A Security Framework for Distributed Ledgers

Christoph Egger[1], Mike Graf[2], Ralf Küsters[2], Daniel Rausch[2], Viktoria Ronge[1], and Dominique Schröder[1]

[1] Friedrich-Alexander University Erlangen-Nürnberg, Erlangen, Germany
Email: {christoph.egger,viktoria.ronge,dominique.schroeder}@fau.de
[2] University of Stuttgart, Stuttgart, Germany
Email:{mike.graf,ralf.kuesters,daniel.rausch}@sec.uni-stuttgart.de

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

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

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

1 Introduction

In the past few years, researchers made significant progress in formalizing and analyzing the security of blockchain protocols [2,3,15,20,24]. Initially based on trace properties [14,19,32], blockchain security research has moved to simulation-based security which leverages modularity and strong security guarantees offered by universal composability (UC) frameworks [8,10,28]. Several ideal blockchain functionalities have been proposed – most notably the functionality by Badertscher et al. [3], called $\mathcal{G}_{\text{ledger}}$, and its privacy-preserving derivative $\mathcal{G}_{\text{PL}}$ [24]. They have successfully been applied to prove the security of various, partly newly designed blockchains (see, e.g., [2,3,24]). 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 [6], OmniLedger [27], and Canton [37], 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. 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 [13,18,21,22,33].

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.

1.1 The Ideal Ledger Functionality $\mathcal{F}_{\text{ledger}}$

As explained above, the main goal of our ideal distributed ledger functionality $\mathcal{F}_{\text{ledger}}$ is to be able to capture and analyze the security of essentially arbitrary distributed ledger protocols, including classical blockchains, as well as higher-level protocols built thereupon. The design of and features offered by $\mathcal{F}_{\text{ledger}}$ follow this main goal:

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, 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 [29]), 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 important Corda system.

1.2 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 $\mathcal{G}_{\text{ledger}}$ [3] realizes a suitable instantiation of $\mathcal{F}_{\text{ledger}}$, which precisely captures the security properties provided by $\mathcal{G}_{\text{ledger}}$, and demonstrate that this result also holds for its privacy-preserving variant $\mathcal{G}_{\text{PL}}$ [24]. Hence, any realization of $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{PL}}$ also realizes $\mathcal{F}_{\text{ledger}}$, which covers all published UC analyses of blockchains, including Bitcoin [3], Ouroboros Genesis [2], and Ouroboros Crypsinous [24].

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, $\mathcal{F}_{\text{ledger}}$ only requires

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 $\mathcal{G}_{\text{PL}}$ and $\mathcal{F}_{\text{ledger}}$ are quite different as well.

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

Altogether, these results do not only show that $\mathcal{F}_{\text{ledger}}$ can cover the blockchain literature, but also helps to unify existing models and results.

1.3 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* [5, 6, 34, 35]. 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, including Hewlett Packard Enterprise, Intel, Microsoft, and also by NASDAQ [13, 18, 21, 33]. 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 [23, 36]. Also SWIFT, which is used by literally every bank in the world, partners with R3 [12]. Her Majesty's Land Registry uses Corda [22] and several consulting groups identify Corda as the most prominent distributed ledger technology and attest Corda to bring common standards to distributed ledgers [1, 4, 31].

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, including blockchains. 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*. In partial consistency, which we formalize for the first time in this work, parties see only part of the state but where 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 properties via an instantiation of $\mathcal{F}_{\text{ledger}}$. We then analyze the security of Corda, i.e., we show that it 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. Furthermore, while the official specification of Corda requires security only under the assumption that *all* of the 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.

1.4 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-blockchains distributed ledgers. As demonstrated in this work, it covers both traditional blockchains and non-blockchain distributed ledgers. Our functionality offers high flexibility to support a wide variety of different implementations with various security properties while simultaneously exposing a simple and implementation independent interface to higher-level protocols. Thereby $\mathcal{F}_{\text{ledger}}$ not only generalizes but also unifies the landscape of existing functionalities for blockchains.

