
- state $\subset \{0, 1\}^*$, state = ϵ	{The ledger state, i. e., a sequence of blocks containing the content.
- nextBC $\in \{0, 1\}^*$, nextBC = ϵ	{Stores the current adversarial suggestion for extending the ledger state.
- buffer $\in \mathbb{N} \times \{0, 1\}^* \times \mathbb{N} \times \{0, 1\}^*$, buffer = ϵ .	{The buffer of submitted input values of the form (txId, txContent, submissionTime, submittingParty).
- $\tau_L \in \mathbb{N}$, $\tau_L = 0$.	{The current time
- $\tau_{\text{state}} \in \mathbb{N} \times \mathbb{N}$, $\tau_{\text{state}} = \emptyset$.	{A vector containing (for each state block) the time when the block was added to the ledger state.
- $I_H^T \in \{0, 1\}^* \times \{0, 1\}^* \times \mathbb{N}$, $I_H^T = \emptyset$.	{The timed honest-input sequence with data of the form (msg, submittingParty, submissionTime).
- PM $\subset P \times \{d, s, c\} \times \mathbb{N} \times \mathbb{N} \times \{0, 1\}^*$, PM = \emptyset	{Storage to manage parties. An entry contains the party id, wheter the party is d - desynchronized, s - synchronized, or c - corrupted. The registration time, the pointer to the current state (or the block number), and its current state.
CheckID (pid, sid, role): Accept all messages with the same sid.	
Corruption behavior:	
- LeakedData:	
if $\exists (pid, ps, \tau, pt, state) \in \text{PM}$:	{Record corruption in PM
PM.remove(pid, ps, τ , pt, state); PM.add(pid, c, τ , pt, state)	
else:	
PM.add(pid, c, τ , pt, state)	
return \perp	{We do not leak additional data during corruption
- AllowAdvMessage: \mathcal{A} is not allowed to call subroutines on behalf of a corrupted party.	

Figure 8: The ideal ledger functionality $\mathcal{G}_{\text{ledger}}$ translated into the iUC model (Pt. 1)

MessagePreprocessing:	
recv msg from I/O or NET:	
send Read to $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{clock}} : \text{clock})$	{Update the clock
wait for t from $\mathcal{G}_{\text{clock}} : \text{clock}; \tau_L \leftarrow t$	{Update the current time in $\mathcal{G}_{\text{ledger}}$
send GetRegistered to $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{clock}} : \text{clock})$	{Request which parties are properly registered at $\mathcal{G}_{\text{clock}}$
wait for P	
for $p \in P$ do:	{Honest parties connected since delay rounds become "synchronized"
if $p \in P \wedge (p, d, \tau, pt, state) \in PM \wedge \tau_L - \tau > \text{delay}:$	
$PM.\text{remove}(p, d, \tau, pt, state)$	
$PM.\text{add}((p, s, \tau, pt, state))$	
for $p \in P$ do:	{Remove the "synchronized" flag from parties that are not registered to the clock.
if $(p, s, \tau, pt, state) \in PM \wedge p \notin P:$	
$PM.\text{remove}(p, s, \tau, pt, state)$	
$PM.\text{add}((p, d, \tau, pt, state))$	
if $(pid, s, _ _ _) \in PM \vee (pid, d, _ _ _) \in PM:$	{Additional handling for messages received by honest parties
$I_H^T.\text{add}(msg, pid, \tau_L)$	{Store transaction as honest.
$N \leftarrow (N_1, \dots, N_l) \leftarrow \text{extendPolicy}(I_H^T, \text{state}, \text{nxtBC}, \text{buffer}, \tau_{\text{state}})$	{Check if the state needs to be extended
if $N \neq \epsilon:$	{Verify whether current chain extension behavior matches extendPolicy
$\text{state.add}(\text{Blockify}(N_1), \dots, \text{Blockify}(N_l))$	{Add blocks that are necessary to fulfill the extendPolicy to the state.
for $i = 1$ to l do:	
$\tau_{\text{state}}.\text{add}(\tau_L)$	{Add timestamps for each block to τ_L
for $tx \in \text{buffer}$ do:	{Clean up transactions in buffer that are not valid according to state.
if $\text{validate}(tx, \text{state}, \text{buffer}) = \text{false}:$	
$\text{buffer.remove}(tx); \text{nxtBC} \leftarrow \epsilon$	
if $\exists (p, s, \tau, pt, state) \in PM, \text{s.t. } \text{state} - pt > \text{windowSize} \vee pt < \text{state} :$	{Update state of honest and synced participants if necessary
for $(\hat{p}, \hat{s}, \hat{\tau}, \hat{pt}, \hat{state}) \in PM$ do:	
{If the pointer of one party is too far away from the current head of the chain or pointer and state are not synced, update to head (as fallback)	
$PM.\text{remove}((\hat{p}, \hat{s}, \hat{\tau}, \hat{pt}, \hat{state}, \hat{pt}); PM.\text{add}((\hat{p}, \hat{s}, \hat{\tau}, \text{state} , \text{state.current}))$	
$\text{nxtBC} \leftarrow \epsilon$	{Clear proposed block(s)
Main:	
recv Register from I/O s.t. $pid \notin P:$	{Registration process for honest parties
$P.\text{add}(pid); PM.\text{add}(pid, d, \tau_L, \epsilon, \epsilon)$	
reply Register	
recv (Register, pid, sid) from NET s.t. $pid \notin P:$	{Registration process for honest parties
$P.\text{add}(pid); PM.\text{add}(pid, c, \tau_L, \epsilon, \epsilon)$	
reply Register	
recv DeRegister from I/O s.t. $(pid, s \vee d, \tau, pt, state) \in PM:$	{Deregistration of honest parties
$P.\text{remove}(pid); PM.\text{remove}(pid, ps, \tau, pt, state)$	
reply DeRegister	
recv (DeRegister, pid, sid) from NET s.t. $(pid, c, \tau, pt, state) \in PM:$	{Deregistration of corrupted parties
$P.\text{remove}(pid); PM.\text{remove}(pid, ps, \tau, pt, state)$	
reply DeRegister	
recv (GetRegistered, sid) from NET:	{ \mathcal{A} requests the current set of participants
reply (GetRegistered, sid, P)	
recv GetRegistered from I/O:	{ \mathcal{E} requests the current set of participants
reply (GetRegistered, P)	

Figure 9: The ideal ledger functionality $\mathcal{G}_{\text{ledger}}$ translated into the iUC model (Pt. 2)

Main:	
rcv (Submit, tx) from I/O s.t. $(pid_{\text{cur}}, s \vee d, _, _) \in \text{PM}$: Choose a unique $txId$; $\hat{tx} \leftarrow (txId, tx, \tau_L, pid)$ if validate(\hat{tx} , state, buffer) = true: buffer.add(\hat{tx}) send (Submit, \hat{tx}) to NET	{Honest parties submit a transaction {Record \hat{tx} as queued for consensus
rcv (Submit, pid, tx) from NET s.t. $(pid, c, _, _) \in \text{PM}$: Choose a unique $txId$; $\hat{tx} \leftarrow (txId, tx, \tau_L, pid)$ if validate(\hat{tx} , state, buffer) = true: buffer.add(\hat{tx}) send (Submit, \hat{tx}) to NET	{Corrupted parties submit a transaction {Record \hat{tx} as queued for consensus
rcv Read from I/O s.t. $(pid, ps, \tau, pt, state) \in \text{PM}, ps \in \{s, d\}$: $state' \leftarrow state _{\min\{pt, state \}}$ PM.remove($pid, ps, \tau, pt, state$); PM.add($pid, ps, \tau, pt, state'$) reply (Read, $state'$)	{Handling of read requests from honest parties
rcv (Read, pid) from NET s.t. $(pid, d, \tau, pt, state) \in \text{PM}$: reply (Read, state, buffer, I_H^T)	{ \mathcal{A} gets “full” knowledge of the status of the chain
rcv MaintainLedger from I/O s.t. $(pid, s \vee d, _, pt, state) \in \text{PM}$: if predictTime(I_H^T) > τ_L : send (Update, sid) to $(pid, sid, \mathcal{G}_{\text{clock}} : \text{clock})$ else : send (MaintainLedger) to NET	{MaintainLedger command for honest parties
rcv (NextBlock, $hFlag, txid_1, \dots, txid_l$) from NET: $nxtBC \leftarrow \epsilon$ for $i = 1$ to l do : if $\exists (txid_i, tx_i, \tau_i, pid_i) \in \text{buffer}$: $nxtBC.add(txid_i, tx_i, \tau_i, pid_i)$ $nxtBC.add((hFlag, nxtBC))$ reply (NextBlock, ack)	{Handling of suggested block candidates by \mathcal{A}
rcv (SetSlack, $(p_1, pt'_1), \dots, (p_l, pt'_l)$) from NET s.t. $\forall pid \in \{p_1, \dots, p_l\} : (pid, s, _, _) \in \text{PM}$: $u \leftarrow \{(p_1, pt'_1), \dots, (p_l, pt'_l)\}$ if $\exists (p_j, pt'_j) \in u$, s.t. $ state - pt'_j > \text{windowSize} \vee pt'_j < state_j $: for all $(pid', (s), \tau', pt', state') \in \text{PM}$ do : PM.remove($pid', (s), \tau', pt', state'$) PM.add($pid', (s), \tau', state , state$) else : for $i = 1$ to l do : PM.remove($p_i, ps_i, \tau_i, pt_i, state_i$) PM.remove($p_i, ps_i, \tau_i, pt'_i, state_i$) reply (SetSlack, ack)	{ \mathcal{A} may set the exact “state” of a synchronized party depending on windowSize {In the case that windowSize is violated or a pointer is moved backwards {Update all synchronized parties to “longest chain”
rcv (DesyncState, $(p_1, state'_1), \dots, (p_l, state'_l)$) from NET: s.t. $pid \in P \wedge ps_i = d, \forall i = 1, \dots, l \wedge (p_i, ps_i, \tau_i, pt_i, state_i) \in \text{PM}$ for $i = 1$ to l do : PM.remove($p_i, ps_i, \tau_i, pt_i, state_i$) PM.remove($p_i, ps_i, \tau_i, \epsilon, state'_i$) reply (DesyncState, ack)	{ \mathcal{A} may set the “state” of de-synchronized parties

 Figure 10: The ideal ledger functionality $\mathcal{G}_{\text{ledger}}$ translated into the iUC model (Pt. 3)

Description of the protocol $\mathcal{G}_{\text{clock}} = (\text{clock})$:

Participating roles: {clock}
Corruption model: *incorruptible*

Description of M_{clock} :

Implemented role(s): {clock}
Subroutines: $\mathcal{G}_{\text{ledger}}$: client
Internal state:

- $P \subset \{0, 1\}^* \times \{0, 1\}, P = \emptyset$ {The set of registered participants of the form (pid, activated).}
- $\tau \in \mathbb{N}$, initially $\tau = 0$ {Current time in the $\mathcal{G}_{\text{clock}}$ }
- $F \in \{0, 1\}$, initially $F = 0$ {Round update status of $\mathcal{G}_{\text{ledger}}$.

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

Main:

recv (Register, pid) from I/O: {Handling registration for parties}
 P.add(pid, 0)
reply (Register, pid)

recv (DeRegister, pid) from I/O: {Handling deregistration for parties}
 P.remove(pid_{cur}, _)
reply DeRegister

recv GetRegistered from I/O: {Output currently registered parties}
reply (GetRegistered, P)

recv (ClockUpdate, pid) from I/O: {Handling clock updates from participants}
 P.remove(pid_{cur}, _)
 P.add(pid_{cur}, 1)
if $\forall (pid, activated) \in P : activated = 1 \wedge F = 1$: {Update clock if all parties accepted update}
 $\tau \leftarrow \tau + 1$
for (pid, _) $\in P$ **do**:
 P.remove(pid, _)
 P.add(pid, 0)
 F = 0

recv ClockUpdate from (pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{ledger}}$: client): {Handling clock updates $\mathcal{G}_{\text{ledger}}$ }
 F = 1
if $\forall (pid, activated) \in P : activated = 1$: {Update clock if all parties accepted update}
 $\tau \leftarrow \tau + 1$
for (pid, _) $\in P$ **do**:
 P.remove(pid, _)
 P.add(pid, 0)
 F = 0

recv ClockRead from I/O or NET: {Handling reads from the clock}
reply (ClockRead, τ)

Figure 11: $\mathcal{G}_{\text{ledger}}$'s ideal clock functionality $\mathcal{G}_{\text{clock}}$ [3]

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

Participating roles: {client} Corruption model: <i>incorruptible</i> ^a Protocol parameters: - toMsglist	$\left\{ \begin{array}{l} \text{"Algorithm" that transforms output of extendPolicy from } \mathcal{G}_{\text{ledger}} \text{ to msglist format (including metadata).} \\ \text{extendPolicy may include format/data transformation.} \end{array} \right.$
^a \mathcal{A} can directly access $\mathcal{G}_{\text{ledger}}$ for corrupted parties via NET. Thus, we do not have to handle these cases via $\mathcal{W}_{\text{ledger}}$	

Description of M_{wrapper} :

Implemented role(s): {client} Subroutines: $\mathcal{G}_{\text{ledger}} : \text{client}, \mathcal{G}_{\text{clock}} : \text{clock}$ CheckID(<i>pid, sid, role</i>): Accept all messages with the same <i>sid</i> . Corruption behavior: - DetermineCorrStatus(<i>pid, sid, role</i>): <i>corrupted</i> \leftarrow corr (<i>pid, sid, $\mathcal{G}_{\text{ledger}} : \text{role}$</i>) return <i>corrupted</i>	$\left\{ \begin{array}{l} \text{Request corruption status at } \mathcal{G}_{\text{ledger}} \end{array} \right.$
Internal state: identities $\subset \{0, 1\}^* \times \mathbb{N}$, identities = \emptyset	$\left\{ \text{The set of participants and the round when they occurred first.} \right.$
MessagePreprocessing: recv (<i>pid_{cur}, sid_{cur}, role_{cur}, msg</i>) from I/O: <i>corrupted</i> \leftarrow corr (<i>pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{ledger}} : \text{role}_{\text{cur}}$</i>) if <i>corrupted</i> \wedge command is dedicated to $\mathcal{G}_{\text{ledger}}$: send <i>msg</i> to (<i>pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{ledger}} : \text{client}$</i>) if <i>msg</i> starts with Submit or Read: send Register to (<i>pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{clock}} : \text{clock}$</i>) wait for (Register, <i>sid, pid</i>) from $\mathcal{G}_{\text{clock}} : \text{clock}$ send Register to (<i>pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{ledger}} : \text{client}$</i>) wait for (Register, <i>sid, pid</i>) from $\mathcal{G}_{\text{ledger}} : \text{client}$ for (<i>pid, _</i>) \in identities do : send CorruptionStatus? to (<i>pid, sid_{cur}, $\mathcal{G}_{\text{ledger}} : \text{client}$</i>) wait for (CorruptionStatus, <i>corrupted</i>) if <i>corrupted</i> : identities.remove(<i>pid, _</i>) if (<i>pid_{cur}, _</i>) \notin identities: send ClockRead to (<i>pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{clock}} : \text{clock}$</i>) wait for (ClockRead, <i>round</i>) identities.add(<i>pid_{cur}, round</i>) recv <i>msg</i> from (<i>pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{ledger}}$</i>): send (<i>pid_{cur}, sid_{cur}, role_{cur}, msg</i>) to I/O	$\left\{ \begin{array}{l} \text{For all I/O inputs, ensure that the entity is registered.} \\ \text{Request corruption status at } \mathcal{G}_{\text{ledger}} \\ \text{See defined commands for } \mathcal{G}_{\text{ledger}} \text{ above} \\ \text{For corrupted parties: } \mathcal{W}_{\text{ledger}} \text{ acts as forwarder} \\ \text{Register unknown party before its first submit/read operation} \\ \text{Register party at } \mathcal{G}_{\text{clock}} \\ \text{Register party at } \mathcal{G}_{\text{ledger}} \\ \text{Remove corrupted identities} \\ \text{Record unknown parties} \end{array} \right.$
Main: recv (Submit, <i>msg</i>) from I/O: send (Submit, <i>msg</i>) to (<i>pid_{cur}, sid_{cur}, $\mathcal{G}_{\text{ledger}} : \text{client}$</i>)	$\left\{ \begin{array}{l} \text{Submission of a transaction from an uncorrupted party.} \\ \text{Forward the request to } \mathcal{G}_{\text{ledger}} \end{array} \right.$
$\left\{ \begin{array}{l} \text{In case that } \mathcal{G}_{\text{ledger}} \text{ sends a message via I/O, e.g., in the case that } \mathcal{A} \\ \text{triggers sending a message via I/O, the message is forwarded to I/O. Note} \\ \text{that this part is not executed if } \mathcal{W}_{\text{ledger}} \text{ waits for an answer, e.g., during} \\ \text{the reading process} \end{array} \right.$	

Figure 12: The $\mathcal{G}_{\text{ledger}}$ wrapper $\mathcal{W}_{\text{ledger}}$ (Pt. 1)

Description of the wrapper $\mathcal{W}_{\text{ledger}} = (\text{client})$ (cont.):

Main:	
recv (Read, msg) from I/O: send Read to (pid _{cur} , sid _{cur} , $\mathcal{G}_{\text{ledger}} : \text{client}$) wait for (Read, state) msglist \leftarrow toMsglist(state) reply (Read, msglist)	{Read from state from an uncorrupted identity. {Forward the request to $\mathcal{G}_{\text{ledger}}$. {Translate $\mathcal{G}_{\text{ledger}}$'s state format to $\mathcal{F}_{\text{ledger}}$'s message list format
recv (ClockUpdate, pid, sid) from NET: send ClockUpdate to (pid, sid, $\mathcal{G}_{\text{clock}} : \text{clock}$)	{ \mathcal{A} triggers clock updates for honest and corrupted parties {Trigger clock update.
recv (MaintainLedger, pid, sid) from NET: send MaintainedLedger to (pid, sid, $\mathcal{G}_{\text{ledger}} : \text{client}$)	{ \mathcal{A} triggers MaintainLedger for honest and corrupted parties
recv (GetCurRound) from NET: send ClockRead to (ϵ , sid _{cur} , $\mathcal{G}_{\text{clock}} : \text{clock}$) wait for (ClockRead, round) reply (GetCurRound, round)	{ \mathcal{A} requests GetCurRound
recv GetRegistered from I/O: send GetRegistered to (pid _{cur} , sid _{cur} , $\mathcal{G}_{\text{ledger}} : \text{ledger}$) wait for (GetRegistered, sid, P) from $\mathcal{G}_{\text{ledger}} : \text{ledger}$ output = {(pid, round) \in identities pid \in P} reply (GetRegistered, output)	{ $\mathcal{F}_{\text{ledger}}$ outputs only honest parties including the round they registered
recv DeRegister from I/O: send DeRegister to (pid _{cur} , sid _{cur} , $\mathcal{G}_{\text{clock}} : \text{clock}$) wait for DeRegister send DeRegister to (pid _{cur} , sid _{cur} , $\mathcal{G}_{\text{ledger}} : \text{ledger}$) wait for DeRegister identities.remove(pid _{cur} , _)	{ \mathcal{E} triggers deregistering of (pid, sid, role) from $\mathcal{G}_{\text{clock}}$ and $\mathcal{G}_{\text{ledger}}$ {Mimic behavior of $\mathcal{F}_{\text{ledger}}$
reply DeRegister	

Figure 13: The $\mathcal{G}_{\text{ledger}}$ wrapper $\mathcal{W}_{\text{ledger}}$ (Pt. 2)

Description of the subroutine $\mathcal{F}_{\text{submit}}^{\mathcal{G}_{\text{ledger}}} = (\text{submit})$:

Participating roles: {submit}	
Corruption model: <i>incorruptible</i>	
Protocol parameters:	
- validate	{Validation "algorithm" (from $\mathcal{G}_{\text{ledger}}$) that states whether a transaction is valid according to already ordered messages.}
- toBTX	{Algorithm that transforms an input tx to the format expected in $\mathcal{G}_{\text{ledger}}$, i.e., $(\text{msg}', \text{txID}, \tau_L, \text{pid})$. It includes housekeeping, e.g. txID is unique.}
- toState	{Algorithm that transforms $\mathcal{F}_{\text{ledger}}$'s message list format to $\mathcal{G}_{\text{ledger}}$'s state format. It may include housekeeping, e.g. generating of unique block ids.}

Description of M_{submit} :

Implemented role(s): {submit}	
Subroutines: $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$: update	
CheckID (<i>pid, sid, role</i>):	Accept all messages with the same <i>sid</i> .
Main:	
recv (Submit, <i>msg, internalState</i> ^a) from I/O:	{See Figure 5 for definition of <i>internalState</i> and the local variables it includes}
send (Preprocess, <i>msg, internalState</i>) to (<i>pid_{cur}, sid_{cur}</i> , $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$: update)	{Trigger $\mathcal{G}_{\text{ledger}}$'s pre-processing at $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ }
wait for (Preprocess, <i>msglist', buffer, leakage</i>)	{ $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ provides the up-to-date/validated data to $\mathcal{F}_{\text{submit}}^{\mathcal{G}_{\text{ledger}}}$ }
if validate(<i>toBTX(msg), toState(msglist')</i> , <i>buffer</i>) = true:	{Emulate $\mathcal{G}_{\text{ledger}}$'s validation behavior}
reply (validationProcessed, true, [<i>msg, leakage</i>])	{ \mathcal{A} receives all details of <i>msg</i> }
else:	
reply (validationProcessed, false, [<i>msg, leakage</i>])	{Leak <i>msg</i> to \mathcal{A} }
^a For brevity we use data from <i>internalState</i> with the local variant of the variable name from $\mathcal{F}_{\text{ledger}}$. This includes local variables such as <i>msglist, requestQueue, readQueue, and round</i> .	

Figure 14: $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$'s write/submit functionality $\mathcal{F}_{\text{submit}}^{\mathcal{G}_{\text{ledger}}}$

Description of the subroutine $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}$ = (read):

Participating roles: {read} Corruption model: <i>incorruptible</i> Protocol parameters: <ul style="list-style-type: none"> - $\text{windowSize} \in \mathbb{N}$ {The window size (number of blocks) of the sliding window. - $\delta \in \mathbb{N}$ {The upper bound for network delay in rounds.