- We show in Section 3 that our functionality subsumes $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{PL}}$. In particular, this allows for directly transferring all published results on the modular security of blockchains, such as Bitcoin and the Ouroboros family, to our functionality. We further discuss that other published ideal blockchain functionalities, which have so far only been used to model setup assumptions, are also captured by $\mathcal{F}_{\text{ledger}}$.
- In Section 4, we provide the first formal model and security analysis of a non-blockchain distributed ledger, Corda. As part of this, we develop and formalize the novel security notion of partial consistency. Our analysis precisely defines the level of privacy achieved by Corda where, on the one hand, we find that partly weaker security assumptions are necessary than officially required, and, on the other hand, we find some subtle information leaks. 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. As already mentioned in the introduction, $\mathcal{F}_{\text{ledger}}$ further employs several core concepts to keep this interface highly general and in particular independent of a specific realization. In what follows, we explain $\mathcal{F}_{\text{ledger}}$ in detail.

Description of $\mathcal{F}_{\text{ledger}}$: Our functionality $\mathcal{F}_{\text{ledger}}$ is defined in the iUC framework [8], which is a recently proposed, expressive, and convenient general framework for universal composability similar in spirit to Canetti's UC model [10]. 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 *static* structure of the functionality.[3] 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 in-built 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 parameterized 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 parameters/subroutines in more detail.

During a run of $\mathcal{F}_{\text{ledger}}$, there can be multiple instances of the ideal functionality, each of which models a single session of a distributed ledger that can be uniquely addressed by a session ID (SID). Each of these instances/sessions handles an unbounded number of parties that can read and write from 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,

---

[3] We choose machines, instead of just algorithms, as parameters since they are more flexible in terms of storing and sharing state, and since they can interact with the adversary.
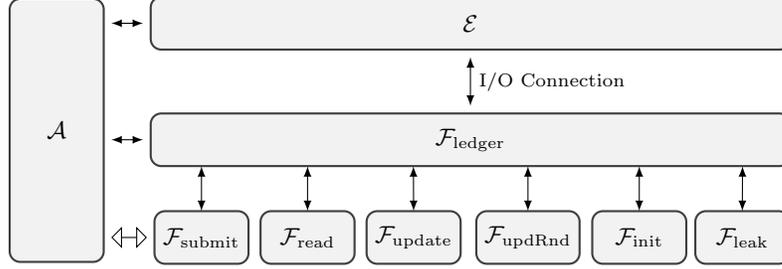
Fig. 1. Overview of $\mathcal{F}_{\text{ledger}}$ and its subroutines. The open headed arrow indicates that $\mathcal{A}$ also connects to all of $\mathcal{F}_{\text{ledger}}$'s subroutines

and then reading from that state (cf. Figure 2 for a formal definition of these operations). Dishonest parties and further details are discussed afterward.

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 $\mathcal{F}_{\text{submit}}$,[4] 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 submitting a time stamp. This meta transaction can be useful, e.g., to store block boundaries of a blockchain in those cases where this should be captured as an explicit property of a realization. Similar to ideal functionalities for blockchains, the global transaction list of $\mathcal{F}_{\text{ledger}}$ is determined and updated by the adversary, subject to restrictions that ensure expected security properties.

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

---

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

```
Main (excerpt):

recv (Submit, msg) from I/O:                                                    {Submission request from an honest identity
    send (Submit, msg, internalState) to (pid_cur, sid_cur, F_submit : submit)
    wait for (Submit, response, leakage) s.t. response ∈ {true, false}
    if response = true:
        reqCtr ← reqCtr + 1
        requestQueue.add(reqCtr, round, pid_cur, msg)
    send (Submit, response, leakage) to NET

recv (Update, msg) from NET:                                                    {Update request triggered by the adversary.
    send (Update, msg, internalState) to (ε, sid_cur, F_update : update)
    wait for (Update, listAdd, updRequestQueue, leakage)
            s.t. listAdd ⊂ N × {round} × {tx, meta} × {0, 1}* × N × {0, 1}*
    max ← max{i|(i, _, _, _, _, _) ∈ msglist}
    check ← listAdd ≠ ∅ ∨ updRequestQueue ≠ ∅
    for i = max + 1 to max + |listAdd| do:
        if ∄₁(i, _, _, _, _, _) ∈ listAdd:
            check ← false
        if ∃(i, _, meta, _, a, b) ∈ listAdd ∧ (a ≠ ⊥ ∨ b ≠ ⊥):
            check ← false
    if check:
        msglist.add(listAdd)
        for all item ∈ updRequestQueue do:
            requestQueue.remove(item)
    reply (Update, check, leakage)

recv (Read, msg) from I/O:                                                      {Read request from an honest identity
    send (InitRead, msg, internalState) to (pid_cur, sid_cur, F_read : read)
    wait for (InitRead, local, leakage) s.t. local ∈ {true, false}
    if local:
        send responsively (InitRead, leakage) to NET     (⋆)
        wait for (InitRead, suggestedOutput)
        send (FinishRead, msg, suggestedOutput, internalState)
                to(pid_cur, sid_cur, F_read : read)
        wait for (FinishRead, output, leakage')
        if output = ⊥:
            Go back to (⋆) and repeat the request (local variables suggestedOutput, output, and leakage' are
            cleared)
        send responsively (FinishRead, leakage') to NET
        wait for ack
        reply (Read, output)
    else:
        readCtr ← readCtr + 1; readQueue.add(pid, readCtr, round, msg)
        send (Read, readCtr, leakage) to NET

recv (DeliverRead, readCtr, suggestedOutput) from NET s.t.
    (pid, readCtr, r, msg) ∈ readQueue:                                         {Deliver network read
    send (FinishRead, msg, suggestedOutput, internalState)
            to(pid_cur, sid_cur, F_read : read)
    wait for (FinishRead, output, leakage')
    if output ≠ ⊥:
        send responsively (FinishRead, readCtr, leakage') to NET
        wait for ack
        readQueue.remove(pid, readCtr, r, msg)
        send (Read, output) to (pid, sid_cur, I/O)
    else:
        send nack to NET
```

Fig. 2. Excerpt of F_{ledger}'s specification handling submit, read, and update operations. See Figure 5 to 7 in Appendix B for the full specification. pid_{cur} identifies the current party and sid_{cur} the current session of the ledger. round models the current time.

Upon receiving the response from F_{update}, F_{ledger} ensures that appending the proposed extension to msglist still results in an ordered list of transactions. If this is the case, then F_{ledger} applies the proposed changes to both lists. In any case, F_{ledger} sends the leakage from F_{update} as well as a boolean indicating whether any changes have been applied to the adversary.

The functionality F_{ledger}, by default, only guarantees 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 F_{update}. For example, F_{update} can be used to enforce the security properties of *double spending protection* and *no creation*. On a technical level, such an instantiation of F_{update} would be defined in such a way that it expects the incoming update requests from the adversary to contain a proposed extension of the global state that does not cause double-spending and does not contain transactions for honest parties that were not previously submitted. If the incoming request does not meet this criterion, F_{update} denies the update request by returning an empty extension to F_{ledger}. Otherwise, the changes proposed by the adversary are returned to and then consequently applied by F_{ledger}.

We note that the default guarantee provided by $\mathcal{F}_{\mathrm{ledger}}$ (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 $\mathcal{F}_{\mathrm{ledger}}$, namely via a suitable instantiation of $\mathcal{F}_{\mathrm{read}}$ (see below).