Description of M_{read} :

Implemented role(s): {read} Subroutines: $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$: update pointer : $\{0, 1\}^* \rightarrow \mathbb{N} \cup \{\perp\}$ {Mapping from identities to last transaction in their state; initially \perp}
CheckID($pid, sid, role$): Accept all messages with same sid .
MessagePreprocessing: recv msg from I/O: For all parties pid that have been de-registered since the last call of $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}$ according to <i>transcript</i> (included in the incoming message or <i>internalState</i>), set $\text{pointer}[pid] \leftarrow \perp$.
Main: recv (InitRead, $msg, \text{internalState}^a$) from I/O: reply (InitRead, false, msg) {Model non-local reads}
recv (FinishRead, $msg, \text{outID}, \text{internalState}$) from I/O: {outID is the suggestedOutput from \mathcal{A}. For de-synced parties, it is the read output, for synced parties, it is a pointer to the output}
Let $(\text{pid}_{\text{cur}}, \text{roundRegistered})$ for some $\text{roundRegistered} \in \mathbb{N}$ be the unique tuple in identities.
send ProcessingOpen? to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$: update) {Enforce that pending updates are propagated to $\mathcal{F}_{\text{ledger}}$}
wait for (ProcessingOpen?, $response$) {If there are pending updates, they need to be propagated to $\mathcal{F}_{\text{ledger}}$}
if $response$: reply (FinishRead, \perp, ϵ) {Party is synced}
if $\text{roundRegistered} + \delta \leq \text{round}$: {\mathcal{A} is supposed to send the same or later pointer}
if $\text{outID} \notin \mathbb{N} \vee (\text{pointer}[\text{pid}_{\text{cur}}] \neq \perp \wedge \text{outID} < \text{pointer}[\text{pid}_{\text{cur}}])$:
reply (FinishRead, \perp, ϵ)
Let $(\text{outID}, \text{committingRound}, \text{tx}, \text{msg}, _, _)$ be the unique transaction from msglist with ID outID .
if $(\text{outID}, \text{committingRound}, \text{tx}, \text{msg}', _, _)$ does not exist in msglist: {\mathcal{A} is supposed to send an existing pointer}
reply (FinishRead, \perp, ϵ)
Let $b_{\text{outID}} \in \mathbb{N}$ be the block number of the block that contains the transaction with ID outID (according to $(\text{meta}, \text{cut})$ messages in msglist starting from Block 1).
Let $b_{\text{current}} \in \mathbb{N}$ be the block number of the current/most recent block in msglist .
if $b_{\text{outID}} + \text{windowSize} < b_{\text{current}}$:
reply (FinishRead, \perp, ϵ)
$\text{pointer}[\text{pid}_{\text{cur}}] \leftarrow \text{outID}$ {Update pointer}
Let output be the subsequence of msglist (only including the counter ctr , the submission round submissionRound , submitting party pid' , and the message body msg'') until (and including) message outID . {Map to output from $\mathcal{G}_{\text{ledger}}$}
reply (FinishRead, output, ϵ)
else: {Party is de-synced.}
reply (FinishRead, outID, ϵ) {\mathcal{A} determines the output. outID is suggestedOutput}
recv (CorruptedRead, $pid, msg, \text{internalState}$) from I/O:
reply (FinishRead, ϵ) {\mathcal{A} already knows the full chain}

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

Figure 15: $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$'s read functionality $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}$

Description of the subroutine $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}} = (\text{update})$:

Participating roles: {update}	
Corruption model: <i>incorruptible</i>	
Protocol parameters:	
– validate	{Validation “algorithm” (from $\mathcal{G}_{\text{ledger}}$) that states whether a transaction is valid according to already ordered messages}
– extendPolicy	{extendPolicy from $\mathcal{G}_{\text{ledger}}$, defines how state “evolves”}
– toMsglist	{“Algorithm” that transforms output of extendPolicy from $\mathcal{G}_{\text{ledger}}$ to msglist format (including metadata), extendPolicy may include format/data transformation}
– toState	{Algorithm that transforms $\mathcal{F}_{\text{ledger}}$ ’s message list format to $\mathcal{G}_{\text{ledger}}$ ’s state format, it may include housekeeping, e.g. generating of unique block IDs}
– ToBuffer	{Algorithm that transforms $\mathcal{F}_{\text{ledger}}$ ’s requestQueue format to $\mathcal{G}_{\text{ledger}}$ ’s buffer format. It may include housekeeping, e.g. generating of unique IDs}
– toBTX	{Algorithm that transforms an input tx to the format expected in $\mathcal{G}_{\text{ledger}}$, i.e., (msg’, txID, τ_L , pid). It includes housekeeping, e.g. txID is unique}

Description of M_{update} :

Implemented role(s): {update}	
CheckID (pid, sid, role):	Accept all messages with the same sid.
Internal state:	
– recordedActivations : {0, 1}*	{Emulated honest activations of $\mathcal{G}_{\text{ledger}}$ }
– msgListAppend ^P : {0, 1}*	{Cache for pending update msglist}
– UpdRequestQueue ^P : {0, 1}*	{Cache for pending update requestQueue}
Main:	
rcv (Update, [pending&new, msg], internalState ^a) from I/O:	{See Figure 5 for definition of internalState and the local variables it includes}
if msg ≠ (I_H^T , nxtBC, corrBuffer, τ_{state}), s.t. I_H^T , corrBuffer, τ_{state} match the format from $\mathcal{G}_{\text{ledger}}$, nxtBC is a sequence of nxtBC’s in the format of $\mathcal{G}_{\text{ledger}}$:	{Check message format}
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
if \exists a transaction of an uncorrupted pid in corrBuffer:	
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
recordedActivations.add(I_H^T)	{Update recordedActivations}
ProcessUpdate()	{See definition of ProcessUpdate below. The procedure gets the complete internal state of $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ and all currently used local variables as input. It may write to local and global variables or create new local variables.}
msgListAppend ^P ← \emptyset ; UpdRequestQueue ^P ← \emptyset	{Clear variables for pending upgrade}
reply (Update, msgListAppend, updRequestQueue, leakage)	{Return updates}
rcv (Update, [finalizePending], internalState) from I/O:	{ \mathcal{A} may finalize an update without triggering a new one}
msgListAppend ← msgListAppend ^P ; updRequestQueue ← UpdRequestQueue ^P	
leakage ← (msgListAppend, updRequestQueue)	
msgListAppend ^P ← \emptyset ; UpdRequestQueue ^P ← \emptyset	{Clear variables for pending upgrade}
reply (Update, msgListAppend, updRequestQueue, leakage)	{Return updates}
^a For brevity we use data from internalState with the local variant of the variable name from $\mathcal{F}_{\text{ledger}}$. This includes local variables such as msglist, requestQueue, readQueue, and round.	

Figure 16: $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ ’s update functionality $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ (Pt. 1)

Description of M_{update} (cont.):

Main:

recv (Preprocess, msg , $internalState$) **from** I/O: {Update preprocessing for Submit

send responsively (Preprocess, msg , $internalState$) **to** NET

wait for (Preprocess, I_H^T , $nxtBC$, $corrBuffer$, τ_{state})

while $msg \neq (I_H^T, nxtBC, corrBuffer, \tau_{state})$, **s.t.**

1. I_H^T , $corrBuffer$, τ_{state} match the format from $\mathcal{G}_{\text{ledger}}$,
2. $nxtBC$ is a sequence of $nxtBC$'s in the format of $\mathcal{G}_{\text{ledger}}$,
3. \nexists a transaction of an uncorrupted pid in $corrBuffer$ **do**

send responsively (Preprocess, msg , $internalState$) **to** NET {Query \mathcal{A} regarding state update

wait for (Preprocess, I_H^T , $nxtBC$, $corrBuffer$, τ_{state})

recordedActivations.add(I_H^T) {Update recordedActivations

ProcessUpdate() {See definition of inspace procedure below

$state \leftarrow msglist \cup msgListAppend$ {Output for $\mathcal{F}_{\text{submit}}$

$msgListAppend^P \leftarrow (msgListAppend)$; $UpdRequestQueue^P \leftarrow (updRequestQueue)$ {Record pending updates

reply (Preprocess, $state$, ToBuffer($updRequestQueue$) \cup $corrBuffer$, $leakage$) {Return data to $\mathcal{F}_{\text{submit}}$

recv ProcessingOpen? **from** I/O: {I/O can query whether there are pending

if $msgListAppend^P \neq \emptyset \vee UpdRequestQueue^P \neq \emptyset$: {state updates, e.g., , used by $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{ledger}}}$

reply (ProcessingOpen?, true)

else:

reply (ProcessingOpen?, true)

Procedures and Functions:

procedure ProcessUpdate() :

if $msgListAppend^P \neq \emptyset \vee UpdRequestQueue^P \neq \emptyset$:

$msglist.add(msgListAppend^P)$; $requestQueue \leftarrow requestQueue^c$ {Include pending updates in update process

$updRequestQueue \leftarrow \emptyset$

$N \leftarrow \text{extendPolicy}(I_H^T, \text{toState}(msglist), nxtBC, \text{ToBuffer}(requestQueue) \cup corrBuffer, \tau_{state})$ {Emulate $\mathcal{G}_{\text{ledger}}$ behavior during update

$msgListAppend \leftarrow msgListAppend^P$; $updRequestQueue \leftarrow UpdRequestQueue^P$

if $N \neq \epsilon$: {Process update, if extendPolicy produces an update

$msgListAppend.add(\text{toMsglist}(N, msglist))$ {Transform output of extendPolicy to msglist format

for all tx in $requestQueue$ **do:**

if $\neg \text{validate}(\text{toBTX}(tx), \text{toState}(msglist), \text{ToBuffer}(updRequestQueue) \cup corrBuffer)$: {Emulate $\mathcal{G}_{\text{ledger}}$'s update behavior

Remove entry of tx from $requestQueue$ and add it to $updRequestQueue$

$leakage \leftarrow (msgListAppend, updRequestQueue)$

Figure 17: $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$'s update functionality $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$ (Pt. 2)

Description of the subroutine $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}_{\text{ledger}}}$ = (updRnd):

Participating roles: {updRnd} Subroutines: $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$: update Corruption model: <i>incorruptible</i> Protocol parameters: <ul style="list-style-type: none"> - $\rho \in \mathbb{N}$ {The upper bound in rounds after which a honest tx should be in the state. - $\text{growthWindow} \in \mathbb{N}$ {The number of rounds the growthRate is related to - $\text{growthRate} \in \mathbb{N}$ {The number of "blocks" that should be added at in any - $\text{ensureLiveness} \in \{\text{true}, \text{false}\}$ {If true, liveness is ensured - $\text{ensureGrowth} \in \{\text{true}, \text{false}\}$ {If true, chain-growth is ensured

Description of M_{updRnd} :

Implemented role(s): {updRnd} Subroutines: $\mathcal{F}_{\text{update}}^{\mathcal{G}_{\text{ledger}}}$: update CheckID(<i>pid, sid, role</i>): Accept all messages with the same <i>sid</i> . Main: <ul style="list-style-type: none"> recv (UpdateRound, <i>msg, internalState</i>^a) from I/O: {See Figure 5 for definition of <i>internalState</i> and the local variables it includes send (ProcessingOpen?) to (<i>pid_{cur}, sid_{cur}, $\mathcal{F}_{\text{update}}$</i> : update) {Check whether there are pending state updated wait for (ProcessingOpen?, <i>response</i>) if <i>response</i>: {Round may not proceed if there are pending updates reply (UpdateRound, false, ϵ) if $\text{ensureLiveness} \wedge \exists (_, r, _) \in \text{requestQueue}$, s.t. $r \in \mathbb{N} \wedge r < \text{round} - \rho$: reply (UpdateRound, false, ϵ) if $\text{ensureGrowth} \wedge \text{round} \geq \text{growthWindow}$: if $\#(_, _, \text{meta}, \text{cut}, _, _) \text{ in the last growthWindow committing rounds in } \text{msglist} \text{ is smaller than } \text{growthRate}$: reply (UpdateRound, false, ϵ) if $\exists (\text{pid}, \text{responseID}, \text{round}, \text{msg}) \in \text{readQueue} \wedge (\text{pid}, _) \in \text{identities}$: {Check that all honest read requests in this round are processed reply (UpdateRound, false, ϵ) else: reply (UpdateRound, true, ϵ) <p>^aFor brevity we use data from <i>internalState</i> with the local variant of the variable name from $\mathcal{F}_{\text{ledger}}$. This includes local variables such as <i>msglist, requestQueue, readQueue, and round</i>.</p>
--

Figure 18: $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$'s time update functionality $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}_{\text{ledger}}}$

Description of the subroutine $\mathcal{F}_{\text{leak}}^{\mathcal{G}_{\text{ledger}}} = (\text{leak})$:

Participating roles: {leak} Corruption model: <i>incorruptible</i>

Description of M_{leak} :

Implemented role(s): {leak} CheckID(<i>pid, sid, role</i>): Accept all messages with the same <i>sid</i> . Main: <ul style="list-style-type: none"> recv (Corrupt, <i>pid, sid, internalState</i>) from I/O: {See Figure 5 for definition of <i>internalState</i> and the local variables it includes reply (Corrupt, ϵ) {\mathcal{A} already had full overview of the state
--

Figure 19: $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$'s leakage subroutine $\mathcal{F}_{\text{leak}}^{\mathcal{G}_{\text{ledger}}}$

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

<p>Participating roles: $\{\text{init}\}$ Corruption model: <i>incorruptible</i> Protocol parameters:</p> <ul style="list-style-type: none"> - extendPolicy - toMsglist 	<p>$\{\text{extendPolicy from } \mathcal{G}_{\text{ledger}}. \text{ Defines how state "evolves"}\}$ $\{\text{Algorithm that transfers output from extendPolicy to the format of msglist as specified in } \mathcal{F}_{\text{ledger}} \text{ (Figure 5)}\}$</p>
---	--

Description of M_{init} :

<p>Implemented role(s): $\{\text{init}\}$ CheckID($pid, sid, role$): Accept all messages with the same sid. Main:</p>	<p>$\{\mathcal{F}_{\text{init}}^{\mathcal{G}_{\text{ledger}}}$ runs extendPolicy wich may produce a Genesis block</p>
<p>recv Init from I/O: $N \leftarrow \text{extendPolicy}(\emptyset, \emptyset, \epsilon, \emptyset, \epsilon)$ $msglist \leftarrow \text{toMsglist}(N)$ reply (Init, \emptyset, $msglist$, \emptyset, $msglist$)</p>	<p>$\{\text{Leak data to } \mathcal{A}\}$</p>

Figure 20: The initialization functionality $\mathcal{F}_{\text{init}}^{\mathcal{G}_{\text{ledger}}}$ of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$.

E OTHER IDEAL BLOCKCHAIN FUNCTIONALITIES

In this section, we show that $\mathcal{F}_{\text{ledger}}$ indeed capture so-far published ideal ledger functionalities:

The ideal blockchain functionality \mathcal{G}_{PL} with privacy. Kerber et al. proposed a derivative of $\mathcal{G}_{\text{ledger}}$, called \mathcal{G}_{PL} [25], to analyze and prove security, including transaction privacy, of the proof-of-stake protocol Ouroboros Cryptsinous: \mathcal{G}_{PL} is designed for and tailored towards proof-of-stake blockchain protocols while also including mechanisms that allow for modeling privacy of transactions and during block generation. Using similar techniques as used in Theorem 3.1, we can instantiate $\mathcal{F}_{\text{ledger}}$ (denoted by $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$) in such a way that it captures the same security properties as the private blockchain functionality \mathcal{G}_{PL} .

THEOREM E.1 (INFORMAL). *Let $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$ be the instantiation of $\mathcal{F}_{\text{ledger}}$ that encodes the security properties provided by \mathcal{G}_{PL} . Then we have that \mathcal{G}_{PL} (plus a wrapper) realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$.*

We provide further details regarding \mathcal{G}_{PL} , its relation to $\mathcal{F}_{\text{ledger}}$, and provide a proof sketch below. Analogously to Corollary 3.2, this theorem directly implies that Ouroboros Cryptsinous realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$.

Besides $\mathcal{G}_{\text{ledger}}$ and \mathcal{G}_{PL} , there are also various other ideal functionalities for blockchains [17, 18, 28], all of which are similar to the first ideal functionality introduced by Kiayias et al. [28]. These functionalities are relatively simple compared to $\mathcal{G}_{\text{ledger}}$ and \mathcal{G}_{PL} : They offer just a submission and read interface. Once a party submits a transaction, it is immediately added to the global state. Furthermore, parties reading from that state always obtain the full global state. Badertscher et al. explain in [5] that “the proposed ledger-functionality (introduced in [28]) is too strong to be implementable by Bitcoin”, which holds true also for other blockchains from practice as well as for the closely related functionalities from [17, 18]. In other words, the main purpose of these functionalities is to serve as idealized setup assumptions for building and analyzing higher-level protocols. Clearly, one can instantiate $\mathcal{F}_{\text{ledger}}$ appropriately to offer the exact same setup assumptions.

E.1 \mathcal{G}_{PL} Realizes $\mathcal{F}_{\text{ledger}}$ (Sketch)

In this section, we sketch how $\mathcal{F}_{\text{ledger}}$ can be instantiated such that Kerber et al.’s ideal functionality for privacy in blockchains \mathcal{G}_{PL} [25] also realizes that instantiation of $\mathcal{F}_{\text{ledger}}$. Since \mathcal{G}_{PL} is a variant of $\mathcal{G}_{\text{ledger}}$, we first briefly describe the key differences between \mathcal{G}_{PL} and $\mathcal{G}_{\text{ledger}}$ and how \mathcal{G}_{PL} models privacy properties.¹³ Then, we explain how the subroutines of $\mathcal{F}_{\text{ledger}}$ need to be instantiated – we call the resulting instantiation $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$ in what follows – in order to prove that \mathcal{G}_{PL} realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$.

E.1.1 The Private Ledger Functionality \mathcal{G}_{PL} . The functionality \mathcal{G}_{PL} inherits most of its construction from $\mathcal{G}_{\text{ledger}}$ while mainly adding two additional features: (i) \mathcal{G}_{PL} allows for modeling privacy properties of blockchains, and (ii) \mathcal{G}_{PL} includes (initial) stake distributions. This is to enable capturing the Proof-of-Stake-based blockchain Ouroboros Cryptsinous [25].

On a technical level, Kerber et al. add the following new parameters to \mathcal{G}_{PL} to model privacy: (i) Lkg , a leakage algorithm, (ii) blindTx , an algorithm that blinds transaction content before this is leaked to \mathcal{A} , and (iii) blind , a blinding algorithm that hides private details of the state. To model privacy properties with \mathcal{G}_{PL} , each party gets a blind’ed response to its read requests. That is, it cannot directly access the full global state of \mathcal{G}_{PL} . Similarly, \mathcal{A} only gets access to a blind’ed version of \mathcal{G}_{PL} ’s state and all potential leakages, such as the leakage during transaction submission, are blinded via one of the additional parameters.

Furthermore, \mathcal{G}_{PL} includes an explicit (coin) ID generation interface Generate that allows to generate private IDs, e.g., private coin IDs which is necessary for Ouroboros Cryptsinous to, e.g., handle privacy during “mining”. Finally, Kerber et al. also change the behavior of some parameters compared to $\mathcal{G}_{\text{ledger}}$. In particular, (i) the activation list I_H^T stores blinded data as it is leaked later on to \mathcal{A} , (ii) validate gets the states of the participants, honest participants, and the set of generate IDs (coins) as additional input.

Note that \mathcal{G}_{PL} does not fix a particular stake update mechanism in its definition. \mathcal{G}_{PL} handles stake updates – in the spirit of $\mathcal{G}_{\text{ledger}}$ – by suitable instantiations of parameters/algorithms that can derive the (current) stake from the global state (including the generated coins) and store some information about the stake in the global state. \mathcal{G}_{PL} uses the same parameters/algorithms in similar ways as $\mathcal{G}_{\text{ledger}}$.

E.1.2 \mathcal{G}_{PL} Realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$. Now, we sketch the instantiation $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$ which is realized by \mathcal{G}_{PL} . Since \mathcal{G}_{PL} use similar/the same parameters as $\mathcal{G}_{\text{ledger}}$, this instantiation is very similar to the instantiation used for $\mathcal{G}_{\text{ledger}}$ in Appendix D and uses the same high-level idea, namely, running the parameters of \mathcal{G}_{PL} inside the subroutines of $\mathcal{F}_{\text{ledger}}$. There is, however, one conceptual difference between $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ and $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$: In $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$, we were able to abstract from some of the (not security relevant) technical details of $\mathcal{G}_{\text{ledger}}$, such as lists of honest activations, by letting the simulator take care of these aspects. This in turn resulted in a cleaner and simpler specification of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$. However, this is not possible for $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$: The information that \mathcal{G}_{PL} leaks strongly depends on the exact parameters that are used, so it is not fixed which exact information would be available to a simulator. Hence, the simulator actually cannot take over certain tasks. Instead, we use an alternative approach that we already mentioned in the discussion of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ in Appendix D, namely, we encode the full logic of \mathcal{G}_{PL} with all technical details within $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$.

¹³For a detailed presentation of \mathcal{G}_{PL} , we refer to the original paper [25].

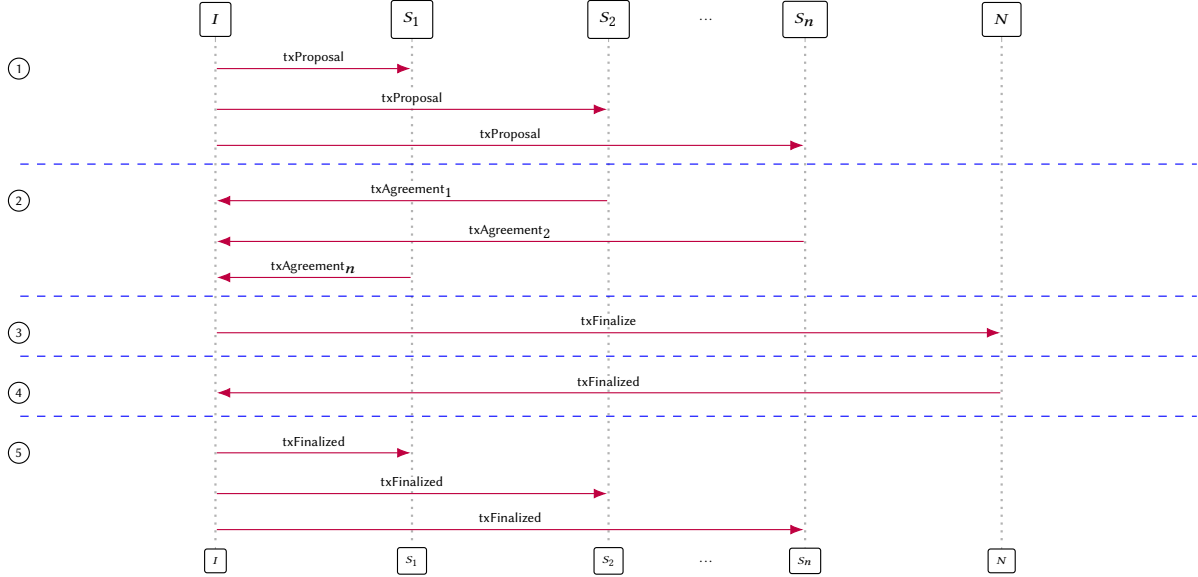


Figure 21: Example transaction submission in Corda with a transaction initiator I , signees S_1, \dots, S_n , and a notary N .

On a technical level, $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$ differs from $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{ledger}}}$ mainly in the following points: (i) $\mathcal{F}_{\text{init}}^{\mathcal{G}_{\text{PL}}}$ needs to distribute the initial stake distribution of \mathcal{G}_{PL} , (ii) $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{PL}}}$ needs a special handling for the Generate command which matches \mathcal{G}_{PL} , (iii) handling of leakage needs to be enhanced by \mathcal{G}_{PL} 's leakage parameters, and (iv) Read needs to blind outputs before they are delivered to the requestor.

Similarly to Section 3, resp. Appendix D, we then also have to add a wrapper \mathcal{W}_{PL} in front of \mathcal{G}_{PL} and $\mathcal{G}_{\text{clock}}$ to map interfaces and abstraction levels. \mathcal{W}_{PL} mainly works as $\mathcal{W}_{\text{ledger}}$ with the following major difference: As \mathcal{W}_{PL} maps a $\mathcal{F}_{\text{ledger}}$ commands to $\mathcal{G}_{\text{ledger}}$'s, resp. \mathcal{G}_{PL} 's format, \mathcal{W}_{PL} maps generate messages of the form (Read, [Generate, msg]) (dedicated for $\mathcal{F}_{\text{read}}^{\mathcal{G}_{\text{PL}}}$) to a Generate for \mathcal{G}_{PL} .

Using the instantiation and wrapper as sketched above, we then obtain the following result:

THEOREM E.2. (informal) Let \mathcal{G}_{PL} be the ideal blockchain functionality with arbitrary parameters such that all parameterized algorithms are deterministic. Let $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$ be the instantiation of $\mathcal{F}_{\text{ledger}}$ as described above, where the internal subroutines use the same parameters as \mathcal{G}_{PL} . Furthermore, if \mathcal{G}_{PL} is parameterized such that it provides liveness and/or chain-growth, then let the parameters of the subroutine $\mathcal{F}_{\text{updRnd}}^{\mathcal{G}_{\text{PL}}}$ in $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$ be set such that it also enforces the same properties. Then:

$$(\mathcal{W}_{\text{PL}} \mid \mathcal{G}_{\text{PL}}, \mathcal{G}_{\text{clock}}) \leq \mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$$

We do not provide a formal proof for this theorem. Intuitively, the theorem follows from the fact that the full logic of \mathcal{G}_{PL} is included in $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$, i.e., both functionalities behave in the same way (up to different abstraction levels which are mapped via the wrapper).

F FULL DETAILS: CORDA REALIZES $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$

In the following, we illustrate a flow between different parties in Corda using the submission of a transaction as an example. This is succeeded by full details for Theorem 4.1, including formal specifications of all machines and a formal proof of the theorem.

F.1 Corda Example Run

Figure 21 illustrates the process of submitting a transaction to Corda. The process is started by an initiator who submits, validates, and signs the transaction, creating a so-called *transaction proposal*. In Step 1, this proposal is sent to all signees who also validate the transaction and, if they agree with the transaction, also sign the proposal to signal consent. These signatures are returned to the initiator (Step 2), who then forwards the proposal with all signatures to the responsible notary (Step 3). After performing all of the checks mentioned above, the notary adds its own signature and returns the resulting finalized transaction to the initiator (Step 4). After validating the signature, the initiator adds this finalized transaction (and its dependencies) to its own partial view of the global transaction graph and forwards the finalized transaction to all signees (Step 5) who also validate all signatures and then add the transaction (and its dependencies) to their partial views.

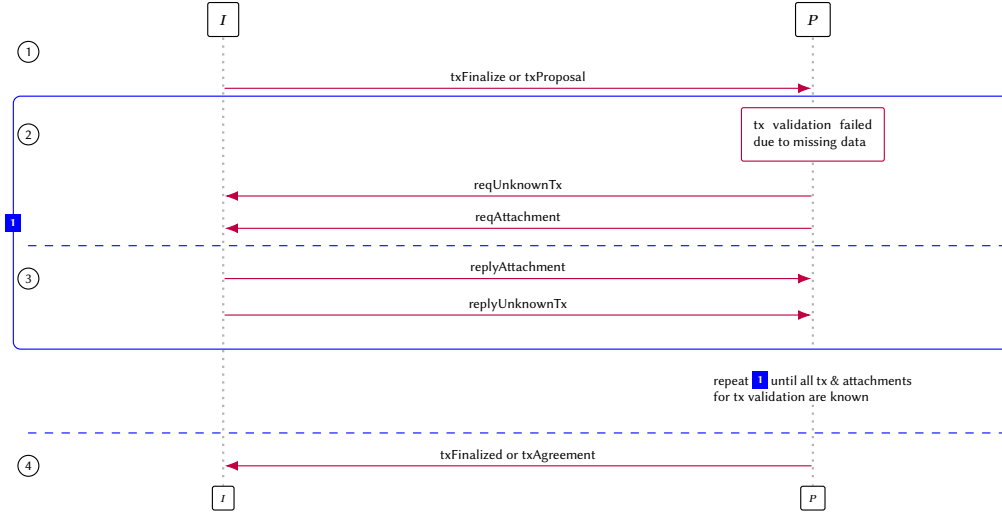


Figure 22: A notary or a client P requests related transactions and attachments from an initiator I if P is not aware of them.

Corda requires by design that the initiator can validate the transaction, i. e., he knows all dependencies (all of which must also have been validated at some point). This is achieved either by the owner of the initiating client having the dependency previously submitted to the ledger, or by signees sending the missing dependencies proactively to the initiator (which are validated and then added to its partial view of the global graph by the initiator) – for brevity, we say a party *pushes* a transaction to another party to denote proactively sending transactions.

Unlike the initiator, signees and the notary might not be aware of the full dependencies of a new transaction yet. In order to be able to validate the transaction between Steps 1 and 2 respectively Steps 3 and 4, they request, as depicted in Figure 22, any missing dependencies from the initiator (who must know the full dependencies). This process is iterated recursively until all dependencies have been received, validated, and added to their partial views of the global graph. If any information is missing or invalid, then the transaction submission protocol is aborted by that party without generating a signature.

F.2 The Corda Model \mathcal{P}^c

In this section, we provide additional details and highlights regarding our Corda model \mathcal{P}^c introduced in Section 4.2. We provide the formal definitions of \mathcal{P}^c in Figures 23 to 35 at the end of this section.

Remark: To simplify presentation, we introduced \mathcal{P}^c as protocol (client | notary, $\mathcal{F}_{unicast}$, \mathcal{F}_{cert} , \mathcal{F}_{ro}) in Section 4.2. Formally, the Corda protocol is defined as $\mathcal{P}^c = (\mathcal{P}_{ledger}^c : \text{client} \mid \mathcal{P}_{ledger}^c : \text{notary}, \mathcal{F}_{unicast}, \mathcal{F}_{cert}, \mathcal{F}_{ro})$ (more details below). As already outlined in Section 4.2, the Corda model \mathcal{P}^c mainly follows the specification of the Corda protocol as introduced in Section 4.1 and Section F.1. The core of \mathcal{P}^c is specified in the client machine \mathcal{P}_{client}^c and the notary machine \mathcal{P}_{notary}^c . One of their main duties consists in the resolution and validation of transactions and attachments as well as the sequence of stages/statuses a transaction passes from being first introduced to a client/notary until it is accepted into a party’s local state (see Table 1 for an overview of these statuses). These stages are encoded in a buffer in both machines, \mathcal{P}_{client}^c and \mathcal{P}_{notary}^c , namely $buffer_{TxSig}$. The largest part of the protocol is then the bookkeeping needed to implement the state machine. We discuss both the validation function/process and the sequence of stages in the following. We first focus on the client’s perspective and then explain the differences between clients and notaries.

We start by defining the different transaction types.

(Common) Transactions. In \mathcal{P}^c , we model transactions as $tx = (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal})$ (all of them are bitstrings but *initiator*, *signee*₁, . . . , *signee*_{*m*}, *notary*, *formerNotary* are expected to be PIDs). In *proposal*, we denote input states as $tx_{inputStates} = (txId, outputIndex)$ and output states as $tx_{outputStates}$. We denote by $m \in \mathbb{N}$ the number of signees in a transaction. In common transactions, we expect that *notary* = *formerNotary*.

Notary Change Transactions. We model notary change transactions as transactions of the same form as defined for common transactions above. In contrast to common transactions, we expect in notary change transactions that *notary* \neq *formerNotary*. Also, a notary change transaction has only *one* input state and generates exactly *one* output state that is identical to the input. We model that all notaries accept notary change transactions, i. e., the former notary agrees on the transaction as well as the new notary.

Reference States. Matching the design of Corda, we model the transactions that add reference states to the Corda graph via transactions of the same format as above [41]. To mark the transaction as a reference state, we model that *proposal* starts with the string “reference” followed by the input states $tx_{inputStates} = (txId, outputIndex)$. In what follows, there is barely any additional logic for handling reference state/transactions compared to “normal” transactions. But we emphasize that reference transactions cannot be consumed, i. e., they can be used in several transactions as input. Thus, re-using a reference state/transaction is not classified as double-spending. We reflect this in the function that classifies double-spending `isDoubleSpend` (see Figure 23).

Transaction Processing. Clients can learn about transaction proposals in two different ways:

1. They may get the transaction from *the network* if a peer wishes to send it/push it to them or they requested it for validating a transaction or
 2. via a submission call over *I/O* from the environment. In this case, they are either the initiator or a signee of that transaction
- Depending on how a client learns about a transaction, the processing differs whether one wants to validate or finalize the transaction.

In what follows, we describe the processing of a transaction in Corda including the different processing stages in more detail. We also provide an overview of the different processing stages in Table 1. The simplest case – Case 1. above – concerns transactions that are sent by other protocol members. The transaction is marked as `unrequested` or `requested` depending on whether it was sent as part of dependency resolution (when trying to validate another transaction) or proactively. Transactions in the `requested` stage directly pass to the set of valid transactions `verifiedTx` once successfully validated.

Concerning Case 2. above, there are two subcases: Firstly, if the environment sends a transaction via the submission interface to a client and the client is one of the transaction’s signees, the transaction is marked as `approved` and the client waits for the initiator to ask for a signature. Once the initiator asks for the signature, the transaction is resolved, i. e., the client may query the initiator for missing dependencies, validated and – if successfully validated – marked as `sign` (or `waitSign`, see below). In this stage, the client signs the transaction proposal to signal her agreement to the transaction and sends the transaction including her signature to the transaction initiator. Internally, the client marks the transaction as `expNotarization` meaning that she is waiting for a notarization/finalization of the transaction. When the client receives a valid (forwarded) notarization from the transaction initiator, she finalizes the transaction and moves it from `bufferTxSig` to the set of finalized transactions `verifiedTx`.

Secondly, if the client is the initiator of the transaction received via *I/O*, the client marks the transaction as `expColSigs`. Then, she checks that she can indeed process the transaction, i. e., she is aware of all dependencies, and validates it. Once validated, the client flags the transaction as `reqSigs`, signs the transaction, and asks the transaction’s signees to agree on it. At the client, the transaction stays in this stage until the client received all necessary signatures. At this point, the client marks the transaction as `reqNotarize` and sends it to the notary for notarization. Once she receives the notarization, she validates it and moves the transaction from `bufferTxSig` to `verifiedTx`. She distributes the notarization to all signees afterwards.

If in one of the steps above, the client, more specifically, a signee, is not available to validate a transaction, she queries the transaction initiator for the missing dependencies. In this case, there are two different cases to handle: (i) the transaction itself needs to be approved, i. e., it is currently in status `sign`. Then, the client moves the transaction to `waitSign` and continues processing as explained above as soon she has access to the transaction’s full dependencies. (ii) If the transaction tx is not in status `sign`, i. e., tx is necessary for validating another transaction, the client moves tx to `reqDeps`. She stores the objects, she requested for validating tx in the status `requested`. If the client has access to the full dependencies of a transaction in `reqDeps` or `requested`, she moves the transaction after a successful validation from `bufferTxSig` to `verifiedTx`.

A client stops further processing of a transaction if validation fails, i. e., she removes the transaction from `bufferTxSig`.

Remark: We note that the processing of common transactions, notary change transactions, and reference states/transactions are identical. Only functions used on these different transaction types cause a different handling, particularly, the algorithm `executeValidation`.

Transaction Validity Checks Explained. As part of the Corda protocol, clients and notaries always validate a transaction’s correctness (according to the protocol specification). During validation, the function `resolveTx` (see the \mathcal{P}_{client}^c specification in Figure 28) is responsible for collecting all necessary information to complete a transaction validation. Concretely, to validate a transaction, the client needs to know (and verify) all transactions and reference states the processed transaction depends on and all attachments used in these transactions. To resolve unknown data, `resolveTx`, i. e., the party executing `resolveTx`, sends `GetAttachment` and `SendTransactionFlow` messages (via $\mathcal{F}_{unicast}$) to the transaction initiator.

The \mathcal{P}_{client}^c function `validateAllLocal` acts as a housekeeping method. Every time new information is received this may allow finishing processing other transactions, e. g., if collecting their dependencies is completed or if a missing signature arrives. Concretely, `validateAllLocal` implements a fixed point iteration where all resolved transactions are validated and moved to `verifiedTx` (if they are valid) until no more transactions can be processed. `validateAllLocal`, i. e., the party executing it, may send messages via $\mathcal{F}_{unicast}$, e. g., if a signee can now successfully validate a transaction proposal and agree to it or if transaction initiators collected enough signatures to proceed with the notarization of a transaction.

The function `validate` is responsible for assessing the validity of transactions. It only assesses validity if all necessary dependencies are available. In the first step, `validate` checks whether a transaction has already been accepted (and is therefore in `verifiedTx`) or qualifies as double-spending. In both cases no further processing is necessary. Next, all clients mentioned in

the transaction need to exist according to the network map and have the correct roles, e. g., only entities with role notary are allowed to act as the transaction’s notary. Further, `validate` checks if the correct number of signatures exists. We define m to be the number of signees. If pid_{cur} is a signee or the initiator of the transaction, the number of signatures might be smaller than $m + 2$ as this transaction might still be in the signing process. Otherwise, either there has to be a signature for every signee, the initiator, and for the notary. Finally, if dependencies or attachments are missing, they are optionally requested which ends the execution of `validate`. If all checks are verified, the client collects all inputs to the transaction (or attachment) and executes the verification algorithm `executeValidation` (see Figure 23 for an intuition of the expected properties of `executeValidation`). The validation algorithm `executeValidation` receives as input all (direct) input transactions and reference states for the current transaction as well as their attachments.

Differences Between Clients and Notaries. Notaries act similarly to clients: they receive transactions but only from clients via $\mathcal{F}_{unicast}$. These arriving transactions pass several processing stages in $buffer_{TxSig}$ until they get accepted into `verifiedTx` (or declined). However, there are several noteworthy differences: Notaries never receive dependencies proactively and never take the active role of an initiator, but act similarly to a transaction signee. If transactions are valid, notaries always “accept” transactions. We do not model a notary’s “will” to agree on a transaction as we do for signees: notaries agree by design to all valid transactions. Once they checked the validity of a transaction – which includes that the transaction is not involved in a double-spending attempt – the notary signs the transaction and adds them to `verifiedTx`. Thus, notaries have much simpler processing compared to clients and also fewer processing statuses. We selected status names such that they correspond with client states (also see Table 1).

F.2.1 Further Notable Details of \mathcal{P}^c . Here, we list further notable details of \mathcal{P}^c .

- All $pids$ of \mathcal{P}^c participants are expected to be prefixed by their role, i. e., pid is typically of the form `client||pid'` or `notary||pid'` where $pid' \in \{0, 1\}^*$.¹⁴

Note that we model Corda transactions as $tx = (initiator, [signee_1, \dots, signee_m], notary, formerNotary, proposal)$ (all of them are bit strings). We denote in $proposal$ input states as $tx_{inputStates} = (txId, outputIndex)$ and output states as $tx_{outputStates}$. We denote by $m \in \mathbb{N}$ the number of signees in a transaction.

F.3 Full Details: Corda realizes \mathcal{F}_{ledger} instantiation for Corda

As already stated in Section 4.3, the instantiation \mathcal{F}_{ledger}^c of \mathcal{F}_{ledger} is the protocol $(\mathcal{F}_{ledger} | \mathcal{F}_{init}^c, \mathcal{F}_{submit}^c, \mathcal{F}_{update}^c, \mathcal{F}_{read}^c, \mathcal{F}_{updRnd}^c, \mathcal{F}_{leak}^c, \mathcal{F}_{storage}^c)$, with formal definitions of the instantiated subroutines provided in Figures 36 to 49 at the end of this section.

F.4 Final Result: Corda Realizes \mathcal{F}_{ledger}^c

Having explained both the ideal protocol \mathcal{F}_{ledger}^c (see Section 4.3 and Appendix B) and its intended realization \mathcal{P}^c (see also Appendix F.2), we can now formally state and prove the main result of this section: Theorem 4.1 (also, see Figure 4 in Section 4.3).

Before going into details, we first define different knowledge notations and the notion of synchronized states that we use in the proof of Theorem F.1. To define these terms, we use the following abbreviations from Theorem F.1: \mathcal{R} denotes the real protocol, i. e., \mathcal{P}^c , \mathcal{I} denotes the ideal protocol \mathcal{F}_{ledger}^c , and \mathcal{R}' denotes the version of \mathcal{R} which is simulated in the simulator.

Knowledge and State Synchronization During Simulation: During the upcoming proof, we often rely on the induction hypothesis that the “states” of \mathcal{R}' (and thus also in \mathcal{R}) and \mathcal{I} are “synchronized” or simply both are “synchronized”. The notation of synchronized states of an honest entity expresses that – at a certain point during the run – the “knowledge” of every honest entity regarding its transaction graph and attachments in \mathcal{R}' is a “subset” of the entities knowledge in \mathcal{I} .¹⁵ In detail, let $entity$ be an honest entity at some point during a run of $\{\mathcal{S}, \mathcal{I}\}$ and $txIDs^{\mathcal{R}}$ the transaction IDs extracted from `verifiedTx`, $attachmentIDs^{\mathcal{R}}$ the set of attachment IDs extracted from attachments, and $txIDs_{buffered}^{\mathcal{R}}$ the transaction IDs extracted from $buffer_{TxSig}$ from the entities state in \mathcal{R}' , i. e., the simulated instance of the machine $entity$ (\mathcal{P}_{client}^c or \mathcal{P}_{notary}^c). Let $txIDs^{\mathcal{I}}$ be the set of transaction IDs such that

- $txIDs^{\mathcal{I}}$ is the set of transaction IDs extracted from \mathcal{F}_{ledger}^c ’s message list in \mathcal{I} as follows:
 1. For all transactions in `msglist` where $entity$ is the initiator, signee, or notary of the transaction, add the transaction ID to $txIDs^{\mathcal{I}}$.
 2. For all push operations where $entity$ is the receiver of the operation, i. e., for all message list entries that contain a message of the form $(txID, pids, pid)$, add $txID$ to $txIDs^{\mathcal{I}}$.
 3. For all transactions in `requestQueue` where $entity$ is the initiator or signee (or notary) of the transaction. If there exists a submit transaction of this transaction submitted by the transaction’s initiator in `requestQueue`, then add the transaction to $txIDs^{\mathcal{I}}$.
 4. Do recursively: for all transaction IDs in $txIDs^{\mathcal{I}}$, add their input transaction IDs from `msglist` to $txIDs^{\mathcal{I}}$.
 5. All leaked transactions/transactions generate by \mathcal{A} are in $txIDs^{\mathcal{I}}$ as they are not private.

¹⁴For $a, b \in \{0, 1\}^*$, $a || b := ab$ denotes the concatenation of a and b .

¹⁵We do not expect equality of both states as \mathcal{I} abstracts network delay during communication and might have access to data that still needs to be delivered in \mathcal{R}' .

Transaction Status	Explanation	Processing
\perp	Error status	None
unrequested	Transaction was pushed to the client, but she did not request it.	None
requested	Intermediate step for processing requested dependent objects.	If full dependencies are available, validate transaction, otherwise request missing dependencies
approved	Signee received the transaction via I/O from \mathcal{E} , i.e., the client agrees when the initiator asks for agreement to or a signature of the transaction.	Wait for SignTransactionFlow message
expColSigs	Intermediate step for initiators, transaction waits for processing in the CollectSignaturesFlow	Wait for CollectSignaturesFlow, signee signatures needed
reqSigs	Initiator is waiting for signee agreement/signature.	Collect signatures, then start notarization
reqDeps	Signee/notary currently does not have access to the full dependencies of the transaction and queried the transaction initiator for missing dependencies.	Wait for missing dependencies from the initiator, start validation or agreement process, after the collection of all dependencies (transactions and attachments)
sign	Intermediate step for signees, the transaction is queued for agreement/signing.	Sign transaction and send a response to the initiator
waitSign	Intermediate step for signees and notaries, the transaction is queued for agreement/signing but dependencies are missing.	Collect dependencies
expNotarization	Signee agreed to the transaction and has sent her approval to the initiator.	Wait for notarization forwarded via RecvFinalityFlow
reqNotarization	Initiator waits for notarization.	Finalize transaction, i.e., move the transaction to verifiedTx, forward transaction to signees via RecvFinalityFlow

Table 1: \mathcal{P}^c : Transaction statuses and following processing steps in these statuses

- $attachmentIDs^{\mathcal{I}}$ is the set of all attachment IDs extracted from $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s variables msglist and requestQueue such that *entity* submitted the attachment to $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$. Additionally, for all transactions in $txIDs^{\mathcal{I}}$, extract the attachment IDs from the transactions and add them to $attachmentIDs^{\mathcal{I}}$.
- $txIDs_{\text{buffered}}^{\mathcal{I}}$ is the set of transaction IDs extracted from $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s requestQueue where *entity* is signee, initiator, or notary of the transaction or the entry in requestQueue encodes a transaction push operation to *entity*.

We say \mathcal{R}' and \mathcal{I} are *synchronized* (at a point in time during a run of $\{\mathcal{S}, \mathcal{I}\}$) iff for every honest entity it holds true that:

$$txIDs^{\mathcal{R}'} \subset txIDs^{\mathcal{I}} \wedge attachmentIDs^{\mathcal{R}'} \subset attachmentIDs^{\mathcal{I}}$$

We now define/summarize sets, we use during the proof of Theorem F.1:

Knowledge Sets:

For referencing, we use the following sets:

- $txIDs^{\mathcal{R}}(pid, sid, role)$,
- $attachmentIDs^{\mathcal{R}}(pid, sid, role)$,
- $txIDs^{\mathcal{I}}(pid, sid, role)$, and
- $attachmentIDs^{\mathcal{I}}(pid, sid, role)$.

They are defined as above but explicitly referencing their owner $(pid, sid, role)$.

Real/Potential Knowledge:

We call the set $txIDs^{\mathcal{R}}(entity) \cup attachmentIDs^{\mathcal{R}}(entity)$ the *real knowledge* of *entity* (in \mathcal{R}'). We call the set $txIDs^{\mathcal{I}}(pid, sid, role) \cup attachmentIDs^{\mathcal{I}}(pid, sid, role)$ the *potential knowledge* of *entity* (in \mathcal{I})

Active Knowledge:

In \mathcal{I} , we call the union of the set of an *entity*'s known transactions ($knowTransactions(entity) = \{txID \mid (pid, txID) \in knownTransaction\}$) and the set of its known attachments (defined as $knowAttachments(entity) = \{ID^a \mid (pid, ID^a) \in knownAttachments\}$) the *active knowledge* of *entity*.

Current Knowledge:

We call the set of all transactions and attachment ID that an entity in \mathcal{R}' has access to before and including the current activation the *current knowledge* of the entity at this point during the run.

With this additional notation, we now state and prove our security and privacy statement for Corda.

THEOREM F.1. *Let $\eta \in \mathbb{N}$ be the security parameter and $\Sigma = (\text{gen}(1^\eta), \text{sig}, \text{ver})$ be an EUF-CMA secure signature scheme. Let $\mathcal{P}^{\mathbf{c}}$ be the Corda protocol that uses the signature scheme Σ , using parameters/further algorithms as explained in Figure 23 and further parameters selected arbitrarily but such that all parameterized algorithms are deterministic and in polynomial time. Let $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ be the instantiation of $\mathcal{F}_{\text{ledger}}$ as described above, where the internal subroutines use the same parameters as $\mathcal{P}^{\mathbf{c}}$. Then:*

$$\mathcal{P}^{\mathbf{c}} \leq \mathcal{F}_{\text{ledger}}^{\mathbf{c}}$$

PROOF. As part of the proof, we first define a responsive simulator \mathcal{S} such that the real world running the protocol $\mathcal{R} := \mathcal{P}^{\mathbf{c}}$ is indistinguishable from the ideal world running $\{\mathcal{S}, \mathcal{I}\}$, with the protocol $\mathcal{I} := \mathcal{F}_{\text{ledger}}^{\mathbf{c}}$, for every ppt environment \mathcal{E} .

The simulator \mathcal{S} is defined as follows: it is a single machine that is connected to \mathcal{I} and the environment \mathcal{E} via their network interfaces. In a run, there is only a single instance of the machine \mathcal{S} that accepts and processes all incoming messages. The simulator \mathcal{S} internally simulates the realization \mathcal{R} , including its behavior on the network interface connected to the environment, and uses this simulation to compute responses to incoming messages (see below for details as Corda's privacy properties hide several details from \mathcal{S}). For ease of presentation, we will refer to this internal simulation by \mathcal{R}' . Before we explain the simulator in detail, we explain how the simulator deals with Corda's privacy.

Design Rational of \mathcal{S} and Handling of Blinded Data: Due to Corda's privacy properties, \mathcal{S} does not have full access to the data of submitted transactions/attachments and read operations of honest entities. Nevertheless, \mathcal{S} follows the same approach as the simulator in the proof of Theorem D.1: \mathcal{S} internally simulates \mathcal{R} . In contrast to Theorem D.1, \mathcal{S} has to use leaked data from $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ to simulate a "blinded" version of \mathcal{R} . By definition, $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ leaks many details of a transaction including (i) the transaction's ID, (ii) the length of the transaction, (iii) initiator entity, signee entities, and (former) notary entities, (iv) transaction inputs and outputs (only IDs), and (v) used attachments (only IDs). This means that Corda mainly hides the contents of a transaction body.

During the simulation, \mathcal{S} replaces the original (blinded) transaction which has not been leaked so far with a unique dummy transaction (identified by the original transaction ID). The dummy transaction has the expected Corda transaction format and uses the leaked information from $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$, e. g., initiator, signees, and notaries, where possible. The dummy transaction includes all the leaked data such that \mathcal{R}' can directly execute format checks on it. The unknown part of the dummy transaction is filled with random bitstrings such that the length of the dummy transaction matches the length of the original transaction. For attachments,

$\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ leaks the attachment ID and the length of the attachment itself. Thus, \mathcal{S} uses a unique random bit string to replace the attachment definition during the simulation and identifies the attachment by its original ID. Instead of using the original transaction/attachment in the simulation, \mathcal{S} usually uses the dummy transaction/attachment.

There are mainly three issues/tasks, \mathcal{S} faces due to this “blinded” simulation:

Firstly, after the corruption of an entity, \mathcal{S} has to provide the appropriate leakage to an adversary \mathcal{A} . Thus (and also for further reasons), we define \mathcal{S} such that it keeps corruption in \mathcal{R}' and \mathcal{I} in sync. When corrupting an entity *entity* in \mathcal{I} , \mathcal{S} gets full access to the entity’s internal state which includes full details and plaintexts of (i) the attachments known by *entity*, (ii) transaction subgraphs in which *entity* is involved in (from the transaction *entity* is directly involved down to all dependencies in the subgraph in the direction of the transactions input transactions until one reaches the initial issuance transactions), additionally (iii) all subgraphs about which *entity* was directly informed via a push message from another entity (also following the direction of the input transactions in the subgraph).¹⁶ As soon as $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ leaks attachment or transaction details, \mathcal{S} gets access to them and replaces the dummy transaction/attachment data in \mathcal{R}' with the actual transaction/attachment data.¹⁷ Then, \mathcal{S} extracts the leaked data for the corrupted party from \mathcal{R}' (as specified in $\mathcal{P}_{\text{client}}^{\mathbf{c}}$ and $\mathcal{P}_{\text{notary}}^{\mathbf{c}}$).

Secondly, \mathcal{S} cannot validate blinded transactions. However, an (honest) initiator of a transaction leaks whether she interpreted a transaction as valid. If \mathcal{S} only has access to blinded transaction data, instead of calling the `executeValidation` during the simulation of \mathcal{R}' , \mathcal{S} reuses the information whether the submission was accepted by $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ and replaces the execution of `executeValidation` with this output. If there is no appropriate leakage provided by $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$, \mathcal{S} queries \mathcal{I} regarding the validity of a transaction in the context of an entity’s current state.

Thirdly, if \mathcal{R}' , or more specifically, a corrupted entity, uses the random oracle \mathcal{F}_{ro} to generate a fresh transaction or attachment ID or to retrieve an ID, \mathcal{S} checks whether the contains a transaction or an attachment. If it is one of both, \mathcal{S} forwards the \mathcal{F}_{ro} request as `Update[GetID]` message to $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$. $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ will request \mathcal{S} for a unique ID if $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ is not aware of the requested bitstring, i. e., there was no \mathcal{F}_{ro} request regarding this bitstring before. \mathcal{S} simulates the request in \mathcal{F}_{ro} and provides an unambiguous hash. Otherwise and also after setting the ID, $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ will provide the ID to \mathcal{S} . The simulator forwards the ID as the answer of \mathcal{F}_{ro} to \mathcal{R}' or to the corrupted entity that queried \mathcal{F}_{ro} . If the request to \mathcal{F}_{ro} is neither a transaction nor an attachment, \mathcal{S} generates a random and unique hash/ID (not clashing with one from the generated IDs in $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$) and forwards it to the requestor.