Reading from the global state. A higher-level protocol can instruct a party of $\mathcal{F}_{\mathrm{ledger}}$ 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. In contrast, a non-local read would instead model a thin client that first has to retrieve the data contained in the ledger via the network, and hence, cannot make any guarantees on when (and if at all) the read request finishes. Transferring the property of local reads from the realization also to the idealization is very convenient for higher-level protocols built on top of the idealization: such higher-level protocols get the guarantee of an immediate response to all read requests, instead of having to manually deal with arbitrarily delayed and dropped responses as well as intermediate state changes.

More specifically, when $\mathcal{F}_{\mathrm{ledger}}$ receives a read request, the subroutine $\mathcal{F}_{\mathrm{read}}$ is used in $\mathcal{F}_{\mathrm{ledger}}$ 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. $\mathcal{F}_{\mathrm{ledger}}$ provides the adversary with the responses of $\mathcal{F}_{\mathrm{read}}$; 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 $\mathcal{F}_{\mathrm{read}}$, 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 $\mathcal{F}_{\mathrm{read}}$. Finally, the resulting output is forwarded by $\mathcal{F}_{\mathrm{ledger}}$ 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, i.e., the adversary has to decide upon immediately and return an output for the read request and without modifying the global state beforehand (in Figure 2 the operation "send responsively" indicates such network messages with immediate responses). As shown in [7], this mechanism can, in principle, also be added to Canetti's UC model. Non-local read requests are split into two separate activations of $\mathcal{F}_{\mathrm{read}}$, 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 $\mathcal{F}_{\mathrm{read}}$ appropriately. For example, $\mathcal{F}_{\mathrm{read}}$ can be used to enforce the traditional notion of *consistency*, where all honest parties read (prefixes of) the same global state (cf. Appendix C for an example specification). In this case, the adversary's response would be interpreted as a pointer to some transaction in the global state. The resulting output generated by $\mathcal{F}_{\mathrm{read}}$ would then be the prefix of the global state up to and including the transaction that the pointer indicates. $\mathcal{F}_{\mathrm{read}}$ 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, updating the global state, and reading from that state, we now explain several further details and features of $\mathcal{F}_{\mathrm{ledger}}$.

Initialization of $\mathcal{F}_{\mathrm{ledger}}$***.*** 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, such as an initial stake distribution or an initial set of PIDs that are allowed to participate in the ledger. To allow for capturing such initially shared state $\mathcal{F}_{\mathrm{ledger}}$ includes an ideal initialization subroutine $\mathcal{F}_{\mathrm{init}}$ that can be defined by a protocol designer and is used to initialize the starting values of all internal variables of $\mathcal{F}_{\mathrm{ledger}}$, 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 zero 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*. If one wants to model, say, that a submitted transaction is included in the global transaction list after at most time t has passed, then $\mathcal{F}_{\text{updRnd}}$ forbids the adversary from advancing time if there are any transactions that have been in the buffer for time t and must therefore first be moved into the global transaction list. 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.

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

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

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

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 also 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 propose to use a more general interpretation of corruption in realizations instead: parties in a realization of $\mathcal{F}_{\text{ledger}}$ should consider themselves to be corrupted 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.

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 wide 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 he is under the direct control of the adversary but also if he 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.[5]

Public and private ledgers. Existing functionalities for blockchains have so far been modeled as so-called global functionalities using the GUC extension [9] 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.

---

[5] We note that this concept can easily be extended to capture multiple different levels of "broken" assumptions, e.g., to handle cases where the assumption for the security property of liveness is broken, but another assumption that guarantees the property of consistency still holds. The main requirement here is that the environment can check that real and ideal world are consistent in their "corruption levels", e.g., by allowing the environment to request the current corruption level. This extended mechanism can also be used with $\mathcal{F}_{\text{ledger}}$.