Based on the above information, the simulation runs as follows:

Network communication from/to the environment

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

Corruption handling

- The simulator \mathcal{S} keeps the corruption status of entities in \mathcal{R}' and \mathcal{I} synchronized. That is, whenever an entity in \mathcal{R}' starts to consider itself corrupted, the simulator first corrupts the corresponding entity in \mathcal{I} before continuing the simulation.
- As explained above, \mathcal{S} applies leaked attachment and transaction data to \mathcal{R}' by (i) replacing dummy values by actual values and (ii) replacing/regenerating signatures on dummy values with signatures of original values in \mathcal{R}' .
- If an entity is explicitly corrupted, \mathcal{S} also corrupts the party in $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$. In turn, $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ sends the leakage caused by the corruption to \mathcal{S} . Then, \mathcal{S} applies the leakage to \mathcal{R}' as explained above.
- If \mathcal{S} receives messages from/for corrupted entities in \mathcal{I} , \mathcal{S} forwards these messages in the name of the corrupted entity on the network interface to the environment. Conversely, whenever a corrupted entity of \mathcal{R}' wants to output a message to a higher-level protocol, \mathcal{S} instructs the corresponding entity of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ to output the same message to the higher-level protocol.
- \mathcal{S} maps requests from corrupted *entity* to \mathcal{F}_{ro} depending on their content differently: (i) if *entity* wants to query \mathcal{F}_{ro} for an transaction or attachment ID, \mathcal{S} maps the request to an `Update` call with “flavor” `[GetID]` and forwards the request to $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$. \mathcal{S} maps $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ ’s responses to the answer format of \mathcal{F}_{ro} , and returns it to the corrupted/requesting party (see above). (ii) Otherwise, the requests are handled by \mathcal{S} as explained above: \mathcal{S} simulates \mathcal{F}_{ro} such that the outputs do not clash with the `GetID` interface from above. (iii) Messages from corrupted parties to $\mathcal{F}_{\text{unicast}}$ are forwarded to the simulation \mathcal{R}' .

Current state queries to \mathcal{S}

$\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ may frequently asks \mathcal{S} regarding the current state of an entity. In this case, \mathcal{S} extracts the known transaction and attachment IDs from the entities state in \mathcal{R}' and forwards all transaction/attachment IDs that the entity has seen to $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$. Additionally, \mathcal{S} records for all entities which transaction/attachment IDs have been in their state before.

Transaction submission

Whenever an honest *entity* receives a request (`Submit, msg`) to submit a new transaction *msg*, the subroutine $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ processes

¹⁶The leaked data is an over-approximation of the corrupted entity’s knowledge where instant delivery of dependent transactions and attachments is assumed.

¹⁷Replacing blinded data by original data particularly includes the regeneration and replacement of all signatures in \mathcal{R}' which were generated based on the blinded data.

the three possible message formats (i) a transaction proposal (including reference transactions and notary change transactions), (ii) submission of an attachment, and (iii) sharing/pushing of an existing transaction from one party to another party and leaks appropriate data for \mathcal{S} to distinguish the above operations. We assume in the following that the handled objects are valid. Otherwise, \mathcal{S} simulates in \mathcal{R}' the input of the object and simulates that validity checks fail.

Case (i): Assuming that none of the entities involved in the submitted transaction is corrupted, \mathcal{S} receives a blinded version of the transaction, generates a dummy version of the transaction as explained above, and forwards the dummy transaction in a submission request to \mathcal{R}' . In the case that one of the involved entities in the transaction is corrupted and the submit request was received by one of the transaction's signees, $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ leaks the full transaction to \mathcal{S} .¹⁸ If the submit was requested by the initiator (and one of the involved entities is corrupted), $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ leaks the transaction graph below the submitted transaction including the used attachments to \mathcal{S} . Then, \mathcal{S} replaces blinded data in \mathcal{R}' with leaked data and forwards the submitted transaction (in plain) to \mathcal{R}' . In particular, the input to \mathcal{R} may trigger internal communication in \mathcal{R}' via $\mathcal{F}_{\text{unicast}}$. See below for details.

Case (ii): \mathcal{S} triggers that the attachment is stored in the submitting entities state by sending an `Update[attachment]` message to $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$. When \mathcal{S} gets reactivated after the submission, he generates a (unique) dummy version of the attachment and simulates its input to \mathcal{R}' .

Case (iii): When \mathcal{S} receives the leakage from the transaction exchange/push operation, he triggers the exchange according to the leaked data in \mathcal{R}' . In particular, the input to \mathcal{R} may trigger internal communication in \mathcal{R}' via $\mathcal{F}_{\text{unicast}}$ (see below for details). In the case that $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ leaks data during Case (iii), it updates \mathcal{R}' as explained above.

After simulating the (blinded) input to the honest *entity* in \mathcal{R}' , \mathcal{S} outputs the result of the activation from \mathcal{R} to the environment.

Read requests

Whenever an honest entity receives a request of the form `(Read, msg)` to read from the global state, $\mathcal{F}_{\text{ledger}}$ forwards this (local) read request to \mathcal{S} and waits to receive a suggested output in the form of a list of transactions. \mathcal{S} simulates the read request in \mathcal{R}' . Note that the request is independent of the input `msg`. \mathcal{S} extracts the transaction IDs from the simulated read request and forwards the suggested output to \mathcal{I} .

Simulation of communication via $\mathcal{F}_{\text{unicast}}$ and dependencies

In many cases, an activation of \mathcal{R}' leads to messages sent via $\mathcal{F}_{\text{unicast}}$ in \mathcal{R}' . To simulate the Corda protocol properly, \mathcal{S} does internal bookkeeping regarding the status of the different messages and transactions depending on these transactions (already done in \mathcal{R}' in $\mathcal{P}_{\text{client}}^{\mathbf{c}}$ and $\mathcal{P}_{\text{notary}}^{\mathbf{c}}$). If \mathcal{A} triggers the delivery of a message via `Deliver`, \mathcal{S} forwards this request to the simulated version of $\mathcal{F}_{\text{unicast}}$ and processes it in \mathcal{R}' .

Further details

- \mathcal{S} keeps the clocks/rounds of \mathcal{R}' and $\mathcal{F}_{\text{ledger}}$ synchronous. That is, before continuing the simulation, \mathcal{S} sends `UpdateRound` to $\mathcal{F}_{\text{ledger}}$ whenever a round update in the simulated $\mathcal{F}_{\text{unicast}}$ is performed.
- If $\mathcal{F}_{\text{ledger}}$ queries \mathcal{S} for ID generation via `GenerateID`, \mathcal{S} selects a unique ID (length of the ID is η) and returns it to $\mathcal{F}_{\text{ledger}}$. Note that \mathcal{S} does internal bookkeeping for the IDs to keep them unique and to prevent collisions with other \mathcal{F}_{To} queries.

This concludes the description of the simulator. It is easy to see that (i) $\{\mathcal{S}, \mathcal{I}\}$ is environmentally bounded¹⁹ and (ii) \mathcal{S} is a responsive simulator for \mathcal{I} , i. e., restricting messages from \mathcal{I} are answered immediately as long as $\{\mathcal{S}, \mathcal{I}\}$ runs with a responsive environment. We now argue that \mathcal{R} and $\{\mathcal{S}, \mathcal{I}\}$ are indeed indistinguishable for any (responsive) environment $\mathcal{E} \in \text{Env}(\mathcal{R})$.

Now, let $\mathcal{E} \in \text{Env}(\mathcal{R})$ be an arbitrary but fixed environment. Before we argue in detail regarding the indistinguishability of \mathcal{R} and \mathcal{I} , we briefly analyze the possibility of distinguishing \mathcal{R} and \mathcal{I} based on generated IDs and signatures.

Collisions of IDs: \mathcal{R} uses a random oracle \mathcal{F}_{To} to generate IDs (in the length of the security parameter η) which may cause a collision in the IDs whereas \mathcal{S} generates unique IDs without collisions for \mathcal{I} . Observe that the possibility that \mathcal{E} distinguishes between \mathcal{R} and \mathcal{I} based on an ID collision is negligible in η .

Signatures: Similarly, \mathcal{E} could use $\mathcal{F}_{\text{cert}}$ to distinguish between \mathcal{R} and \mathcal{I} , e. g., by guessing transaction, signee, and signature pairs and verifying then at $\mathcal{F}_{\text{cert}}$. However, due to Σ 's EUF-CMA security, the possibility of guessing such a triple is negligible in η . Thus, the probability that \mathcal{E} may distinguish between \mathcal{R} and \mathcal{I} based on signatures is negligible in η as well.

In the following, we only consider runs where the events explained above do not occur. We will go over all possible interactions on the network and the I/O interface and argue, by induction, that all of those interactions result in identical behavior towards the environment, i. e., are also indistinguishable. At the start of a run, there were no interactions on the network or I/O, interface yet. Thus, the base case holds true. In the following, assume that all network and I/O interactions so far have resulted in the same behavior visible towards the environment in both the real and ideal world.

Interaction with honest entities via I/O: Firstly, we show that the I/O behavior during the simulation towards the environment is indistinguishable from the I/O behavior of \mathcal{R} . For brevity and readability, we often use expressions like “ \mathcal{S} pushes the attachment/the

¹⁸As signees/notaries may need to query further knowledge at the initiator, the transaction is the only thing they have access to for sure.

¹⁹As all algorithms are in polynomial time and `executeValidation` ensures that the execution of attachments finishes in polynomial time.

transaction to \mathcal{I} ” although we are only aware of the leakage of the mentioned attachment. The expression usually means that the operation regarding a blinded transaction/attachment is executed identifying the attachment/transaction by its ID.

Submission requests: By construction of \mathcal{R} and \mathcal{I} , submission requests do not directly result in output to the environment. But they might influence future read requests. Thus, the main goal here is to prove that \mathcal{R}' and \mathcal{I} , more specifically, the states of all honest parties, stay synchronized after transaction submission. Note that \mathcal{S} can distinguish the three following cases due to the format of the leakage provided by \mathcal{I} .

Transaction submission: There are two basic cases to distinguish: (i) the submitting party is the initiator of the submitted transaction or (ii) the submitting party is a signee of the transaction.²⁰ In Case (ii), \mathcal{I} 's leakage informs \mathcal{S} whether the submitted transaction was added to the submitting entity's buffer, i. e., $txIDs_{buffered}^{\mathcal{I}}(entity)$. If the transaction was added to the entities buffer at \mathcal{I} , the transaction will be added to the entities state in \mathcal{R}' (namely to the set $txIDs_{buffered}^{\mathcal{R}}(entity)$) as well (as \mathcal{R}' and \mathcal{I} execute the same steps during the submission of a transaction from a signee. Further, all necessary data to perform these checks in the simulation are included in the leakage from \mathcal{I} to \mathcal{S} , except for the result of the execution of `executeValidation` which is, however, part of the leakage of $\mathcal{F}_{ledger}^{\mathcal{C}}$). Note that – if the transaction initiator already processed and accepted the submitted transaction – this step may add the transaction, its subgraph, and the used attachments to $txIDs^{\mathcal{I}}(entity)/attachmentIDs^{\mathcal{I}}(entity)$. However, \mathcal{R}' and \mathcal{I} are still synchronized.

Before Case (i) occurs, i. e., if the submitting entity is the transaction's initiator, \mathcal{I} will query \mathcal{S} for the initiator's *current knowledge*. \mathcal{S} replies with the current knowledge of the entity (in the form of IDs) as explained above. As \mathcal{R}' and \mathcal{I} were synchronized before the operation, they stay synchronized after the operation as well (we will step over the different Corda flows below and show that this conclusion is valid) and \mathcal{I} accepts the current knowledge from \mathcal{S} : Observe that the current state in \mathcal{R}' is increasing in this situation and \mathcal{S} keeps track of declined transactions and attachments: the current state delivered by \mathcal{S} to \mathcal{I} (i) is always a superset of the previous current state and (ii) only contains valid transactions, and (iii) the current knowledge of *entity* in \mathcal{R}' is by induction hypothesis a subset of \mathcal{I} potential knowledge before the activation. We can thus conclude that \mathcal{I} will accept \mathcal{S} 's update and that \mathcal{R}' and \mathcal{I} are still synchronized after \mathcal{S} pushed the current knowledge to \mathcal{I} .

Hereafter, \mathcal{I} leaks the submitted transaction (with *entity* as initiator) to \mathcal{S} . The leakage includes whether \mathcal{I} accepted the transaction and triggers \mathcal{S} .

1. In the case that \mathcal{I} rejects the transaction: The submitted transaction is rejected during the simulation of \mathcal{R}' as well due to the replacement of `executeValidation` by the validation result from \mathcal{I} . Thus, \mathcal{R}' and \mathcal{I} stay synchronized in this case. Observe that the entity in \mathcal{R}' may now push several transactions to other entities. However, the information is not processed for other entities in \mathcal{R}' so far - thus their knowledge does not change and \mathcal{R}' and \mathcal{I} are still synchronized.
2. In the case that \mathcal{I} accepts the transaction. Observe that the entity's potential knowledge in \mathcal{I} does not change at this point. The transaction however is added to $txIDs_{buffered}^{\mathcal{I}}$. In the simulation of \mathcal{R}' the transaction submission is accepted as well and added to $txIDs_{buffered}^{\mathcal{R}}$:

As stated above, real knowledge and active knowledge are synchronous before the transaction is validated in \mathcal{I} . Further, \mathcal{R}' and \mathcal{I} execute the same checks on the same active knowledge (and \mathcal{S} has access to the necessary data for the checks as this is included in \mathcal{I} leakage). Only the check `executeValidation` in \mathcal{R} is replaced by `true` (as \mathcal{I} accepted). This concludes that \mathcal{R}' and \mathcal{I} are still synchronized.

Observe that the entity starts the `CollectSignaturesFlow` in \mathcal{R}' and starts to distribute the transaction to the signees. As already explained, at this point, no further knowledge in \mathcal{R}' changes as the information still waits for delivery. Thus, \mathcal{R}' and \mathcal{I} are synchronized after the simulation of \mathcal{R}' .

Attachment submission: The leakage of \mathcal{I} indicates whether a submitted attachment is valid. In case it is valid, however, it is not directly added to $attachmentsIDs^{\mathcal{I}}(entity)$, i. e., it is not part of the entity's knowledge so far. As \mathcal{R}' uses the leakage from \mathcal{I} instead of checking the attachment to simulate validity checks for the blinded attachment, \mathcal{R}' will accept the attachment if \mathcal{I} accepts it. If the attachment leaks, \mathcal{R}' uses the same (deterministic) algorithm to validate the attachment as \mathcal{I} and will also accept the attachment if \mathcal{I} accepts it. \mathcal{R}' adds the attachment immediately to the entity's real knowledge $attachmentsIDs^{\mathcal{R}}(entity)$ after the activation. In this case, \mathcal{S} moves the attachment immediately via an `Update` into the submitting parties knowledge $attachmentsIDs^{\mathcal{I}}(entity)$ at \mathcal{I} . Thus, \mathcal{R}' and \mathcal{I} are synchronized after the simulation of \mathcal{R}' .

Transaction push operation: Before the leakage of \mathcal{I} 's indicates that $entity_a$ pushed a transaction with ID $txID$ to $entity_b$, it queries \mathcal{S} for the current state of $entity_a$. Due to the same reasoning as above, \mathcal{I} accepts \mathcal{S} 's proposed current state update for $entity_a$. Thus, \mathcal{R} and \mathcal{I} are still synchronized and the real knowledge in \mathcal{R}' is equal to the active knowledge in \mathcal{I} .

Observe that \mathcal{I} 's leakage indicates whether the push operation was accepted. In any case, the pushed transaction is not directly added to the entity's potential knowledge $txIDs^{\mathcal{I}}(entity)$.

After \mathcal{I} leaks whether it accepted the transaction push operation of $txID$ from $entity_a$ to $entity_b$, the simulation \mathcal{R}' outputs the same result as the same logic and the same algorithms are running with the same input (as real knowledge and active knowledge

²⁰ \mathcal{S} can determine the role of the submitting entity as \mathcal{I} 's leakage includes the roles of the involved parties.

are in sync and \mathcal{R}' uses whether \mathcal{I} accepted the push operation). In the case that both reject the operation, we have nothing to show. If both accept the operation, $txID$ is added immediately to the entity's real knowledge. In this case, \mathcal{S} immediately pushes the transaction (ID) via the command `Update[txExchange]` to the entity's potential knowledge. Observe, that this operation adds all transactions from $msglist$ which are in the transaction subgraph below the transaction $txId$ and all attachments mentioned in this subgraph to the entity's potential knowledge. Thus, \mathcal{R}' and \mathcal{I} are synchronized after this operation.

Simulation of internal Corda protocol steps and $\mathcal{F}_{unicast}$: Before arguing that read operations are indistinguishable between \mathcal{R} and \mathcal{I} , we prove that \mathcal{E} cannot use \mathcal{S} 's simulation of \mathcal{P}^c in connection with the simulated network operations to distinguish between \mathcal{R} and \mathcal{I} . By construction of \mathcal{R} and \mathcal{I} , the simulation of $\mathcal{F}_{unicast}$ does not directly result in an input to the environment via I/O. However, the network simulation might influence future read requests and produce output to the network. Thus, the main goal here is to prove that \mathcal{R}' and \mathcal{I} are still synchronized after the activation of $\mathcal{F}_{unicast}$ and \mathcal{S} can generate the "expected" output to the network, i. e., the behavior on the network during this part of the simulation is indistinguishable.

Note that there is no interaction in \mathcal{I} as long as \mathcal{S} does not explicitly involve \mathcal{I} . Thus, the potential knowledge in \mathcal{I} stays constant until \mathcal{S} triggers \mathcal{I} .

Note that $\mathcal{F}_{unicast}$ does not allow sending messages on behalf of other entities, i. e., $\mathcal{F}_{unicast}$ ensures an authenticated secure channel for communication.

Note that the leakage to the network generated by simulation of $\mathcal{F}_{unicast}$ is indistinguishable between \mathcal{R} and \mathcal{I} as \mathcal{S} 's dummy transactions have the same length as the original transaction and all other leaked data is available to \mathcal{S} in the simulation of \mathcal{R}' .

Further note that honest entities in \mathcal{R}' mostly process transactions/messages which are dedicated from them, i. e., they do not reply to invocations where they do not have the matching state or are not involved as initiator, signee, or notary in the transaction. If a process step has error handling, we will discuss this in the following. Otherwise, there is no error handling and the entity simply declines further processing of a transaction/message.

Observe that only the `SignTransactionFlow` operation is a result of a submit request. It triggers the simulation of the Corda flow processing in \mathcal{R}' , \mathcal{S} waits for the `Deliver` command on the network. The command determines, which $\mathcal{F}_{unicast}$ message (with a unique message ID) needs to be delivered in \mathcal{R}' . Let $entity_s$ be the sender and $entity_r$ be the recipient of the message.

`SignTransactionFlow` Messages: We can distinguish the following cases:

1. The delivered message in the simulation was sent by an uncorrupted entity $entity_s$ (which was the initiator of the transaction):

In this case, there exists a valid submit transaction from the initiator in $txIDs_{buffered}^{\mathcal{R}}(entity_s)/txIDs_{buffered}^{\mathcal{I}}(entity_s)$. If there exists a message from $entity_r$ in $txIDs_{buffered}^{\mathcal{R}}(entity_r)/txIDs_{buffered}^{\mathcal{I}}(entity_r)$ indicating that $entity_s$ approves the message (and $entity_s$ is a signee of the transaction), the simulation will continue by validating the transaction. Otherwise, the transaction is directly declined.

In the case that the transaction proposal is not declined: observe that \mathcal{S} actually can simulate the steps of $entity_r$: the blinded transaction available in \mathcal{R}' contains all necessary information for the checks and execution of `executeValidation` is replaced by `true` if the transaction already passed the validation by $entity_s$ (otherwise `false`). In every case, the set $txIDs_{buffered}^{\mathcal{R}}(entity_s)$ does not change as either the transaction was already in the set or is declined and does not enter the set. In the case of acceptance, there are two sub-cases to consider:

- a) a direct acceptance of transaction in the simulation of $entity_r$. In this case, the simulation in \mathcal{R}' signs the dummy transaction according to the protocol and generates a `Signature` message that is sent back to $entity_s$.
- b) due to missing dependencies, $entity_r$ may request further dependencies from $entity_s$ via $\mathcal{F}_{unicast}$, more specifically via `SendTransactionFlow` or `GetAttachment` requests.

In the case that $entity_r$ did not receive approval of the transaction in advance, i. e., there is no matching transaction for `SignTransactionFlow` request in $txIDs_{buffered}^{\mathcal{R}}(entity_r)$, $entity_r$ declines the request (again, \mathcal{S} has access to all information that are necessary to execute the simulation).

2. The delivered message in the simulation was sent by a corrupted $entity_s$ (such that $entity_s$ is the initiator of the transaction because otherwise $entity_r$ directly declines the request). At this point, there is only one difference in the behavior of $entity_r$ compared to the explained above. The execution of the validation in the simulation is done on transactions in plain – so \mathcal{S} executes `executeValidation` in the simulation. This is possible as the corrupted party needs to provide the transactions in plain to $entity_s$ in the execution of \mathcal{R}' . If the corrupted entity references transactions such that \mathcal{S} is not aware of the transaction content, \mathcal{S} can query \mathcal{I} for the output of `executeValidation` in the context of $entity_r$'s state.²¹

Observe that in every case, we still have that \mathcal{R}' and \mathcal{I} are synchronized as the knowledge of the parties does not change.

`Signature` Messages: We can distinguish the following cases:

1. In the case that $entity_r$ receives the message from an uncorrupted entity $entity_s$, $entity_r$ verifies that it is waiting for the signature of $entity_s$ for the mentioned transaction (note that \mathcal{S} has access to this information due to leaked data).

²¹Note that the call at \mathcal{I} will not fail during \mathcal{I} validation, as the entity's current state only increases and $entity_r$'s state is always a subset of the entity's potential knowledge.

- a) If the message from $entity_s$ indicates (due to the signature check) that $entity_r$ accepts the message: (i) $entity_r$ may wait for further signatures or (ii) $entity_s$ collected all expected signatures and starts the `FinalityFlow`. Again, S has access to all necessary data to perform the simulation, `executeValidation` is replaced by the validity information leaked by I .
 - b) If the message from $entity_s$ indicates that it declines the transaction, $entity_r$ removes the transaction from her internal buffer $txIDs_{buffered}^{\mathcal{R}}(entity_r)$.
2. In the case that $entity_r$ receives the message from a corrupted entity $entity_s$, simulation works in the same way as explained above. Analogously, to the explanations in `SignTransactionFlow`: As $entity_s$ is the initiator of the transaction, all data is available in plain for simulation.

Observe that in every case, we still have that \mathcal{R}' and \mathcal{I} are synchronized as the knowledge of the parties does not change.

`Notarise` : Processing of `Notarise` is analogously to the processing of `SignTransactionFlow` with additional checks regarding the signatures of signees.

1. In the case that $entity_r$ (a notary in \mathcal{R}') receives the message from an uncorrupted entity $entity_s$ ($entity_s$ is the initiator of the transaction), it processes the messages as explained above (`executeValidation` is replaced by leakage from \mathcal{I} , S has access to all necessary data as explained above). The outcome of the simulation may trigger (i) `SendTransactionFlow` or `GetAttachment`, requests to $entity_s$ (via $\mathcal{F}_{unicast}$) (ii) or $entity_r$ notarizes the transaction and `NotariseRes` message to $entity_s$ (if all checks and signatures in the simulation verify).

In the latter case, the transaction is added to $entity_r$ real knowledge. According to the specification of S for this case, she triggers an update at \mathcal{I} that moves the notarised transaction entries from $txIDs_{buffered}^{\mathcal{I}}$ to $txIDs^{\mathcal{I}}$. Observe that \mathcal{I} will accept the update as S does not violate on of \mathcal{I} checks:

- S immediately pushes attachments or transaction push messages to \mathcal{I} 's msglist.
- transactions sent to notarization from honest initiators in \mathcal{R}' are checked with a subset of the rules from the update process in \mathcal{I} , such as format checks, role checks, and double-spending prevention.
- If all honest parties agree on a transaction, this is already known in \mathcal{I} due to the approval message via I/O and because the initiator of the transaction ($entity_s$) has access to all necessary information to validate, process, and distribute the information.
- \mathcal{I} does not require approval for a transaction from corrupted parties.

As the update in \mathcal{I} is not rejected, the potential knowledge of all entities involved in the notarized transaction increases: In \mathcal{I} , all entities involved in the transaction add to the transaction itself, the transaction subgraph below the transaction, and all used attachments in the subgraph to their potential knowledge.

2. In the case that $entity_r$ (a notary in \mathcal{R}') receives the message from a corrupted entity $entity_s$, S has full access to all transactions necessary to execute the notarization (or he may request the output of `executeValidation` at \mathcal{I}). Thus, the simulation works as in the case above. Note that corrupted entities cannot forge the signatures of other entities. Thus, we can follow the explanation from above. S will propose the notarised transaction to \mathcal{I} . \mathcal{I} will accept it according to the reasoning above.

Observe that in every case, we still have that \mathcal{R}' and \mathcal{I} are synchronized as the increase of knowledge in \mathcal{R}' is always a subset of the knowledge in \mathcal{I} .

`NotariseRes` : We can distinguish the following cases:

1. In the case that $entity_r$ (the initiator of the notarized transaction in \mathcal{R}') receives the message from an uncorrupted entity $entity_s$ (the notary): The simulation of $entity_r$ process the notarization message. In the case that the simulation accepts the notarization, the `FinalityFlow` is triggered. Note that S can simulate this operation as the execution of `executeValidation` is replaced as above. Observe that the transaction is added to real knowledge. As the transaction was already in its potential knowledge (see previous steps of honest parties), \mathcal{R}' and \mathcal{I} are still synchronized.
2. In the case that $entity_r$ receives the message from a corrupted entity $entity_s$ (and $entity_r$ is involved in the transaction notarized), then $entity_r$ is corrupted as well. Thus, S has full access to all transaction details necessary to perfectly simulate this situation.

Observe that in every case, we still have that \mathcal{R}' and \mathcal{I} are synchronized.

`RecvFinalityFlow` : We can distinguish the following cases:

1. In the case that $entity_r$ receives the message from an uncorrupted entity $entity_s$ (the transaction initiator), there is all information available to simulate this process (based on the leakage of \mathcal{I}). If the entity $entity_r$ accepts the `RecvFinalityFlow` message, it adds the transaction to the entity's real knowledge. As the transaction was already in its potential knowledge (see previous steps of honest parties), \mathcal{R}' and \mathcal{I} are still synchronized.
2. In the case that $entity_r$ receives the message from a corrupted entity $entity_s$ ²² (and $entity_r$ is involved in the transaction): Note that S has sufficient information to simulate this step. In the case that $entity_r$ accepts the incoming message: As $entity_r$ is not corrupted, we can conclude that the notary of the transaction in \mathcal{R}' is not corrupted. Thus, S already pushed the transaction

²²Note that we do not discuss the case when $entity_r$ is corrupted as it matches the case above.

to \mathcal{I} before this step which includes that the transaction is already part of $entity_r$ potential knowledge. In the simulation, the transaction is added to $entity_r$ real knowledge in this step. In the case that $entity_r$ declines the `RecvFinalityFlow` message, we have nothing to show.

Observe that in every case, we still have that \mathcal{R}' and \mathcal{I} are synchronized.

`SendTransactionFlow` and `RecvTransactionFlow`: According to the explanations above, the simulator \mathcal{S} can simulate this in any case. In the case that the process is executed between honest entities, we can conclude similarly to above that \mathcal{R}' and \mathcal{I} stay synchronized as the submission of the transaction added it already to $entity_r$ potential knowledge. However, as all transactions leaked to \mathcal{A} are (potentially) known by honest parties (see also the definition of the different known sets), we can conclude that if a corrupted party sends a transaction (graph) and the entity accepts it, the knowledge of the entity does not change. Thus, \mathcal{R}' and \mathcal{I} still stay synchronized (as the entity is not in the set of honest entities any longer).

`GetAttachment` and `RecvAttachment`: According to the explanations above, \mathcal{S} can simulate this in any case. We emphasize here, that \mathcal{S} can call `executeValidation` at \mathcal{I} in the context of the receiving party's current state²³ to validate the transaction in the context of the receiving party. In the case that honest entities execute this process, we can conclude – similarly to above – that \mathcal{R}' and \mathcal{I} stay synchronized as already a submission added the transaction and it to $entity_r$ potential knowledge. Observe that corrupted entities may push attachment to $entity_r$, although they were not requested. As $entity_r$ declines such push message, we can conclude that \mathcal{R}' and \mathcal{I} are synchronized.

Read requests: Whenever an honest entity $entity$ receives a request (`Read, msg`) to read from its restricted view on the global state, \mathcal{I} , forwards this (local) read request to \mathcal{S} and waits to receive a suggested output. \mathcal{S} simulates the read request internally (as input to an instance of \mathcal{P}_{client}^c). \mathcal{S} extracts the transaction and attachment IDs from the output and forwards them in a `InitRead` message to \mathcal{I} . In the next step, \mathcal{I} requests the current knowledge of $entity$ from \mathcal{S} . \mathcal{S} extracts the current knowledge from \mathcal{R}' as explained above. As already argued above: as \mathcal{R}' and \mathcal{I} are synchronized, no operation adds knowledge on one of both sides here. In particular, \mathcal{S} 's suggested output for the read request will not fail \mathcal{I} 's validations:

1. The read output of an honest party simulated in \mathcal{R}' is a monotonically increasing set that always contains the previous read output as a subset (due to the specification of Corda).
2. The read output is always a subset of the entity's current knowledge in \mathcal{R}' (due to the specification of `Corda/ \mathcal{P}_{client}^c`).
3. If the party is uncorrupted, the output in \mathcal{R}' only consists of transactions notarized by uncorrupted notaries. Thus, all transactions in the suggested read output are in `msglist` in \mathcal{I} and can be accessed.
4. As the entity in \mathcal{R}' only accepts pushed transactions (including attachments) of notarized transactions (subgraphs), potential additional transactions added to the entity's read output are already leaked (see explanation during flow explanations).
5. Observe that \mathcal{R}' , more specifically, $\mathcal{F}_{unicast}$, enforces the upper bound of δ rounds for eventual message delivery (initiator receives notarization δ rounds after notarization, signees after 2δ rounds) are not violated by \mathcal{S} .
6. Honest entities in the simulation will output transactions only if they are aware of their full subgraph (and output the subgraph as well).
7. If an honest entity shares a transaction (subgraph) with another honest entity, the operation will finish after $|subgraph(tx)|$ rounds where $subgraph(tx)$ is the subgraph of/below the exchanged transaction (such that all input references are part of the graph, down to the issuance transactions). This follows as the receiving entity queries in the worst case the full subgraph of the transaction before adding the transaction to its `verifiedTx`. In the worst case, these are $|subgraph(tx)|$ objects to query (if the graph is a chain). Thus, the push operation will be finished at least after $|subgraph(tx)| + 1$ rounds.

Observe that entities in \mathcal{R}' are considered corrupted as soon as they have transactions notarized by corrupted notaries in `verifiedTx`. Thus, we do not have to consider the case, that the entity should output transactions/attachments not in `txIDs \mathcal{I}` or `attachmentIDs \mathcal{I}` as the entity is then corrupted.

Corrupting parties: In the following, we argue that (i) \mathcal{S} is always able to keep corrupted parties in \mathcal{R}' and \mathcal{I} synchronous. As there are no restrictions for corruption in \mathcal{I} , we have nothing to show here. Observe that one corruption operation (of a notary in \mathcal{R}') may lead to several corruption operations in \mathcal{I} due to our corruption model. Further, (ii) \mathcal{S} is always able to generate the appropriate (internal) state of a freshly corrupted entity such that leakage during corruption cannot be used to distinguish between \mathcal{R} and \mathcal{I} . This follows from the specified leakage:

1. \mathcal{S} gets access to the complete potential knowledge of an entity as soon it is corrupted. \mathcal{S} replaces the dummy transactions in \mathcal{R}' with the original, plain transaction (including regeneration of signatures). In the case that there are some transactions/attachments provided from a corrupted party to the newly corrupted party, the simulation in \mathcal{R}' already contains the data in plain.
2. \mathcal{S} gets access to transactions, transaction subgraph, and the used attachments if a corrupted party is involved in the transaction (as initiator, signee, or notary). Again, \mathcal{S} includes the leaked data in \mathcal{R}' and regenerates signatures depending

²³Note that \mathcal{I} accepts the current knowledge provided by \mathcal{S} as the current knowledge in \mathcal{R} is a subset of the potential knowledge.

on this data. Thus, \mathcal{S} has access to all data in plain and can perfectly simulate the processing of the transaction including the handling of signatures.

Interaction via network: First, observe that \mathcal{I} provides \mathcal{S} sufficient information about all requests performed by higher-level protocols, such as the blinded transactions submitted to the ledger, blinded attachments, transaction exchange operation, and read requests to correctly simulate (blinded) messages on the network interface: \mathcal{S} is able (according to the explanations above) to simulate the Corda protocol blinded but indistinguishable from a real execution. In particular, \mathcal{S} has access to all necessary data in plain to provide the correct leakage or to provide data in plain in case corrupted parties are involved in a transaction. Further, \mathcal{S} can call \mathcal{I} to validate transactions in the context of an entity's state. As a result, the network behavior simulated by \mathcal{S} towards the environment is indistinguishable from the network behavior of \mathcal{R} . As already argued above, it also follows that the corruption statuses of entities in the real and ideal world are always identical.

Current time requests: As the simulator updates the $\mathcal{F}_{\text{ledger}}$'s internal clock every time an update to $\mathcal{F}_{\text{unicast}}$ in \mathcal{R}' occurs, both worlds always output the same value for the current time. Observe that the \mathcal{S} 's time update requests always passes the checks in $\mathcal{F}_{\text{updRnd}}^{\mathbf{c}}$:

$\mathcal{F}_{\text{unicast}}$ enforces that all messages are delivered in at most δ rounds in \mathcal{R} . $\mathcal{F}_{\text{updRnd}}^{\mathbf{c}}$ guarantees that a transaction tx in *requestQueue*, where all involved parties are honest, are part of the msglist after at most $(3 + 4 \cdot |\text{subgraph}(tx)| \cdot \delta)$ rounds (where *subgraph*(tx) is the transaction subgraph below tx as explained above). In the worst-case scenario, the simulated notary and at least one signee do not know any dependencies of tx and the subgraph is a chain, we conclude that the bound from $\mathcal{F}_{\text{updRnd}}^{\mathbf{c}}$ as follows: We assume w.l.o.g. that the initiator was the last involved party that submitted the transaction to *requestQueue* (otherwise if signees do not add their approval to *requestQueue* before the initiators *CollectSignaturesFlow* reaches them, they will decline the transaction. In this case, the last agreement is later than the one of the initiator but we can guarantee the bound for the initiator – so we do not violate the bound for the actual last agreement of the transaction.)

1. It takes at most δ time units/ rounds to deliver data via the *CollectSignaturesFlow* messages from initiators to signees.
2. signees may request at most $|\text{subgraph}(tx)|$ dependencies from the initiator. Thus, delivery/processing needs at most $2 \cdot |\text{subgraph}(tx)| \cdot \delta$ rounds.
3. It takes at most δ rounds for the last signee to deliver her approval messages to the initiator.
4. The initiator needs at most δ rounds to deliver the notarization request to a notary
5. The notary may request at most $|\text{subgraph}(tx)|$ dependencies from the initiator. Thus, delivery/processing needs at most $2 \cdot |\text{subgraph}(tx)| \cdot \delta$ rounds. Then, \mathcal{S} triggers the *Update* to \mathcal{I} in the case that the notarization succeeded.

Overall, we conclude that \mathcal{S} triggers the update in this case after at least $(3 + 4 \cdot |\text{subgraph}(tx)| \cdot \delta)$ rounds after the last agreement was recorded. Thus, \mathcal{I} will accept the time updates of \mathcal{S} .

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

Description of parameters/algorithms used in the protocols $\mathcal{P}_{\text{client}}^c$ and $\mathcal{P}_{\text{notary}}^c$:

`executeValidation(tx, txDependencies, txAttachments)`

Among other things, `executeValidation` should be selected such that it

- interprets attachments and checks whether a transaction is valid w.r.t. these attachments,
- ensures that attachments are deterministic and enforces that the overall runtime of all attachments (and further validations) is in polynomial time
- checks whether the transaction is contractually valid, i. e., , fulfills the requirements given by the contracts for each state and the transaction. `executeValidation` does not check signature validity.
- checks whether all inputs have the same notary.
- checks whether all transaction outputs have the same notary as the consumed inputs (if the transaction is not a notary change transaction).
- checks that the owner of the transaction's *inputs* are a subset of the participant's transactions.
- checks that the owner of the transaction's *output* states are a subset of the participant's transactions.
- checks the correct format of the transactions, i. e., , whether they are correct common transactions, reference states/transactions, and notary change transactions, including checking correct roles (as available in $\mathcal{P}_{\text{client}}^c$ and $\mathcal{P}_{\text{notary}}^c$). In this case, `executeValidation` may also be called with just a single input `tx`

`validateAttachment(attachment)` This algorithm checks whether a single attachment is valid according to certain system-defined rules.

`isResolvable(tx)` This algorithm checks whether all dependencies and attachments for validating a transaction, i. e., , needed by `executeValidation`, are present. The dependencies need to be in the internal state `verifiedTx` or `bufferTxSig`.

`isValidId(pid, role)` This algorithm checks whether the given `pid` belongs has the `role` according to a predefined networkmap.

`isDoubleSpend(tx)` This algorithm checks whether the given `tx` is a double-spend compared to transactions in the internal state `verifiedTx`. That is, if there exists a transaction in `verifiedTx` that is not the same transaction (without a notary signature) but uses one of the input states of `tx` as well, `isDoubleSpend` recognizes a double-spending attempt. Note that the re-usage of a reference transaction is not considered as double-spending.

Figure 23: Algorithms and parameters used by $\mathcal{P}_{\text{client}}^c$ and $\mathcal{P}_{\text{notary}}^c$

Description of the protocol $\mathcal{P}_{\text{client}}^{\mathbf{c}} = (\text{client})$:

Participating roles: {client}	
Corruption model: <i>dynamic corruption without secure erasures</i>	
Protocol parameters:	
- executeValidation	{Algorithm for verifying transactions}
- networkmap	{Set of identities in the Corda instance, entries of form (type, id)}
- validateAttachment	{Algorithm for validating attachments}

Description of M_{client} :

Implemented role(s): {client}	
Subroutines: $\mathcal{F}_{\text{cert}}$: cert, $\mathcal{F}_{\text{unicast}}$: unicast, \mathcal{F}_{ro} : randomOracle, $\mathcal{P}_{\text{notary}}^{\mathbf{c}}$: notary,	
Internal state:	
- verifiedTx $\subset \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^*$, verifiedTx = \emptyset	{Verified transactions, of form (txId, tx, Σ), Σ is a set containing tuples of the form (pid, σ)}
- attachments $\subset \{0, 1\}^* \times \{0, 1\}^*$, attachments = \emptyset	{Set of known attachments and identifiers as (attachmentId, attachment)}
- bufferTxSig $\subset \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \times \text{txStatus}^{\mathbf{c}, a}$ bufferTxSig = \emptyset	{Transaction currently in process as (txId, tx, Σ , pid, status). Σ is a set of tuples of the form (pid, σ), status $\in \text{txStatus}^{\mathbf{c}}$ }
- bufferReqValid $\subset \{0, 1\}^* \times \{0, 1\}^*$, bufferReqValid = \emptyset	{Set of transactions and attachments allowed to be queried as (txId/attachmentId, allowedRequesters). bufferReqValid is an ordered set}
CheckID (pid, sid, role): Accept all messages addressed to the same entity (pid, sid, role). Only accept role = client.	
Corruption behavior:	
- DetermineCorrStatus (pid, sid, role):	
if corr = true:	{Checks whether client itself is corrupted}
return true	
corrRes \leftarrow corr(pid _{cur} , sid _{cur} , signer)	
if corrRes = true:	{Checks whether $\mathcal{F}_{\text{cert}}$ instance is corrupted}
return true	
for all ($_$, tx, $_$) \in verifiedTx \cup ($_$, tx, $_$, reqNotarization) \in bufferTxSig \cup ($_$, tx, $_$, expNotarization) \in bufferTxSig do:	
tx \leftarrow ($_$, $_$, notary, formerNotary, $_$)	
corrRes ₁ \leftarrow corr(formerNotary, sid _{cur} , signer)	{Checks if at least one of all notaries used in a tx either for transactions in verifiedTx or in bufferTxSig is corrupted}
corrRes ₂ \leftarrow corr(notary, sid _{cur} , signer)	
if corrRes ₁ \vee corrRes ₂ :	
return true	
We expect the ITM to enforce the transaction format $tx = (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal})$ where in <i>proposal</i> input states are of form $tx_{\text{inputStates}} = (\text{txId}, \text{outputIndex})$ and output states are of form $tx_{\text{outputStates}}$. The symbol \oplus emphasizes that we use this format specification.	
Main:	
recv (Read, msg) from I/O:	{Reading from Corda's transaction graph}
shortendVerifiedTx \leftarrow \emptyset	
for all (txId, tx, σ) \in verifiedTx do:	
shortenedVerifiedTx.add((txId, tx))	
reply (Read, attachments, shortendVerifiedTx)	{Output the transaction graph of pid excluding signatures}
^a For clients, we set $\text{txStatus}^{\mathbf{c}} = \{\perp, \text{unrequested}, \text{requested}, \text{approved}, \text{expColSigs}, \text{reqSigs}, \text{reqDeps}, \text{sign}, \text{waitSign}, \text{expNotarization}, \text{reqNotarization}\}$. The set $\text{txStatus}^{\mathbf{c}}$ includes the different statuses a transaction passes during processing	

Figure 24: The Corda client $\mathcal{P}_{\text{client}}^{\mathbf{c}}$ (Pt. 1)

Main:

```

recv (Submit, msg) from I/O s.t. msg = (attachment, attachment): {Receive a new attachment
  send attachment to (pidcur, sidcur,  $\mathcal{F}_{\text{To}}$  : randomOracle) {Generate attachment ID
  wait for attachmentId'
  if bufferReqValid.contains(ATT.add(attachmentId'))a  $\wedge$  validateAttachment(attachment) = true:
    attachments.add(ATT.add(attachmentId'), attachment) {Add (labeled) attachment and attachment ID to attachments
  recv (Submit, msg) from I/O s.t. msg = (tx, tx)  $\wedge$  executeValidation(tx) = true  $\oplus$ : {Receive a new tx
  send tx to (pidcur, sidcur,  $\mathcal{F}_{\text{To}}$  : randomOracle)
  wait for (txId') {Generate tx ID
  txId  $\leftarrow$  TX.add(txId') {TX.add(txId') concatenates the string TX with txId', allows to identify the ID as a transaction ID
  if initiator = pidcur: {initiator starts Corda's flow processing
    if  $\forall$  (txId'', outputIndex'')  $\in$  txinputStates, s.t.
      verifiedTx.contains(txId'', tx'', _) , s.t. tx''outputStates are the output states of tx''  $\wedge$ 
      outputIndex''  $\in$  tx''outputStates: {All dependencies are known
      bufferTXSig.add((txId, tx, 0,  $\epsilon$ , expColSigs)) {Flag tx as "wait for signatures"
      if validate(tx, 0) = valid: {For validate definition see below
        for all (txId', outputIndex')  $\in$  txinputStates do:
          bufferReqValid[txId'][1].add({signee1, ..., signeem, formerNotary})
          {Record that parties are allowed to query for full dependencies of tx
        for all attachmentId  $\in$  proposal do:
          bufferReqValid[attachmentId][1].add({signee1, ..., signeem, formerNotary})
          {Record that parties are allowed to query for full dependencies of tx
        msg  $\leftarrow$  CollectSignaturesFlow(tx) {See below for CollectSignaturesFlow definition
        send (Message, msg) to (pidcur, sidcur,  $\mathcal{F}_{\text{unicast}}$  : unicast)
      else:
        bufferTXSig.remove((_, tx, _, _)) {Clean up declined transaction
        validateAllLocal() {Check whether there are open task, e.g., finalizing a transaction
      else if  $\exists i \in [m] : \text{signee}_i = \text{pid}_{\text{cur}}$ : {As signee: record tx as "will approve"
        bufferTXSig.add((txId, tx, 0,  $\epsilon$ , approved))
  recv (Submit, msg) from I/O s.t. msg = (tx, txId, pidrecv)  $\oplus$ : {pidcur pushes/shares a verified transaction to/with pidrecv
  msg'  $\leftarrow$   $\epsilon$ 
  if verifiedTx.contains((txId, _, _)):
    msg'  $\leftarrow$  (verifiedTx[txId][1], verifiedTx[txId].[2]) {msg' = (tx,  $\Sigma$ )
  else if bufferTXSig.contains((txId, _, _, status)) | status  $\notin$  { $\perp$ , unrequested, requested, reqDeps}:
    msg'  $\leftarrow$  (bufferTXSig[txId][1], bufferTXSig[txId][2]) {msg' = (tx,  $\Sigma$ )
  if msg'  $\neq$   $\epsilon$ :
    send (Message, {(pidrecv, (RecvTransactionFlow, msg'))}) to (pidcur, sidcur,  $\mathcal{F}_{\text{unicast}}$  : unicast)
  recv GetCurRound from I/O or NET: { $\mathcal{A}$  and  $\mathcal{E}$  are allowed to query the clock
  send GetCurRound to (pidcur, sidcur,  $\mathcal{F}_{\text{unicast}}$  : unicast) {Forward requests to  $\mathcal{F}_{\text{unicast}}$ 
  wait for (GetCurRound, round)
  reply (GetCurRound, round)

```

^aThe function contains applied to a string, a set, or tuple outputs true if the substring or element is part of the object being searched.**Figure 25: The Corda client $\mathcal{P}_{\text{client}}^{\text{c}}$ (Pt. 2)**

Main:

```

recv (SignTransactionFlow, (pids, msg)) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast) s.t. msg = (tx,  $\sigma$ )  $\oplus$ :
    {On request of a transaction initiator who started a CollectSignaturesFlow, a client
     signs a transaction if it agrees to the tx}
if bufferTXSig[tx].contains( $\_$ , tx, 0,  $\epsilon$ , approved)  $\wedge$  initiator = pids:
    send (Verify, tx,  $\sigma$ ) to (initiator, sidcur,  $\mathcal{F}_{cert}$  : verifier)
    wait for (VerResult, res) {Check initiators signature}
    if res = valid:
        bufferTXSig[tx][3]  $\leftarrow$  initiator {Transaction tx is already stored in bufferTXSig. It is now marked as re-
        requested for signature and the requestor initiator is stored for possibly later
        answer}
        bufferTXSig[tx][4]  $\leftarrow$  sign {Update processing status}
        (txId, tx,  $\Sigma$ , pid, sign)  $\leftarrow$  bufferTXSig[tx]
        if validate(tx,  $\Sigma$ ) = valid: {For definition of validate, see Figure 27}
            send (Sign, tx) to (pidcur, sidcur,  $\mathcal{F}_{cert}$  : signer)
            wait for (Signature,  $\sigma$ ) {Generate signature for initiator}
            bufferTXSig[tx][4]  $\leftarrow$  expNotarization {Update processing status}
            reply (Message, ((initiator, (Signature, (tx,  $\sigma$ ))))))
    else:
        reply (Message, ((initiator, (Signature, (tx,  $\perp$ ))))))

recv (Signature, (pids, msg)) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast) s.t. msg = (tx,  $\sigma$ )  $\oplus$ :
    {Collect signed responses from the CollectSignaturesFlow}
if bufferTXSig.contains( $\_$ , tx,  $\_$ , reqSigs)  $\wedge$ 
    pids  $\in$  signee[m]  $\wedge$  (pids,  $\_$ )  $\notin$  bufferTXSig[tx][2]:
    {Accept only requested signatures, i. e., those
     transactions in bufferTXSig where pidcur is ini-
     tiator}
    send (Verify, tx,  $\sigma$ ) to (pids, sidcur,  $\mathcal{F}_{cert}$  : verifier)
    wait for (VerResult, res) {Check signee signature}
    if res = true:
        bufferTXSig[txId][2].add(( $\sigma$ , pids))
        if |bufferTXSig[txId][2]| = m + 1:
            {Every signature is only stored once if valid. Thus, if all signees and the
             initiator signed, proceed with notarization}
            if validate(tx,  $\Sigma$ ) = valid:
                {Start FinalityFlow if all signees confirmed tx}
                bufferTXSig[tx][4]  $\leftarrow$  reqNotarization {Mark tx as waiting for notarization}
                msg'  $\leftarrow$  (Notarise, (tx,  $\Sigma$ ))
                send (Message, (formerNotary, msg')) to (pidcur, sidcur,  $\mathcal{F}_{unicast}$  : unicast) {Ask for nota-
                rization}
            else:
                bufferTXSig.remove((txId, tx,  $\_$ ,  $\_$ )) {Remove rejected tx from buffer}

recv (NotariseRes, (pids, msg)) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast) s.t. msg = (tx,  $\sigma$ ):
    {Handle tx finalization from the notary and distribute result among signees}
if bufferTXSig[tx].contains( $\_$ , tx,  $\_$ , formerNotary, reqNotarization), s.t. pids = formerNotary:
    {Only expected signatures from a notary are distributed}
    bufferTXSig[tx][2].add(formerNotary,  $\sigma$ )
    res  $\leftarrow$  validate(tx, bufferTXSig[tx][2]) {Will typically return valid as an honest client triggers this for valid
    tx. See below for the definition of validate}
    if res = valid  $\wedge$  | $\Sigma$ | = m + 2:
        verifiedTx.add(bufferTXSig[tx][0], bufferTXSig[tx][1], bufferTXSig[tx][2])
        bufferTXSig.remove(bufferTXSig[tx])
        for all i  $\in$  [m] do:
            msg.add((signeei, (RecvFinalityFlow, (tx,  $\Sigma$ .add(formerNotary,  $\sigma$ ))))))
        reply (Message, msg) {Distribute notariza-
        tion to signees}

recv (RecvFinalityFlow, (pids, msg)) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast) s.t. msg = (tx,  $\Sigma$ ):
    {Called from initiator pids after FinalityFlow is finished. Stores tx with signing result from notary}
if bufferTXSig[tx].contains( $\_$ , tx,  $\_$ , pids, expNotarization):
    if | $\Sigma$ | = m + 2  $\wedge$  validate(tx,  $\Sigma$ ) = valid: {Check validity and number of signatures}
        verifiedTx.add((txId, tx,  $\Sigma$ ))
        bufferTXSig.remove(bufferTXSig[txId])

```

Figure 26: The Corda client \mathcal{P}_{client}^c (Pt. 3)

Description of M_{Client} (cont.):

Main:

```

recv (SendTransactionFlow, (pids, msg)) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast) s.t. msg = (txId):
    {Client processes/forwards a requested transaction}
    if bufferReqValid[txId][1].contains(pids):
        bufferReqValid[txId][1].remove(pids)
        {Checks whether pid is allowed to request transaction tx}
        if verifiedTx.contains(txId,  $\_$ ): tx  $\leftarrow$  verifiedTx[txId][1]
        {Decline further access}
        else: tx  $\leftarrow$  bufferTxSig[txId][1]
        {verifiedTx[.][1] = tx}
        {bufferTxSig[.][1] = tx}
        for all (txId', outputIndex')  $\in$  txinputStates do:
            {Grant pids access to dependencies of tx}
            bufferReqValid[txId'].add(pids)
        for all attachmentId'  $\in$  tx[4] do:
            bufferReqValid[attachmentId'].add(pids)
        msg' = (RecvTransactionFlow, (verifiedTx[txId][1], verifiedTx[txId][2]))
        {Forward tx details to pids}
        send (Message, (pids, msg')) to (pidcur, sidcur,  $\mathcal{F}_{unicast}$  : unicast)

recv (GetAttachment, (pids, msg)) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast) s.t. msg = attachmentId:
    {Answers attachment request}

    if bufferReqValid[attachmentId].contains(pids):
        msg' = (RecvAttachment, attachments[attachmentId])
        reply (Message, (pids, msg'))

recv (RecvTransactionFlow, (pids, (tx,  $\Sigma$ ))) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast):
    {Entity processes a requested transaction}
    require: executeValidation(tx) = true
    send tx to (pidcur, sidcur,  $\mathcal{F}_{To}$  : randomOracle)
    {Get transaction ID}
    wait for (txId')
    txId  $\leftarrow$  TX.add(txId')
    if ( $\neg$ (bufferTxSig.contains((txId,  $\_$ , pids, s)), s  $\notin$  { $\perp$ , unrequested})  $\wedge$ 
        verifiedTx[txId].contains((txId,  $\_$ ,  $\_$ ))):
        bufferTxSig.add((txId, tx,  $\Sigma$ , pids, unrequested))
        {pidcur did not ask for this tx}
    else if bufferTxSig.contains((txId,  $\perp$ ,  $\perp$ , pids, requested)):
        res  $\leftarrow$  validate(tx,  $\Sigma$ , pids)
        {See definition of validate below}
        if res = valid  $\wedge$  | $\Sigma$ | = m + 2:
            verifiedTx.add(bufferTxSig[tx][0], bufferTxSig[tx][1], bufferTxSig[tx][2])
            bufferTxSig.remove(bufferTxSig[tx])
            validateAllLocal()

recv (RecvAttachment, (pids, attachment)) from ( $\_$ , sidcur,  $\mathcal{F}_{unicast}$  : unicast):
    {Stores received attachments in attachments.}
    send (attachment) to (pidcur, sidcur,  $\mathcal{F}_{To}$  : randomOracle)
    {Get attachment ID}
    wait for (attachmentId')
    if validateAttachment(attachment):
        attachments.add(ATT.add(attachmentId', attachment))
        {Store attachment}
        validateAllLocal()

recv DeRegister from I/O:
    {Deregistration is not part of the Corda model}
    reply DeRegister

Procedures and Functions:

function CollectSignaturesFlow(tx) (tx = ( $\_$ , [signee1, ..., signeem],  $\_$ ,  $\_$ )):
    {Collects signatures for a transaction from all signees}
    bufferTxSig[tx][4]  $\leftarrow$  reqSigs
    {Transaction is marked in bufferTxSig as expColSigs, need to query}
    {signees for signatures}
    send (Sign, tx) to (pidcur, sidcur,  $\mathcal{F}_{cert}$  : signer)
    {Initiator signs transaction proposal}
    wait for (Signature,  $\sigma$ )
    bufferTxSig[tx][3].add(pidcur,  $\sigma$ )
    {Record initiator signature}
    msg  $\leftarrow$   $\epsilon$ 
    {Prepare message to  $\mathcal{F}_{unicast}$ }
    for all i  $\in$  [m] do:
        msg.add((signeei, (SignTransactionFlow, (tx,  $\sigma$ ))))
    return msg

```

Figure 27: The Corda client \mathcal{P}_{client}^c (Pt. 4)

Procedures and Functions:

```

function validate( $tx, \Sigma$ ):
  if verifiedTx.contains( $\_, tx, \_, \Sigma'$ )  $\wedge \Sigma' \neq \Sigma$ :
    if  $|\Sigma'| = m + 2$ :
      bufferTxSig.remove(bufferTxSig[ $tx$ ]); return alreadyNotarised
    else:
      if isDoubleSpend( $tx, verifiedTx$ ):
        {Checks whether one of the input states is already used in another tx in verifiedTx}
        bufferTxSig.remove(bufferTxSig[ $tx$ ]); return doubleSpend
  if  $\neg[\forall i \in [m] : isValidId(signee_i, client) = \text{true} \wedge$ 
    isValidId( $initiator, client$ ) = true  $\wedge$ 
    isValidId( $formerNotary, notary$ ) = true  $\wedge$ 
    if isValidId( $notary, notary$ ) = true  $\wedge$ 
    isValidId( $formerNotary, notary$ ) = true]:
    {isValidId checks whether the given pid is registered with the particular role in networkmap.}
    bufferTxSig.remove(bufferTxSig[ $tx$ ]); return invalid
  if  $|\Sigma| \leq m + 1$ :
    {Before/in case of notarization,...}
    if  $\neg(\forall (pid', \_) \in \Sigma, \text{it holds true that } (pid' = initiator \vee \exists i \in [m], \text{s.t. } signee_i = pid'))$ :
      bufferTxSig.remove(bufferTxSig[ $tx$ ])
      return invalid
    {... all signatures need to be from signees or the initiator}
    else:
      {After notarization, there need to be signatures from all signees, the}
      {initiator, and the correct notary}
      if  $\neg(\text{It holds true that } \forall i \in [m], \text{there } \exists_1 (pid', \_) \in \Sigma, \text{s.t. } signee_i = pid' \wedge$ 
         $\exists_1 (pid', \_) \in \Sigma, \text{s.t. } initiator = pid' \wedge \exists_1 (pid', \_) \in \Sigma, \text{s.t. } formerNotary = pid')$ :
        bufferTxSig.remove(bufferTxSig[ $tx$ ]); return invalid
  for all  $(\sigma, pid_\sigma) \in \Sigma$  do:
    send ( $tx, \sigma$ ) to  $(pid_\sigma, sid_{cur}, \mathcal{F}_{cert} : \text{verifier})$ 
    wait for (VerResult,  $res$ )
    {Checks all provided signatures}
    if  $res = \text{false}$ :
      bufferTxSig.remove(bufferTxSig[ $tx$ ]); return invalid
  if isResolvable( $tx$ ) = false  $\wedge$  bufferTxSig[ $tx$ ][4]  $\neq$  reqDeps:
    {Checks whether all dependencies are}
    {available}
     $msg \leftarrow \text{resolveTx}(tx, initiator)$ 
    {If dependencies are missing, query the initiator}
    send (Message,  $msg$ ) to  $(pid_{cur}, sid_{cur}, \mathcal{F}_{unicast} : \text{unicast})$ 
    {Party has access to all dependencies of tx}
  else if isResolvable( $tx$ ):
    Let  $txDependencies$  be the set of all transaction bodies of (transaction) dependencies of  $tx$  from verifiedTx and
    bufferTxSig.
    {Collect dependencies for upcoming validation}
    Let  $txAttachments$  be the set of all attachments referenced/used in (transaction) dependencies of  $tx$  from verifiedTx
    and bufferTxSig.
    {Validate the transaction regarding the}
    {given context}
     $res \leftarrow \text{executeValidation}(tx, txDependencies, txAttachments)$ 
    return  $res$ 

function resolveTx( $tx, pids$ ) ( $msg$ ):
  if bufferTxSig[ $tx$ ][4] = sign: bufferTxSig[ $tx$ ][4]  $\leftarrow$  waitSign
  else: bufferTxSig[ $tx$ ][4]  $\leftarrow$  reqDeps
  {Proceed with signing after getting}
  {dependencies}
   $msg \leftarrow \epsilon$ 
  for all  $(txId', \_) \in txInputStates$  do:
    {For definition of txInputStates see }
    {Request all transactions necessary for verification but}
    {currently not in verifiedTx}
    if  $\nexists (txId', \_) \in verifiedTx$ :
      bufferTxSig.add( $(txId', \_, \_, pids, requested)$ )
       $msg.add((pids, (SendTransactionFlow, (txId'))))$ 
  for all  $attachmentId \in proposal$  do:
    {Request all attachments necessary for verification}
    {but currently not in attachments}
    if  $(attachmentId, \_) \notin attachments$ :
       $msg.add((pids, (GetAttachment, (attachmentId))))$ 
  return  $msg$ 

```

Figure 28: The Corda client $\mathcal{P}_{\text{client}}^c$ (Pt. 5)

Description of M_{Client} (cont.):

```

Procedures and Functions:
function validateAllLocal() : {Checks for all stored transactions if the inputs and attachments can  
be resolved and the transaction therefore validated}
  identities  $\leftarrow \epsilon$ , msg  $\leftarrow \epsilon$ , resolved  $\leftarrow \text{true}$ 
  while resolved = true do
    resolved  $\leftarrow \text{false}$ , valid  $\leftarrow \epsilon$ 
    for all ( $\_, tx, \Sigma, pid, status$ )  $\in$  bufferTxSig, s.t.
      status  $\notin \{\perp, \text{unrequested}, \text{expNotarization}, \text{reqNotarization}, \text{approved}, \text{reqSigs}\}$  do:
        if isResolvable( $tx$ ): {isResolvable checks whether full dependencies are available in  
verifiedTx and attachments}
          valid  $\leftarrow \text{validate}(tx, \Sigma, \epsilon)$ 
          if valid = valid  $\wedge$  status  $\in \{\text{sign}, \text{waitSign}\}$ : {Need to agree on tx}
            send (Sign,  $tx$ ) to ( $pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{F}_{\text{cert}} : \text{signer}$ )
            wait for (Signature,  $\sigma$ )
            bufferTxSig[ $tx$ ][4]  $\leftarrow \text{expNotarization}$  {Update processing status}
            msg.add( $(\text{initiator}, (\text{Signature}, (tx, \sigma)))$ ) {Send agreement/signature to initiator}
          else if valid = valid  $\wedge$  status = requested  $\wedge |\Sigma| = m + 2$ : {Finalize validation}
            verifiedTx.add(bufferTxSig[ $tx$ ][0], bufferTxSig[ $tx$ ][1], bufferTxSig[ $tx$ ][2])
            bufferTxSig.remove(bufferTxSig[ $tx$ ])
          else if valid = valid  $\wedge$  status = expColSigs:
            msg.add(CollectSignaturesFlow( $tx$ )) {Request signees to agree on the tx}
          if valid = valid  $\wedge$  status  $\in \{\text{requested}, \text{reqDeps}\}$ : {Validate/accept transactions with formerly  
missing dependencies}
            verifiedTx.add(bufferTxSig[ $tx$ ][0], bufferTxSig[ $tx$ ][1], bufferTxSig[ $tx$ ][2])
            bufferTxSig.remove(bufferTxSig[ $tx$ ])
          resolved  $\leftarrow \text{true}$ 
    send (Message, msg) to ( $pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{F}_{\text{unicast}}$ )

```

Figure 29: The Corda client $\mathcal{P}_{\text{client}}^{\text{c}}$ (Pt. 6)

Description of the protocol $\mathcal{P}_{\text{notary}}^{\mathbf{c}} = \{\text{notary}\}$:

Participating roles: {notary}	
Corruption model: <i>dynamic corruption with secure erasures</i>	
Protocol parameters:	
- executeValidation	{Algorithm for verifying a transaction.
- networkmap	{Set of identities in network
- validateAttachment	{Algorithm for validating attachments according to certain rules

Description of M_{notary} :

Implemented role(s): {notary}	
Subroutines: $\mathcal{F}_{\text{cert}} : \text{cert}$, $\mathcal{F}_{\text{unicast}} : \text{unicast}$, $\mathcal{F}_{\text{ro}} : \text{randomOracle}$	
Internal state:	
- verifiedTx $\subset \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^*$, verifiedTx = \emptyset	{List of verified transactions, either by this notary or others, as (txId, tx, σ) where σ is meant to be the notarization signature
- attachments $\subset \{0, 1\}^* \times \{0, 1\}^*$, attachments = \emptyset	{Set of known attachments and identifiers as (attachmentId, attachment)
- bufferTxSig $\subset \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \times \{0, 1\}^* \times \text{txStatus}^{\mathbf{c}, a}$ bufferTxSig = \emptyset	{Transaction currently in process as (txId, tx, Σ , pid, status). Σ is a set of tuples of the form (pid, σ)
CheckID (pid, sid, role):	
Accept all messages addressed to the same entity (pid, sid, role). Only accept role = notary.	
Corruption behavior:	
DetermineCorrStatus (pid, sid, role):	
if corr = true: return true	{Checks whether notary itself is corrupted.
corrRes \leftarrow corr(pid _{cur} , sid _{cur} , signer)	{Checks whether $\mathcal{F}_{\text{cert}}$ instance is corrupted.
if corrRes = true: return true	
for all formerNotary from tx in verifiedTx where formerNotary \neq pid _{cur} do:	
corrRes \leftarrow corr(formerNotary, sid _{cur} , signer)	
if corrRes: return true	{Checks if there was a notary change from a corrupted notary
We expect the ITM to enforce the transaction format $tx = (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal})$ where in <i>proposal</i> input states are of form $tx_{\text{inputStates}} = (\text{txId}, \text{outputIndex})$ and output states are of form $tx_{\text{outputStates}}$. The symbol \oplus emphasizes that we use this format specification.	
Main:	
recv (Notarise, (pids, (tx, Σ))) from (_, sid _{cur} , $\mathcal{F}_{\text{unicast}} : \text{unicast}$) s.t. initiator = pids \oplus :	
require: executeValidation(tx) = true	{Notarises transaction tx if valid and no double-spending
if formerNotary \neq pid _{cur} :	{Do not process wrongly addressed transactions
send (Message, (initiator, (NotariseRes, (tx, misdirected)))) to (pid _{cur} , sid _{cur} , $\mathcal{F}_{\text{unicast}}$)	
else if verifiedTx.contains(_, tx, _):	{tx was already notarised, return stored signature
(txId, tx, Σ) \leftarrow verifiedTx[tx]	
if $\sigma \in \{\perp, \epsilon\}$:	{Tx was accepted but notarization is missing
send (Sign, tx) to (pid _{cur} , sid _{cur} , $\mathcal{F}_{\text{cert}} : \text{signer}$)	
wait for (Signature, σ')	
$\sigma \leftarrow \sigma'$	
verifiedTx[tx][2] $\leftarrow \sigma$	{Store notarization
send (Message, (initiator, (NotariseRes, (tx, σ)))) to (pid _{cur} , sid _{cur} , $\mathcal{F}_{\text{unicast}} : \text{unicast}$)	
else:	{Process and validate tx
send tx to (pid _{cur} , sid _{cur} , $\mathcal{F}_{\text{ro}} : \text{randomOracle}$)	
wait for (txId)	
bufferTxSig.add((TX.add(txId), tx, Σ , pids, sign))	{Record tx, processing status, and additional data in bufferTxSig
res \leftarrow validate(tx, Σ)	{See definition of function validate below
msg \leftarrow validateAllLocal()	
^a For notaries, we set $\text{txStatus}^{\mathbf{c}} = \{\perp, \text{requested}, \text{unrequested}, \text{reqDeps}, \text{sign}\}$. The set $\text{txStatus}^{\mathbf{c}}$ includes the different statuses a transaction passes during processing.	

Figure 30: The Corda notary $\mathcal{P}_{\text{notary}}^{\mathbf{c}}$ (Pt. 1)

Main:

```

recv (RecvTransactionFlow, (pids, (tx,  $\Sigma$ ))) from (_, sidcur,  $\mathcal{F}_{\text{unicast}}$  : unicast): {Entity processes a requested transaction}
require: executeValidation(tx) = true
send tx to (pidcur, sidcur,  $\mathcal{F}_{\text{ro}}$  : randomOracle) {Get transaction ID}
wait for (txId')
txId ← TX.add(txId')
if bufferTXSig.contains((txId,  $\perp$ ,  $\perp$ , pids, requested)):
  res ← validate(tx,  $\Sigma$ , pids) {See definition of validate below}
  if res = valid  $\wedge$  | $\Sigma$ | = m + 2:
    verifiedTx.add(bufferTXSig[tx][0], bufferTXSig[tx][1], bufferTXSig[tx][2])
    bufferTXSig.remove(bufferTXSig[tx])
  validateAllLocal()

recv (RecvAttachment, (pids, attachment)) from (_, sidcur,  $\mathcal{F}_{\text{unicast}}$  : unicast): {Stores received attachments in attachments.}
send (attachment) to (pidcur, sidcur,  $\mathcal{F}_{\text{ro}}$  : randomOracle) {Get attachment ID}
wait for (attachmentId')
if validateAttachment(attachment):
  attachments.add(ATT.add(attachmentId'), attachment) {Store attachment}
  validateAllLocal()

```

Procedures and Functions:

```

function validateAllLocal() : {Checks for all stored transactions if the inputs and attachments can
  identities ←  $\epsilon$ , msg ←  $\epsilon$ , resolved ← true {be resolved and the transaction therefore validated}
  while resolved = true do
    resolved ← false, valid ←  $\epsilon$ 
    for all (_, tx,  $\Sigma$ , pid, status)  $\in$  bufferTXSig, s.t.
      status  $\notin$  { $\perp$ , unrequested} do:
        if isResolvable(tx): {isResolvable checks whether full dependencies are available in
          valid ← validate(tx,  $\Sigma$ ,  $\epsilon$ ) {verifiedTx and attachments}
          if valid = valid  $\wedge$  status  $\in$  {sign, waitSign}: {Need to agree on tx}
            send (Sign, tx) to (pidcur, sidcur,  $\mathcal{F}_{\text{cert}}$  : signer)
            wait for (Signature,  $\sigma$ )
            bufferTXSig.remove(bufferTXSig[tx])
            verifiedTx.add(txId, tx,  $\sigma$ )
            msg.add((initiator, (Notarise, (tx,  $\sigma$ )))) {Send notarization to initiator}
          if valid = valid  $\wedge$  status  $\in$  {requested, reqDeps}: {Validate/accept transactions with formerly
            verifiedTx.add(bufferTXSig[tx][0], bufferTXSig[tx][1], bufferTXSig[tx][2]) {missing dependencies}
            bufferTXSig.remove(bufferTXSig[tx])
          resolved ← true
  send (Message, msg) to (pidcur, sidcur,  $\mathcal{F}_{\text{unicast}}$ )

```

Figure 31: The Corda notary $\mathcal{P}_{\text{notary}}^c$ (Pt. 2)

Procedures and Functions:

```

function validate( $tx, \Sigma$ ):
  if verifiedTx.contains( $\_, tx, \_, \Sigma'$ )  $\wedge \Sigma' \neq \Sigma$ :
    if  $|\Sigma'| = m + 2$ :
      bufferTxSig.remove(bufferTxSig[ $tx$ ]); return alreadyNotarised
    else:
      if isDoubleSpend( $tx, verifiedTx$ ):
        {Checks whether one of the input states is already used in another tx in verifiedTx}
        bufferTxSig.remove(bufferTxSig[ $tx$ ]); return doubleSpend
      if  $\neg[\forall i \in [m] : isValidId(signee_i, client) = \text{true} \wedge$ 
        isValidId( $initiator, client$ ) =  $\text{true} \wedge$ 
        isValidId( $notary, notary$ ) =  $\text{true} \wedge$ 
        isValidId( $formerNotary, notary$ ) =  $\text{true}]$ :
        {isValidId checks whether the given pid is registered with the particular role in networkmap.
        formerNotary not checked as we process the notarization currently.}
        bufferTxSig.remove(bufferTxSig[ $tx$ ]); return invalid
      if  $|\Sigma| \neq m + 1$ :
        bufferTxSig.remove(bufferTxSig[ $tx$ ]); return invalid
        {There are missing signatures}
      if  $\neg(\forall (pid', \_) \in \Sigma, \text{it holds true that } (pid' = initiator \vee \exists i \in [m], \text{s.t. } signee_i = pid'))$ :
        bufferTxSig.remove(bufferTxSig[ $tx$ ])
        return invalid
        {... all signatures need to be from signees or the initiator}
      for all  $(\sigma, pid_\sigma) \in \Sigma$  do:
        send ( $tx, \sigma$ ) to  $(pid_\sigma, sid_{cur}, \mathcal{F}_{cert} : \text{verifier})$ 
        wait for  $(\text{VerResult}, res)$ 
        {Checks all provided signatures}
        if  $res = \text{false}$ :
          bufferTxSig.remove(bufferTxSig[ $tx$ ]); return invalid
        if isResolvable( $tx$ ) =  $\text{false} \wedge$  bufferTxSig[ $tx$ ][4]  $\neq$  reqDeps:
          {Checks whether all dependencies are available}
           $msg \leftarrow \text{resolveTx}(tx, initiator)$ 
          {If dependencies are missing, query the initiator}
          send ( $\text{Message}, msg$ ) to  $(pid_{cur}, sid_{cur}, \mathcal{F}_{unicast} : \text{unicast})$ 
          {Party has access to all dependencies of tx}
        else if isResolvable( $tx$ ):
          Let txDependencies be the set of all transaction bodies of (transaction) dependencies of tx from verifiedTx and
          bufferTxSig.
          {Collect dependencies for upcoming validation}
          Let txAttachments be the set of all attachments referenced/used in (transaction) dependencies of tx from verifiedTx
          and bufferTxSig.
           $res \leftarrow \text{executeValidation}(tx, txDependencies, txAttachments)$ 
          {Validate the transaction regarding the given context}
          return  $res$ 

function resolveTx( $tx, pids$ ) ( $msg$ ):
  if bufferTxSig[ $tx$ ][4] =  $\text{sign}$ : bufferTxSig[ $tx$ ][4]  $\leftarrow$  waitSign
  else: bufferTxSig[ $tx$ ][4]  $\leftarrow$  reqDeps
  {Proceed with signing after getting dependencies}
   $msg \leftarrow \epsilon$ 
  for all  $(txId', \_) \in tx_{inputStates}$  do:
    {For definition of tx_inputStates see ⊕}
    if  $\nexists (txId', \_, \_) \in \text{verifiedTx}$ :
      {Request all transactions necessary for verification but currently not in verifiedTx}
      bufferTxSig.add( $(txId', \_, \_, pids, requested)$ )
       $msg.add((pids, (\text{SendTransactionFlow}, (txId'))))$ 
    for all  $attachmentId \in \text{proposal}$  do:
      if  $(attachmentId, \_) \notin \text{attachments}$ :
        {Request all attachments necessary for verification but currently not in attachments}
         $msg.add((pids, (\text{GetAttachment}, (attachmentId))))$ 
  return  $msg$ 

```

Figure 32: The Corda notary $\mathcal{P}_{\text{notary}}^c$ (Pt. 3)

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

Participating roles: {unicast} Corruption model: <i>incorruptible</i> Protocol parameters: - $\delta \in \mathbb{N}$	<i>{Delay parameter: Models the time the adversary is allowed to prevent messages from being delivered}</i>
--	---

Description of M_{unicast} :

Implemented role(s): {unicast} Subroutines: $\mathcal{P}_{\text{client}}^c$: client, $\mathcal{P}_{\text{notary}}^c$: notary Internal state: - $\text{buffer}_{\text{msg}} \subset \mathbb{N} \times \mathbb{N} \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*$, $\text{buffer}_{\text{msg}} = \emptyset$ - $\text{labels} \in \mathbb{N}$, $\text{labels} = \emptyset$ - $\text{round} \in \mathbb{N}$, $\text{round} = 0$ CheckID (<i>pid, sid, role</i>): Accept all messages with the same <i>sid</i> . Main: rcv (Message, <i>msg</i>) from I/O s.t. $\text{msg} \subset \{(receiver, content) \mid receiver, content \in \{0,1\}^*\}$: <i>leakage</i> $\leftarrow \epsilon$ for all (<i>r, m</i>) $\in \text{msg}$ do : <i>label</i> $\xrightarrow{\$} \mathbb{N}$, s.t. <i>label</i> $\notin \text{labels}$ labels.add(<i>label</i>) <i>leakage</i> .add(<i>label</i> , pid _{call} , <i>r</i> , <i>m</i>) buffer _{msg} .add(<i>label</i> , round, pid _{call} , <i>r</i> , <i>m</i>) send (MultiMessage, <i>leakage</i>) to NET rcv (Deliver, <i>label</i>) from NET: parse (<i>label, _</i> , <i>s, r, m</i>) from buffer _{msg} [<i>label</i>] parse (Command, <i>content</i>) from <i>m</i> buffer _{msg} .remove(<i>label, _</i> , <i>s, r, m</i>) send (Command, (<i>s, content</i>)) to (<i>r, sid</i> , I/O) rcv (UpdateRound) from NET: if $\exists (_, r', _, _) \in \text{buffer}_{\text{msg}} : r' < \text{round} - \delta$: reply (UpdateRound, false, ϵ) else : round $\leftarrow \text{round} + 1$ reply (UpdateRound, true, ϵ) rcv (GetCurRound) from I/O or NET: reply (GetCurRound, round)	<i>{$\mathcal{F}_{\text{unicast}}$ can access the corruption status of clients and notaries}</i> <i>{Buffer for messages consisting of tuples (label, round, sender, receiver, content)}</i> <i>{Already used identifiers}</i> <i>{Current round/time}</i> <i>{Request to deliver several messages to different receivers}</i> <i>{Generate a unique label per message that needs to be delivered}</i> <i>{Record receiver, round, and content length as leakage}</i> <i>{Queue message for delivery}</i> <i>{Forward leakage to \mathcal{A}}</i> <i>{\mathcal{A} triggers delivery of messages}</i> <i>{Fetch data from message queue}</i> <i>{Remove message from delivery queue}</i> <i>{Deliver message to receiver}</i> <i>{\mathcal{A} increasing round ok if all no messages are younger than $\delta -$ rounds}</i> <i>{Ensure all messages are delivered within δ rounds}</i> <i>{Round update accepted}</i> <i>{\mathcal{A} and \mathcal{E} are allowed to query the current round}</i>
--	--

Figure 33: The unicast functionality $\mathcal{F}_{\text{unicast}}$ with eventual message delivery and bounded message dealy

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

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

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

Implemented role(s): {signer, verifier}	
Internal state:	
- $(\text{pk}, \text{sk}) \in (\{0, 1\}^* \cup \{\perp\})^2 = (\perp, \perp)$.	{Key pair.
- $\text{pidowner} \in \{0, 1\}^* \cup \{\perp\} = \perp$.	{Party ID of the key owner.
- $\text{msglist} \subset \{0, 1\}^* = \emptyset$.	{Set of recorded messages.
- $\text{corr} \in \{\text{true}, \text{false}\} = \text{false}$.	{Is signature key corrupted?}
CheckID ($\text{pid}, \text{sid}, \text{role}$):	
Check that $\text{sid} = (\text{pid}', \text{sid}')$:	
If this check fails, output reject.	
Otherwise, accept all entities with the same SID.	
{A single instance manages all parties and roles in a single session. A session models one signature key pair belonging to party pid' .	
Corruption behavior:	
- DetermineCorrStatus ($\text{pid}, \text{sid}, \text{role}$): Return corr.	
Initialization:	
$(\text{pk}, \text{sk}) \xleftarrow{\$} \text{Gen}(1^\eta)$	{Generate public/secret key pair
Parse sid_{cur} as (pid, sid) .	
$\text{pidowner} \leftarrow \text{pid}$.	
Main:	
rcv (Sign, msg) from I/O to ($\text{pidowner}, _$, signer):	
$\sigma \leftarrow \text{sig}^{(p)}(\text{msg}, \text{sk})$.	
add msg to msglist.	
reply (Signature, σ).	{Record msg for verification and return signature.
rcv (Verify, msg, σ) from I/O to ($_$, $_$, verifier):	
$b \leftarrow \text{ver}^{(p)}(\text{msg}, \sigma, \text{pk})$.	{Verify signature.
if $b = \text{true} \wedge \text{msg} \notin \text{msglist} \wedge \text{corr} = \text{false}$:	
reply (VerResult, false).	{Prevent forgery.
else:	
reply (VerResult, b).	{Return verification result.
rcv corruptSigKey from NET:	{Allow network attacker to corrupt signature keys.
$\text{corr} \leftarrow \text{true}$.	
reply (corruptSigKey, ok).	

Figure 34: The ideal certificate functionality $\mathcal{F}_{\text{cert}}$ with in-built PKI

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

Participating roles: $\{\text{randomOracle}\}$ Corruption model: <i>incorruptible</i> Protocol parameters: - $\eta \in \mathbb{N}$	$\{\text{Security parameter, length of the hash}\}$
--	---

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

Implemented role(s): $\{\text{randomOracle}\}$ Internal state: - $\text{hashHistory} \subseteq \{0, 1\}^* \times \{0, 1\}^\eta$, initially $\text{hashHistory} = \emptyset$	$\{\text{The set of recorded value/hash pairs}\}$
CheckID ($pid, sid, role$): Accept all messages with the same sid .	
Main: rcv (pid, x) from I/O or NET: if $\exists h \in \{0, 1\}^\eta$ s.t. $(x, h) \in \text{hashHistory}$: reply (pid, h) else: $h \xleftarrow{\$} \{0, 1\}^\eta$ $\text{hashHistory} \leftarrow \text{hashHistory.add}((x, h))$ reply (pid, h)	$\{\text{Requesting the } \mathcal{F}_{\text{ro}} \text{ for "hashes"}\}$ $\{\text{Extract existing value from hashHistory}\}$ $\{\text{Generate "hash value" uniformly at random}\}$ $\{\text{Store generated key, value pair in hashHistory}\}$

Figure 35: The Random Oracle \mathcal{F}_{ro}

Description of the subroutine $\mathcal{F}_{\text{submit}}^{\mathbf{c}} = (\text{submit})$:

Participating roles: $\{\text{submit}\}$ Corruption model: <i>incorruptible</i> Protocol parameters: - executeValidation - $\text{validateAttachment}$	$\{\text{Validation algorithm that verifies whether it is ok to add a transaction to the current Corda state}\}$ $\{\text{Algorithm that validates attachments}\}$
--	---

Description of $M_{\text{submit}}^{\mathbf{c}}$:

Implemented role(s): $\{\text{submit}\}$ Subroutines: $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage, $\mathcal{F}_{\text{read}}^{\mathbf{c}}$: read CheckID ($pid, sid, role$): Accept all messages with the same sid .	
<p>In the following, we use the abbreviation (1) for the expression $\text{msg} = \text{tx} = [\text{tx}, (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal})]$, (2) for $\text{msg} = (\text{attachment}, \text{attachment})$, and (3) for $\text{msg} = (\text{tx}, \text{txId}, \text{pidr})$. We expect the ITM to enforce the transaction format $\text{tx} = (\text{tx}, (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal}))$ where in <i>proposal</i> input states are of form $\text{tx}_{\text{inputStates}} = (\text{txId}, \text{outputIndex})$ and output states are of form $\text{tx}_{\text{outputStates}}$. We sometimes emphasize with the symbol \oplus that we use this format specification.</p>	
Main: rcv ($\text{Submit}, \text{msg}, \text{internalState}$) from I/O: if $\neg[(1) \vee (2) \vee (3)]$: reply ($\text{Submit}, \text{false}, \epsilon$) if (1) $\wedge \text{pid}_{\text{cur}} \notin \{\text{initiator}, \text{signee}_1, \dots, \text{signee}_m\}$: reply ($\text{Submit}, \text{false}, \epsilon$) if (1) $\wedge \neg[\text{initiator}, \text{signee}_1, \dots, \text{signee}_m \text{ are clients (role = client, pid prefixed by client)} \wedge \text{notary, formerNotary are notaries (role = notary and pid prefixed by notary)} \wedge \forall \text{pid} \in \{\text{initiator}, \text{signee}_1, \dots, \text{signee}_m, \text{notary, formerNotary}\} : \text{pid} \in \text{identities}]$: reply ($\text{Submit}, \text{false}, \epsilon$)	$\{\text{See Figure 5 for definition of internalState and the local variables it includes}\}$ $\{\text{Submitted transactions/attachments need to have the expected data format}\}$ $\{\text{Clients can only submit transactions they are involved in}\}$ $\{\text{Check correct identities and roles}\}$ $\{\text{Clients can submit transactions they are involved in, all parties need to be registered}\}$

Figure 36: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s submit functionality $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ (Pt. 1)

Main:

```

if (1):
    send (GetID, tx, (initiator, [signee1, ..., signeem], notary, formerNotary, proposal))
    to (pidcur, sidcur,  $\mathcal{F}_{\text{storage}}^c$ )
    wait for (GetID, tx, id); msg.remove(tx)
    parse [(inTxId1, idx1), ..., (inTxIdh, idxh)], (out1, ..., oute) from proposal
    leakage ← [(id1a, ..., idla), ((inTxId1, idx1), ..., (inTxIdh, idxh))](out1, ..., oute)
    leakage.add[id, initiator, signee1, ..., signeem, formerNotary]
    if initiator = pidcur:
        send getCurrentKnowledge to (pidcur, sidcur,  $\mathcal{F}_{\text{read}}$  : read)
        wait for (getCurrentKnowledge, transactionspidcurc, attachmentspidcurc)
        send getKnowledge to (pidcur, sidcur,  $\mathcal{F}_{\text{storage}}^c$  : storage)
        wait for (getKnowledge, txGraphpidcur, attachmentspidcur)
        if missingDependencya(msg, transactionspidcurc, attachmentspidcurc):
            reply (Submit, false,  $\epsilon$ )
        Let txDependencies be the set of all transaction bodies of (transaction) dependencies of msg from msglist and requestQueue (which are in transactionspidcurc).
        Let txAttachments be the set of all attachments referenced/used in (transaction) dependencies of msg from msglist and requestQueue (which are in attachmentspidcurc).
        if !executeValidation(msg, txDependencies, txAttachments):
            reply (Submit, false,  $\epsilon$ )
        if isDoubleSpend(msg):
            reply (Submit, false,  $\epsilon$ )
    if one of the parties initiator, signee1, ..., signeem, notary, formerNotary is in CorruptionSet:
        leakage.add(proposal)
        if pidcur = initiator:
            txGraphpidcur.addToTxGraph(id, msg,  $\emptyset$ )
            (txSubGraph, subGraphAttachments) ← getConnectedSubGraphb(msg, txGraphpidcur)
            for all transactions (txID, tx, attachmentstx) in txSubGraph do:
                leakage.add(txID, tx, attachmentstx)
            for all attachments (ida, attachment) in subGraphAttachments do:
                leakage.add(ida, attachment)
        else:
            leakage.add(|proposal|)
    reply (Submit, true, leakage)

```

^amissingDependency(msg, transactions_{pid_{cur}}^c, attachments_{pid_{cur}}^c) returns false if all input transactions of a tx, here msg, and their full dependencies, i. e., transactions down to the issuance transactions, are available in transactions_{pid_{cur}}^c as well as all attachments used in these subgraph(s) below the tx are available in attachments_{pid_{cur}}^c, otherwise true.

^bgetConnectedSubGraph(msg, txGraph_{pid_{cur}}) outputs the (maximal) connected subgraph of txGraph_{pid_{cur}} such that msg's outputs are all arcs of this subgraph ("maximal" means \nexists a connected subgraph txGraph containing msg's output and that is a superset of getConnectedSubGraph(msg, txGraphs)). During processing, getConnectedSubGraph queries $\mathcal{F}_{\text{storage}}^c$ for transaction details (txID, tx, attachments_{tx}) and details of the used attachments (id_a, attachment). The output of getConnectedSubGraph is (Tx, Attachments) where Tx is the set of all transactions from the subgraph including their details and Attachments is the set of all used attachment details.

Figure 37: $\mathcal{F}_{\text{ledger}}^c$'s submit functionality $\mathcal{F}_{\text{submit}}^c$ (Pt. 2)

<p>Main:</p> <p>if (2):</p> <p style="padding-left: 20px;">send (GetID, attachment, attachment) to (pid_{cur}, sid_{cur}, $\mathcal{F}_{\text{storage}}^c$)</p> <p style="padding-left: 20px;">wait for (GetID, attachmentid)</p> <p style="padding-left: 20px;">leakage.add(id, attachment)</p> <p style="padding-left: 20px;">reply (Submit, validateAttachment(attachment), leakage)</p> <p>if (3) \wedge pid_{cur} is a client (role = client and pid_{cur} prefixed by client):</p> <p style="padding-left: 20px;">send getCurrentKnowledge to (pid_{cur}, sid_{cur}, $\mathcal{F}_{\text{read}}$: read)</p> <p style="padding-left: 20px;">wait for (getCurrentKnowledge, transactions_{pid_{cur}}^c, attachments_{pid_{cur}}^c)</p> <p style="padding-left: 20px;">send (GetContent, tx, txID) to</p> <p style="padding-left: 20px;">wait for (GetContent, tx, tx)</p> <p style="padding-left: 20px;">send getKnowledge to (pid_{cur}, sid_{cur}, $\mathcal{F}_{\text{storage}}^c$: storage)</p> <p style="padding-left: 20px;">wait for (getKnowledge, txGraph_{pid_{cur}}, attachments_{pid_{cur}})</p> <p style="padding-left: 20px;">if tx \in transactions_{pid_{cur}}^c \wedge \negmissingDependency(tx, transactions_{pid_{cur}}^c, attachments_{pid_{cur}}^c):</p> <p style="padding-left: 40px;">{pid_{cur} has access to tx and can forward it, missingDependency is defined in Footnote a in Figure 37 on Page 69}</p> <p style="padding-left: 20px;">if pidr in CorruptionSet:</p> <p style="padding-left: 40px;">send (getTxGraph, internalState, incBuffer) to (pid_{cur}, sid_{cur}, $\mathcal{F}_{\text{storage}}^c$: storage)</p> <p style="padding-left: 60px;">{Generate transaction graph including tx from requestQueue, located in $\mathcal{F}_{\text{storage}}^c$}</p> <p style="padding-left: 40px;">wait for (getTxGraph, txGraph)</p> <p style="padding-left: 40px;">(txSubGraph, subGraphAttachments) \leftarrow getConnectedSubGraph(tx, txGraph)</p> <p style="padding-left: 60px;">{getConnectedSubGraph is specified in Footnote b in Figure 37 on Page 69}</p> <p style="padding-left: 40px;">for all transactions (txID, tx, attachments_{tx}) in txSubGraph do:</p> <p style="padding-left: 60px;">leakage.add(txID, tx, attachments_{tx})</p> <p style="padding-left: 60px;">{Leak full tx details}</p> <p style="padding-left: 40px;">for all attachments (id_a, attachment) in subGraphAttachments do:</p> <p style="padding-left: 60px;">leakage.add(id_a, attachment)</p> <p style="padding-left: 60px;">{Leak used attachments}</p> <p style="padding-left: 40px;">reply (Submit, true, leakage)</p> <p style="padding-left: 20px;">else:</p> <p style="padding-left: 40px;">leakage \leftarrow (txID, pid_{cur}, pidr)</p> <p style="padding-left: 40px;">reply (Submit, true, leakage)</p> <p style="padding-left: 60px;">{\mathcal{A} is informed that party pushed tx to entity}</p> <p style="padding-left: 20px;">else:</p> <p style="padding-left: 40px;">reply (Submit, false, ϵ)</p> <p style="padding-left: 60px;">{Input rejected}</p> <p style="padding-left: 20px;">reply (Submit, false, ϵ)</p> <p style="padding-left: 60px;">{Input rejected}</p>	<p>{In case that an attachment was submitted}</p> <p>{Request ID of the attachment}</p> <p>{$\mathcal{F}_{\text{storage}}^c$ returns an ID or false}</p> <p>{\mathcal{A} receives the length of the attachment as leakage}</p> <p>{A client pushes a transaction to another entity}</p> <p>{Query pid_{cur}'s current knowledge}</p> <p>{Request tx content}</p> <p>{$\mathcal{F}_{\text{storage}}^c$ returns an ID or false}</p> <p>{tx is pushed to a corrupted party}</p>
---	---

Figure 38: $\mathcal{F}_{\text{ledger}}^c$'s submit functionality $\mathcal{F}_{\text{submit}}^c$ (Pt. 3)

Description of the subroutine $\mathcal{F}_{\text{read}}^{\mathbf{c}} = (\text{read})$:

Participating roles: {read}
Corruption model: *incorruptible*

Description of $M_{\text{read}}^{\mathbf{c}}$:

Implemented role(s): {read}

Subroutines: $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage

Internal state:

- knownTransactions : $\{0, 1\}^* \times \{0, 1\}^*$ {A party's known transactions. Entries of form (pid, txID)}
- knownAttachments : $\{0, 1\}^* \times \{0, 1\}^*$ {A party's known attachments. Entries of form (pid, id^a)}
- knownTransactions^r : $\{0, 1\}^* \times \{0, 1\}^*$ {A party's last transaction output}
- knownAttachments^r : $\{0, 1\}^* \times \{0, 1\}^*$ {A party's last attachments output}

CheckID(pid, sid, role):

Accept all messages with the same sid.

In the following, we use the abbreviation (1) for the expression $tx = [tx, (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal})]$.

Main:

recv (InitRead, msg, internalState) **from** I/O:

reply (InitRead, true, ϵ)

{Reads in Corda are always local, we do not leak anything to \mathcal{A} }

recv (FinishRead, msg, input, internalState) **from** I/O:

s.t. input = ($[txID_1, \dots, txID_h], [id_1^a, \dots, id_o^a]$)

{Read outputs the tx subgraph pid_{cur} is aware of

Execute code from getCurrentKnowledge here. Ignore its output.

{Update pid_{cur} 's knowledge

$\text{knownTX}^r \leftarrow \{txID' \mid (\text{pid}_{\text{cur}}, txID') \in \text{knownTransactions}^r\}$

{Extract pid_{cur} 's knowledge and what was read by pid_{cur} }

$\text{knownTX} \leftarrow \{txID' \mid (\text{pid}_{\text{cur}}, txID') \in \text{knownTransactions}\}$

$\text{knownAtt}^r \leftarrow \{id' \mid (\text{pid}_{\text{cur}}, id') \in \text{knownAttachments}^r\}$

$\text{knownAtt} \leftarrow \{id' \mid (\text{pid}_{\text{cur}}, id') \in \text{knownAttachments}\}$

if $\text{knownTX}^r \subset \{txID_1, \dots, txID_h\} \subset \text{knownTX} \vee \text{knownAtt}^r \subset \{id_1^a, \dots, id_o^a\} \subset \text{knownAtt}$:

$\text{verifiedTx} \leftarrow \text{msglist}$

for all $(txID', tx', _) \in \text{verifiedTx}$, **s.t.** $txID' \notin \{txID_1, \dots, txID_h\}$ **do**:

{Remove entries that are not in the suggested state}

$\text{verifiedTx.remove}(txID', tx', _)$

if $\exists tx'$ in msglist , **s.t.** tx' is not in $\text{verifiedTx} \wedge \text{pid}_{\text{cur}}$ is initiator of $tx' \wedge$ all parties involved in tx' are not corrupted \wedge round is greater than committing round of $tx' + \delta$:

reply (FinishRead, \perp)

{Initiators of "honest" tx are guaranteed that their tx is after at most δ rounds in their states}

if $\exists tx'$ in msglist , **s.t.** tx' not in $\text{verifiedTx} \wedge \text{pid}_{\text{cur}}$ is signee of $tx' \wedge$ all parties involved in tx' are not corrupted \wedge round is greater than committing round of $tx' + 2\delta$:

reply (FinishRead, \perp)

{Signees of "honest" tx are guaranteed that the tx is after at most 2δ rounds after notarization in their states}

if \exists an operation in msglist that instructs pushing tx' to pid_{cur} , **s.t.** tx' not in $\text{verifiedTx} \wedge$ all parties involved in tx' are not corrupted \wedge round is greater than committing round of $tx' + |\text{subGraph}|(tx')\delta + 1$, **s.t.** $\text{subGraph}(tx')$ is the subgraph below tx' :

reply (FinishRead, \perp)

{Push of transaction should be finished after δ rounds}

$\text{cleanedVerifiedTx} \leftarrow \emptyset, \text{attachments} \leftarrow \emptyset$

{Separate attachment data from tx data}

for all $(txID, tx, _) \in \text{verifiedTx}$ **do**: $\text{cleanedVerifiedTx.add}(txID, tx)$

for all $id^a \in \{id_1^a, \dots, id_o^a\}$ **do**:

{Output suggested attachments}

Let attachment **s.t.** $(id^a, \text{attachment}) \in \text{attachments}_{\text{pid}_{\text{cur}}}$; $\text{attachments.add}(id^a, \text{attachment})$

if there is a transaction in cleanedVerifiedTx **s.t.** parts of its transaction inputs or the transaction subgraph below the transaction are not in cleanedVerifiedTx :

reply (FinishRead, \perp)

{Decline input}

Add all new entries of newly read transactions/attachments to $\text{knownTransactions}^r/\text{knownAttachments}^r$.

reply (FinishRead, (attachments, cleanedVerifiedTx), ϵ)

Figure 39: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s read functionality $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ (Pt. 1)

Main:

```

recv (CorruptedRead,  $pid$ ,  $msg$ ,  $internalState$ ) from I/O: {No CorruptedRead modeled}
reply (FinishRead,  $\epsilon$ )

recv getCurrentKnowledge from I/O: {Other subroutines may ask  $\mathcal{F}_{\text{read}}^{\mathbf{c}}$  for an entity's current state}
send getKnowledge to ( $pid_{\text{cur}}$ ,  $sid_{\text{cur}}$ ,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage) {Request  $pid_{\text{cur}}$ 's knowledge at  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ }
wait for (getKnowledge,  $txGraph_{pid_{\text{cur}}}$ ,  $attachments_{pid_{\text{cur}}}$ )
send responsively (getKnowledge,  $pid_{\text{cur}}$ ) to NET {Query current state of  $pid_{\text{cur}}$  at  $\mathcal{A}$ }
wait for (getKnowledge,  $pid_{\text{cur}}$ , ( $txID_1, \dots, txID_h$ ), ( $id_1^a, \dots, id_o^a$ ))
 $knownTX \leftarrow \{txID' \mid (pid_{\text{cur}}, txID') \in knownTransactions\}$  {Extract previous state of  $pid_{\text{cur}}$ }
 $knownAtt \leftarrow \{id' \mid (pid_{\text{cur}}, id') \in knownAttachments\}$ 
if  $knownTX \subset \{txID_1, \dots, txID_h\} \wedge knownAtt \subset \{id_1^a, \dots, id_o^a\}$ : {(Start to) update  $pid_{\text{cur}}$ 's knowledge}
   $hasAccess \leftarrow \text{false}$ ;  $nonGraphAccess \leftarrow \text{true}$  {if this is compliant with the view of}
  if  $\forall txID' \in \{txID_1, \dots, txID_h\}$ , it holds that  $txID' \in txGraph_{pid_{\text{cur}}} \wedge$  { $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ }
     $\forall id' \in \{id_1^a, \dots, id_o^a\}$ , it holds that  $id' \in attachments_{pid_{\text{cur}}}$ :
  else:
    for all  $txID' \in \{txID_1, \dots, txID_h\}$  do:
      if  $txID' \notin txGraph_{pid_{\text{cur}}}$ : {Check that the adversary only distributes knowledge}
        send (GetContent,  $tx$ ,  $txID'$ ) to ( $pid_{\text{cur}}$ ,  $sid_{\text{cur}}$ ,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage) {she has access to}
        wait for (GetContent,  $tx$ ,  $tx'$ )
        if  $\nexists$  corrupted party in  $tx$ :
           $nonGraphAccess \leftarrow \text{false}$ 
        for all  $id' \in \{id_1^a, \dots, id_o^a\}$  do:
          if  $id' \notin attachments_{pid_{\text{cur}}}$ : {Check that the adversary only dis-}
            send (GetContent, attachment,  $id'$ ) to {tributes knowledge she has access to}
            wait for (GetContent, attachment,  $attachment'$ ) from
            if  $attachment'$  in  $requestQueue$  or  $msglist \wedge \nexists$  a leakage of attachment in transcript:
               $nonGraphAccess \leftarrow \text{false}$ 
          if  $hasAccess \wedge hasNonGraphAccess$ : {Update  $pid_{\text{cur}}$ 's knowledge if checks succeeded}
            for all  $txID' \in \{txID_1, \dots, txID_h\} \setminus knownTX$  do:
               $knownTransactions.add(pid_{\text{cur}}, txID')$ 
            for all  $id' \in \{id_1^a, \dots, id_o^a\} \setminus knownAtt$  do:
               $knownAttachments.add(pid_{\text{cur}}, id')$ 
           $knownTransactions_{pid_{\text{cur}}} \leftarrow \{txID \mid (pid_{\text{cur}}, txID) \in knownTransactions\}$  {Extract previous state of  $pid_{\text{cur}}$ }
           $knownAttachments_{pid_{\text{cur}}} \leftarrow \{id' \mid (pid_{\text{cur}}, id') \in knownAttachments\}$ 
reply (getCurrentKnowledge,  $transactions_{pid_{\text{cur}}}$ ,  $attachments_{pid_{\text{cur}}}$ )

```

Figure 40: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s read functionality $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ (Pt. 2)

Description of the subroutine $\mathcal{F}_{\text{update}}^{\mathbf{c}} = (\text{update})$:

Participating roles: {update} Corruption model: <i>incorruptible</i> Protocol parameters: <ul style="list-style-type: none"> - executeValidation - validateAttachment 	$\left\{ \begin{array}{l} \text{Validation algorithm that verifies whether it is ok to add a transaction to the current Corda state} \\ \text{Algorithm that validates attachments} \end{array} \right.$
--	--

Description of $M_{\text{update}}^{\mathbf{c}}$:

Implemented role(s): {update} Subroutines: $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage CheckID (<i>pid, sid, role</i>): Accept all messages with the same <i>sid</i> . The ITM enforces the transaction format $tx = (tx, (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal}))$ where in <i>proposal</i> input states as $tx_{\text{inputStates}} = (txID, \text{outputIndex})$ and output states as $tx_{\text{outputStates}}$. We may use the symbol \oplus to point to this format specification here. Main: recv (Update, [normal, msg], internalState) from I/O: $msgListAppend \leftarrow \emptyset, updRequestQueue \leftarrow \emptyset, ctr \leftarrow \max\{i \mid (i, _ , _ , _ , _) \in msglist\}$ $\left\{ \begin{array}{l} \text{Build extension of msglist. If msglist} = \emptyset \text{ then max defaults to } -1 \\ \text{input, may update them, and create new variables} \end{array} \right.$ executeBasicChecks() for all (<i>id', pid'</i>) $\in \{(id_1, pid_1), \dots, (id_1, pid_1)\}$ do: $\left\{ \begin{array}{l} \text{Add attachments from honest parties} \\ \text{Get attachment content from } \mathcal{F}_{\text{storage}}^{\mathbf{c}} \end{array} \right.$ send (GetContent, attachment, <i>id'</i>) to ($_ , sid_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage) wait for (GetContent, attachment, <i>attachment'</i>); $ctr \leftarrow ctr + 1$ $msgListAppend.add(ctr, round, tx, attachment', round, pid')$ $\left\{ \begin{array}{l} \text{Add attachments from } \mathcal{A} \end{array} \right.$ Let ($ctr_{\text{tmp}}, (\text{attachment}, attachment'), r, pid'$) be the matching entry in <i>requestQueue</i> . $updRequestQueue.add(ctr_{\text{tmp}}, (\text{attachment}, attachment'), r, pid')$ $msglist' \leftarrow msglist$ for all $txID \in \{txID_1, \dots, txID_l\}$ do: $\left\{ \begin{array}{l} \text{Check that all involved honest parties agreed on the tx} \\ \text{Get tx content from } \mathcal{F}_{\text{storage}}^{\mathbf{c}} \end{array} \right.$ send (GetContent, tx, <i>txID</i>) to ($_ , sid_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage) wait for (GetContent, tx, <i>tx</i>) parse (<i>initiator, signee₁, ..., signee_n</i>) from <i>tx</i> if $\neg \text{checkAgreements}^a(tx, requestQueue, CorruptionSet)$: reply (Update, $\emptyset, \emptyset, \epsilon$) $\left\{ \begin{array}{l} \text{Processing aborted} \\ \text{Start preparing the update} \end{array} \right.$ $id = msglist' + 1$ Let $r = \max\{r_{i_1}, \dots, r_{i_n}\}$ with $(txID, tx_{i_1}, pid_{i_1}, r_{i_1}), \dots, (txID, tx_{i_n}, pid_{i_n}, r_{i_n})$ in <i>reqQueue</i> $msglist'.add(id, round, tx, [tx, tx, true], r, initiator)$ if <i>isDoubleSpend</i> (<i>tx</i>): reply (Update, $\emptyset, \emptyset, \epsilon$) $\left\{ \begin{array}{l} \text{Prevention of double-spending} \\ \text{Remove "processed" tx from requestQueue} \end{array} \right.$ for all <i>txID</i> in <i>msg</i> do: $\left\{ \begin{array}{l} \text{Remove "processed" tx from requestQueue} \end{array} \right.$ for all ($ctr_{\text{temp}}, msg', round', pid'$) identified by <i>txID</i> is in <i>requestQueue</i> do: $updRequestQueue.add(ctr_{\text{temp}}, msg', round', pid')$ for all $txID \in \{txID_1, \dots, txID_l\}$ do: $ctr \leftarrow ctr + 1$ $\left\{ \begin{array}{l} \text{In the order of the input msg} \end{array} \right.$ Let ($ctr_{\text{temp}}, msg', round', initiator'$) be the first message in <i>requestQueue</i> identified by <i>txID</i> such that <i>initiator'</i> is the initiator in <i>msg'</i> $msgListAppend.add(ctr, round, tx, msg', sRound, initiator')$ if there is a transaction in $msglist \cup msgListAppend$ such parts of its transaction inputs or the transaction subgraph below the transaction is not in $msglist \cup msgListAppend$: reply (Update, $\emptyset, \emptyset, \epsilon$) $\left\{ \begin{array}{l} \text{Processing aborted} \end{array} \right.$ reply (Update, $msgListAppend, updRequestQueue, \epsilon$) $\left\{ \begin{array}{l} \text{Return list extension and updated queue} \end{array} \right.$

^acheckAgreements outputs true if for all honest signees or honest initiator, there exists a submit request of *tx* in *requestQueue*

Figure 41: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s update functionality $\mathcal{F}_{\text{update}}^{\mathbf{c}}$ (Pt. 1)

Description of $M_{\text{update}}^{\mathbf{c}}$ (cont.):

Main:

```

recv (Update, [txExchange], internalState) from ( $\_$ , sidcur,  $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$  : client):
  updRequestQueue  $\leftarrow$   $\emptyset$ , msgListAppend  $\leftarrow$   $\emptyset$ 
  ctr  $\leftarrow$  max $\{i \mid (i, \_ \_ \_ \_) \in \text{msgList}\}$ 
  for all ( $\text{ctr}_{\text{temp}}$ , [tx, txID, pidrecv], r, pid)  $\in$  requestQueue do:
    ctr  $\leftarrow$  ctr + 1
    msgListAppend.add(ctr, round, tx, [tx, txID, pidrecv], r, pid)
    updRequestQueue.add( $\text{ctr}_{\text{temp}}$ , [tx, txID, pidrecv], r, pid)
  reply (Update, msgListAppend, updRequestQueue,  $\epsilon$ )

recv (Update, [Validate, txID, pid, txID1, ..., txIDl, id1a, ..., idoa], internalState) from I/O:
   $\{\mathcal{A}$  may access validation information of tx for every party involved in the tx
  send getCurrentKnowledge to (pidcur, sidcur,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage)
  wait for (getCurrentKnowledge, txGraphpidcur, attachmentspidcur)
  send (GetContent, tx, txID) to (pidcur, sidcur,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage)
  wait for (GetContent, tx, tx)
  if {txID1, ..., txIDl} not in txGraphpidcur  $\vee$  {id1a, ..., idoa} not in attachmentspidcur:
  reply (Update,  $\epsilon$ ,  $\epsilon$ ,  $\epsilon$ )
  Remove all entries from txGraphpidcur that are not in {txID1, ..., txIDl}.
  Remove all entries from attachmentspidcur that are not in {id1a, ..., idoa}.
  leakage  $\leftarrow$  executeValidation(tx, txGraphpidcur, attachmentspidcur)  $\wedge$   $\neg$ isDoubleSpend(tx)
  reply (Update,  $\epsilon$ ,  $\epsilon$ , leakage)

recv (Update, (GetID, type, msg), internalState) from I/O:
  send (GetID, type, msg) to ( $\_$ , sidcur,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage)
  wait for (GetID, type, id)
  leakage  $\leftarrow$  id
  reply (Update,  $\epsilon$ ,  $\epsilon$ , leakage)

```

Annotations for Figure 42:

- {Record shared/pushed transactions in msglist
- {If tx exchange is done, we add it to the message list
- {Send update to $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$
- { \mathcal{A} may access validation information of tx for every party involved in the tx
- {Get current knowledge of pid_{cur}
- {Get transaction details
- {Check valid context
- {Validation declined
- {Validity check for tx in the provided context
- {Return the validity of the transaction to \mathcal{A}
- { \mathcal{A} may query $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ for ids
- {Request Id at $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$
- {Return the ID of the requested object to \mathcal{A}

Figure 42: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s update functionality $\mathcal{F}_{\text{update}}^{\mathbf{c}}$ (Pt. 2)

Procedures and Functions:	
function executeBasicChecks :	
if \exists a tx push msg in requestQueue:	{Valid transactions pushed between entities need to be in the state before further updates}
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
reqQueue $\leftarrow \emptyset$	
for all [(type, msg'), sRound, pid'] \in requestQueue do :	{Generate a copy of requestQueue including txID's}
parse tx or attachment from msg' and store result in content	
send (GetID, type, content) to (pid _{cur} , sid _{cur} , $\mathcal{F}_{storage}^c$: storage)	{Get IDs from $\mathcal{F}_{storage}^c$ }
wait for (GetID, id); reqQueue.add([id, type, msg'], sRound, pid')	
if msg' \neq [(txID ₁ , ..., txID _l), [(id ₁ , pid ₁), ..., (id ₁ , pid _l)], (txID _{i₁} , tx _{i₁} , pid _{i₁} , r _{i₁}), ..., (txID _{i_h} , tx _{i_h} , pid _{i_h} , r _{i_h}), (id ₁ ^a , attachment ₁ , pid ₁ ^a), ..., (id _m ^a , attachment _m , pid _m ^a)]:	{Check message format of transactions and attachments "from" corrupted parties}
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
if $\exists i, j \in [l], i \neq j$, s.t. txID _i , txID _j in (txID ₁ , ..., txID _l), \wedge txID _i = txID _j :	{No txID is allowed twice in msglist}
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
for all (txID, tx, pid', r) from [d ₁ , ..., d _h] in msg do :	{Register txIDs for \mathcal{A} 's inputs, d _j of form (txID _j , tx _j , pid _j , r _j)}
send (SetCorrID, tx, txID, tx) to (pid _{cur} , sid _{cur} , $\mathcal{F}_{storage}^c$: storage)	{ \mathcal{A} defines the txID}
wait for (SetCorrID, accepted)	
if \neq accepted: reply (Update, \emptyset , \emptyset , ϵ)	{Abort processing}
for all (txID, tx, pid', r) from [d ₁ , ..., d _h] in msg do :	{Check correct format, identities, roles, d _j of form (txID _j , tx _j , pid _j , r _j)}
if tx does not match \oplus : reply (Update, \emptyset , \emptyset , ϵ)	{Check correct format}
if \neg [initiator, signee ₁ , ..., signee _m are clients (role = client, pid prefixed by client) \wedge notary, formerNotary are notaries (role = notary, pid prefixed by notary) \wedge \forall pid \in {initiator, signee ₁ , ..., signee _m , notary, formerNotary} : pid \in identities]:	
reply (Update, \emptyset , \emptyset , ϵ)	{Check correct identities and roles}
Let Attachments be the set of all available attachments & Tx the set of all available transactions.	
if isDoubleSpend(tx) \vee \neg executeValidation(tx, Tx, Attachments):	{No double-spending and only valid tx allowed}
reply (Update, \emptyset , \emptyset , ϵ)	
for all (id ^a , attachment, pid') in (d ₁ , ..., d _m) in msg do :	{Register attachment IDs for \mathcal{A} 's inputs, d _j are of form (id _j ^a , attachment _j , pid _j ^a)}
send (SetCorrID, attachment, id ^a , attachment) to (pid _{cur} , sid _{cur} , $\mathcal{F}_{storage}^c$: storage)	
wait for (SetCorrID, accepted)	
if accepted = false:	
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
for all (id', pid') \in {(id ₁ , pid ₁), ..., (id ₁ , pid _l)} do :	
send (GetContent, attachment, id') to ($_$, sid _{cur} , $\mathcal{F}_{storage}^c$: storage)	{Get attachment content from $\mathcal{F}_{storage}^c$ }
wait for (GetContent, attachment, attachment')	
if ($_$, (attachment', attachment'), $_$, pid') \notin requestQueue:	
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
if \exists txID \in {txID ₁ , ..., txID _l } s.t. txID is not in reqQueue or {txID _{i₁} , ..., txID _{i_h} }:	{Check that input data exists}
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
for all (id ^a , attachment, pid') \in {(id ₁ ^a , attachment ₁ , pid ₁ ^a), ..., (id _m ^a , attachment _m , pid _m ^a)} do :	
if \neg validateAttachment(attachment):	{Reject invalid attachments from \mathcal{A} }
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}
ctr \leftarrow ctr + 1	
msgListAppend.add(ctr, round, tx, attachment, round, pid')	{Add attachments from \mathcal{A} }
if there is a attachment in requestQueue such that it is not part of updRequestQueue:	
reply (Update, \emptyset , \emptyset , ϵ)	{Processing aborted}

Figure 43: \mathcal{F}_{ledger}^c 's update functionality \mathcal{F}_{update}^c (Pt. 3)

Description of the subroutine $\mathcal{F}_{\text{storage}}^{\mathbf{c}} = (\text{storage})$:

Participating roles: {storage} Corruption model: <i>incorruptible</i> Protocol parameters: - $\eta \in \mathbb{N}$	<i>{The security parameter, defining the length of a “hash-value”}</i>
--	--

Description of $M_{\text{storage}}^{\mathbf{c}}$:

Implemented role(s): {storage} Internal state: - labels $\subset \{0, 1\}^* \times \{\text{tx}, \text{attachment}\} \times \{0, 1\}^*$. - hasAccess{tx, attachment} $\times \{0, 1\}^*$	<i>{Storage for IDs of transactions and attachments {Bookkeeping for data directly known by \mathcal{A}}</i>
CheckID (<i>pid, sid, role</i>): Accept all messages with the same <i>sid</i> .	
Main: rcv (GetID, <i>type, msg</i>) from I/O s.t. <i>type</i> $\in \{\text{tx}, \text{attachment}\}$: if $\exists (id, _ , msg) \in \text{labels}$ for some $id \in \{0, 1\}^*$: reply (GetID, <i>type, id</i>) else: send responsively (SetID, <i>type</i>) to NET wait for (SetID, <i>type, id</i>) while [$ id \neq \eta$] $\vee [\exists (id, _ , _) \in \text{labels}]$ do send responsively (SetID, <i>type</i>) to NET wait for (SetID, <i>type, id</i>) reply (GetID, <i>type, id</i>)	<i>{Request for IDs {If ID exists, return it {Ensure correct length of the ID and its uniqueness {Return newly generated ID</i>
rcv (SetCorrID, <i>type, idProposal, msg</i>) from NET s.t. <i>type</i> $\in \{\text{tx}, \text{attachment}\}$: if $[\exists (idProposal, type', msg') \in \text{labels}, \text{s.t. } msg' \neq msg] \vee idProposal \neq \eta$: reply (SetCorrID, false) else: labels.add(<i>type, idProposal, msg</i>) reply (SetCorrID, true)	<i>{\mathcal{A} may register IDs {If ID is already used for something else or the length of the ID is not η, reject it {Otherwise: store and return approved ID</i>
rcv (getTxGraph, <i>internalState, mode</i>) from I/O: <i>msglistID</i> $\leftarrow \emptyset$ if <i>mode</i> = incBuffer: <i>transactions</i> $\leftarrow \{\text{tx} (_ , _ , \text{tx}, _ , _) \in \text{msglist} \vee (_ , \text{tx}, _ , _) \in \text{requestQueue} \wedge \text{tx is of form } [\text{tx}, (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal})]$ else: <i>transactions</i> $\leftarrow \{\text{tx} (_ , _ , \text{tx}, _ , _) \in \text{msglist} \wedge \text{tx is of form } [\text{tx}, (\text{initiator}, [\text{signee}_1, \dots, \text{signee}_m], \text{notary}, \text{formerNotary}, \text{proposal})]$ for all <i>tx</i> \in <i>transactions</i> do: Let <i>txID</i> s.t. (<i>txID, tx, msg</i>) \in labels <i>txAttachments</i> $\leftarrow \emptyset$ parse (id_1^a, \dots, id_l^a) from <i>msg</i> for all $id^a \in \{id_1^a, \dots, id_l^a\}$ do: Let <i>attachment</i> s.t. ($id^a, \text{attachment}, \text{attachment}$) \in labels <i>txAttachments</i> .add($id^a, \text{attachment}$) <i>msglistID</i> .add(<i>txID, msg, txAttachments</i>) <i>txGraph</i> \leftarrow buildTxGraph ^a (<i>msglistID</i>) reply (getTxGraph, <i>txGraph</i>)	<i>{Generate a tx graph based on msglist, with or without tx from requestQueue {In incBuffer mode, include buffer to graph generation {Only “finalized” messages are in the graph {Extract tx IDs {Extract used attachments {Connect attachments to transactions {Extract attachment content {Generate a tx graph with all necessary content {Return full (notarised) tx graph</i>
^a buildTxGraph generates a (most likely disconnected) directed graph over all entries in <i>msglistID</i> . Transactions are identified by <i>txID</i> and arcs are from a transaction that consumes a tx to its input, i.e., (<i>txID'</i> , <i>txID</i>), s.t. (<i>txID'</i> , <i>outputID</i>) is a consumed input in (<i>txID</i>). The function mainly structures and cleans up “metadata” from <i>msglist</i> and adds components from <i>requestQueue</i> if necessary.	

Figure 44: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s shared subroutine storage $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ (Pt. 1)

Main:

```

recv (GetContent, type, id) from I/O s.t. type  $\in$  {tx, attachment}: {Request for content}
  if  $\exists$ (id, type, content)  $\in$  labels: {Check whether requested object exists}
    reply (GetContent, type, content)
  else: {Otherwise: return  $\epsilon$ }
    reply (GetContent, type,  $\epsilon$ )

recv getKnowledge from I/O: { $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$  subroutine may ask for the possible knowledge of  $\text{pid}_{\text{cur}}$ }
  transactions  $\leftarrow$   $\emptyset$ , attachments  $\leftarrow$   $\emptyset$ 
  Execute getTxGraph call above in normal mode and store output in txGraph.
  for all ( $\_$ ,  $\_$ , tx, msg,  $\_$ ,  $\_$ )  $\in$  msglist do: {Collect knowledge from msglist}
    parse (initiator, signee1, ..., signeem, notary, formerNotary) from msg
    Let id s.t. (id, tx, msg)  $\in$  labels {Get txID}
    if  $\text{pid}_{\text{cur}} \in$  {initiator, signee1, ..., signeem, notary, formerNotary}:
      txSubGraph  $\leftarrow$  getConnectedSubGrapha(msg, txGraph) {Extract  $\text{pid}_{\text{cur}}$ 's tx graph from txGraph}
      for all (txID, tx, attachmentstx) in txSubGraph, s.t. txID is an input to msg do:
        transactions.add(txID, tx); attachments.add(attachmentstx) {Leak txID, content, and connected attachments}
      Execute getTxGraph call above in incBuffer mode and store output in txGraph
      for all ( $\_$ , msg,  $\_$ ,  $\text{pid}_{\text{cur}}$ )  $\in$  requestQueue do: {Collect knowledge from requestQueue}
        parse (initiator, signee1, ..., signeem, notary, formerNotary) from msg
        Let id s.t. (id, tx, msg)  $\in$  labels {Get txID}
        if  $\text{pid}_{\text{cur}} =$  initiator: {tx initiator has access to the full subgraph "below" a transaction}
          txSubGraph  $\leftarrow$  getConnectedSubGrapha(msg, txGraph)
          for all (txID, tx, attachmentstx) in txSubGraph s.t. txID is an input to msg do:
            transactions.add(txID, tx); attachments.add(attachmentstx)
          else if  $\text{pid}_{\text{cur}} \in$  {signee1, ..., signeem, notary, formerNotary}:
            {Signees and notaries have access to the full subgraph "below" a transaction if the initiator has the right to dispatch the data}
            transactions.add(id, msg) {Add tx to  $\text{pid}_{\text{cur}}$ 's knowledge}
          if  $\exists$ ( $\_$ , msg,  $\_$ , initiator)  $\in$  requestQueue  $\vee$   $\exists$ ( $\_$ ,  $\_$ , tx, msg,  $\_$ , initiator)  $\in$  msglist:
            txSubGraph  $\leftarrow$  getConnectedSubGrapha(msg, txGraph)
            for all (txID, tx, attachmentstx) in txSubGraph, s.t. txID is an input to msg do:
              transactions.add(txID, tx); attachments.add(attachmentstx)
            Add all transaction/attachments from hasAccess and all transactions leaked (according to transcript) to transactions or attachments.
          for all (ctrtemp, attachment, r, pid)  $\in$  msglist do:
            send (GetID, attachment, attachment) to ( $\text{pid}_{\text{cur}}$ ,  $\text{sid}_{\text{cur}}$ ,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ ) {Request attachment ID}
            wait for (GetID, attachmentid) { $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  returns an ID or false}
            attachments.add(id, attachment)
          reply (getKnowledge, transactions, attachments) {Return knowledge}

```

^{*a*}getConnectedSubGraph(*msg*, *txGraph*) outputs the (maximal) connected subgraph of *txGraph* such that *msg*'s outputs are all arcs of this subgraph ("maximal" means \nexists a connected subgraph *txGraph* containing *msg*'s output and that is a superset of getConnectedSubGraph(*msg*, *txGraph*)). The data format is the same as in *txGraph*.

Figure 45: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s shared subroutine storage $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ (Pt. 2)

Description of $\mathcal{M}_{\text{storage}}^{\mathbf{c}}$ (cont.):

Main:

```

recv (hasAccess, type, id, mode) from I/O: {Check whether  $\mathcal{A}$  knows this object}
  if mode  $\neq$  subgraph:
    if (type, id)  $\in$  hasAccess  $\vee$  (type, id) was leaked according to transcript:
      reply (hasAccess, true)
    else:
      reply (hasAccess, false)
  else if mode = subgraph  $\wedge$  type = tx:
    if (type, id)  $\in$  hasAccess  $\vee$  (type, id) was leaked according to transcript and all dependent tx/attachments in the id's
    subgraph are in hasAccess or leaked:
      reply (hasAccess, true)
    else:
      reply (hasAccess, false)
  else:
    reply (hasAccess, false)

```

Figure 46: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s shared subroutine storage $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ (Pt. 3)

Description of the subroutine $\mathcal{F}_{\text{updRnd}}^{\mathbf{c}} = (\text{updRnd})$:

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

Description of $\mathcal{M}_{\text{updRnd}}^{\mathbf{c}}$:

Implemented role(s): {updRnd}
Subroutines: $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage
CheckID(pid, sid, role):
 Accept all messages with the same sid.

Main:

```

recv (UpdateRound, msg, internalState) from I/O: {See Figure 5 for definition of internalState and the local vari-
ables it includes}
  send (getTxGraph, internalState,  $\epsilon$ ) to (pidcur, sidcur,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage) {Get tx graph from  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ }
  wait for (getTxGraph, txGraph)
  for all (ctrtemp, tx', submissionRound, pid')  $\in$  requestQueue do:
    send (GetID, tx, tx') to (pidcur, sidcur,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage) {Get tx ID}
    wait for (GetID, txID')
    attachments'  $\leftarrow \epsilon$ 
    parse (id1a, ..., idla) from tx' {Extract used attachments}
    for all ida  $\in$  {id1a, ..., idla} do: {Connect attachments to transactions}
      send (GetContent, attachment, ida) to (pidcur, sidcur,  $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$  : storage) {Get attachments}
      wait for (GetContent, attachment, attachment)
      attachments'.add(ida, attachment)
  if pid'  $\notin$  CorruptionSet: {Liveness can only be guaranteed for uncorrupted participants}
    txGraph'  $\leftarrow$  txGraph
    txGraph'.addToTxGraph(txID', tx', attachments') {addToTxGraph adds the tx extracted from tx
to txGraph'}
    txSubGraph  $\leftarrow$  getConnectedSubGraph(tx, txGraph') {See Footnote a in Figure 45 on Page 77 for the
explanation of getConnectedSubGraph.}
    lastAgreement  $\leftarrow$  findLastAgreementa(tx, requestQueue) {Last round of the latest agree-
ment to tx}
    if participantsAgreedb(tx)  $\wedge$  lastAgreement + (3 + 4 · |txSubGraph|) ·  $\delta$  > round:
      {If the initiator and all signees agree on a transaction, it should be part of the
state after (1 + |txSubGraph|) ·  $\delta$  rounds}
      reply (UpdateRound, false,  $\epsilon$ )
    reply (UpdateRound, true,  $\epsilon$ )

```

^afindLastAgreement(tx, requestQueue) outputs the round of the last agreement to tx in requestQueue.
^bparticipantsAgreed(tx) outputs true if for all uncorrupted signees and the initiator, there exists a submit request for the tx in requestQueue. If all these parties are corrupted, the output is always true, otherwise false.

Figure 47: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s round update functionality $\mathcal{F}_{\text{updRnd}}^{\mathbf{c}}$

Description of the subroutine $\mathcal{F}_{\text{leak}}^{\mathbf{c}} = (\text{leak})$:

Participating roles: {leak}
Corruption model: *incorruptible*

Description of $M_{\text{leak}}^{\mathbf{c}}$:

Implemented role(s): {leak}
CheckID(*pid, sid, role*):
 Accept all messages with the same *sid*.
Subroutines: $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage
Main:
recv (Corrupt, *pid, internalState*) **from** I/O: {See Figure 5 for definition of *internalState* and the local variables
 {it includes
send getKnowledge **to** (*pid_{cur}, sid_{cur}, $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$* : storage) {Query *pid_{cur}*'s "state" at $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$
wait for (getKnowledge, *transaction, attachments*)
leakage \leftarrow (*transaction, attachments*)
reply (Corrupt, *leakage*) { \mathcal{A} receives all transactions from *msglist, requestQueue, and readQueue, the
 newly corrupted party sent, received, or is involved in*

Figure 48: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s leakage subroutine $\mathcal{F}_{\text{leak}}^{\mathbf{c}}$

Description of the protocol $\mathcal{F}_{\text{init}}^{\mathbf{c}} = (\text{Init})$:

Participating roles: {init}
Corruption model: *incorruptible*
Protocol parameters:
 - *networkmap* \subset $\{0, 1\}^* \times \{\text{client, notary}\}$ {Map of identities existing in the network containing tuples of
 (*identifier, type*). We expect that the role prefixes the *pid* in *networkmap*

Description of $M_{\text{init}}^{\mathbf{c}}$:

Implemented role(s): {init}
CheckID(*pid, sid, role*):
 Accept all messages with the same *sid*.
Main:
recv Init **from** I/O or NET: {We allow \mathcal{A} to query $\mathcal{F}_{\text{init}}^{\mathbf{c}}$ as well
identities \leftarrow \emptyset , *msglist* \leftarrow \emptyset , *ctr* \leftarrow 0
for all (*pid, protocolRole*) \in *networkmap* **do:**
identities.add(pid, 0) { $\mathcal{F}_{\text{init}}$ provides the network map to requestors
msglist.add(ctr, 0, meta, (pid, protocolRole), \perp , \perp)
ctr \leftarrow *ctr* + 1
reply (Init, *identities, $\epsilon, \epsilon, \epsilon$*)

Figure 49: $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$'s initialization functionality $\mathcal{F}_{\text{init}}^{\mathbf{c}}$