3 Covering Blockchain with $\mathcal{F}_{\text{ledger}}$

In this section, we demonstrate that $\mathcal{F}_{\text{ledger}}$ is able to capture traditional blockchains as a special case. Instead of illustrating this result via a specific case study in which we prove that, for example, Bitcoin realizes $\mathcal{F}_{\text{ledger}}$, we choose a more general approach: Firstly, we show that the so far most commonly used blockchain functionality $\mathcal{G}_{\text{ledger}}$ [3] realizes a suitable instantiation of $\mathcal{F}_{\text{ledger}}$, which precisely captures the security guarantees of $\mathcal{G}_{\text{ledger}}$, and demonstrate that this result also holds for its privacy-preserving variant $\mathcal{G}_{\text{PL}}$ [24]. Hence, any realization of $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{PL}}$ also realizes $\mathcal{F}_{\text{ledger}}$. This in fact covers all published UC analyses of blockchains, including Bitcoin [3], Ouroboros Genesis [2], and Ouroboros Crypsinous [24]. 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 C). $\mathcal{G}_{\text{ledger}}$ 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 $\in \mathbb{N}$ 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 $\mathcal{G}_{\text{clock}}$, 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*. In order to capture also non-blockchain distributed ledgers, like Corda, which neither uses blocks nor provides consistency, $\mathcal{F}_{\text{ledger}}$ instead only requires the existence of a global ordered list of transactions, which is a property that is common to distributed ledgers in general.
- 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, 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, $\mathcal{F}_{\mathrm{ledger}}$ 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.

- $\mathcal{G}_{\mathrm{ledger}}$ 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. $\mathcal{F}_{\mathrm{ledger}}$ instead allows the adversary to choose arbitrarily when time should advance. The single parameter $\mathcal{F}_{\mathrm{updRnd}}$, which has access to the full global state, can then directly enforce time dependent security properties without requiring synchronization with other parameters.
- $\mathcal{G}_{\mathrm{ledger}}$ uses algorithms as parameters, whereas $\mathcal{F}_{\mathrm{ledger}}$ uses subroutines. Our reasoning for choosing subroutines was explained in Footnote 3 on Page 4.

In summary, the main differences between $\mathcal{G}_{\mathrm{ledger}}$ and $\mathcal{F}_{\mathrm{ledger}}$ are due to *(i)* different levels of abstraction to higher-level protocols and *(ii)* the fact that $\mathcal{G}_{\mathrm{ledger}}$ is built specifically for traditional blockchains. Both of these aspects have to be addressed to show that $\mathcal{G}_{\mathrm{ledger}}$ is a realization of a suitable instantiation of $\mathcal{F}_{\mathrm{ledger}}$. To address *(i)*, we use a wrapper $\mathcal{W}_{\mathrm{ledger}}$ that we add on top of the I/O interface of $\mathcal{G}_{\mathrm{ledger}}$ and which handles messages from/to the environment. This wrapper not only translates the format of data output by $\mathcal{G}_{\mathrm{ledger}}$ to the format used by $\mathcal{F}_{\mathrm{ledger}}$ (e.g., from a blockchain to a list of transactions). It also handles the fact that $\mathcal{F}_{\mathrm{ledger}}$ 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 $\mathcal{G}_{\mathrm{clock}}$ even in the name of honest parties. That is, $\mathcal{A}$ acts as a scheduler capturing real world behavior 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 1). Issue *(ii)* is addressed via a suitable instantiation of the parameters of $\mathcal{F}_{\mathrm{ledger}}$ in order to capture the same (blockchain) properties provided by $\mathcal{G}_{\mathrm{ledger}}$ respectively the parameterized algorithms of $\mathcal{G}_{\mathrm{ledger}}$. This instantiation roughly works as follows, with full definitions and details provided in Appendix C:

- $\mathcal{F}_{\mathrm{init}}$ 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.
- $\mathcal{F}_{\mathrm{submit}}$ executes the validate algorithm to check validity of incoming transactions.
- $\mathcal{F}_{\mathrm{update}}$ 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 $\mathcal{F}_{\mathrm{ledger}}$ 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.
- $\mathcal{F}_{\mathrm{read}}$ 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 $\mathcal{F}_{\mathrm{read}}$ has to always use non-local reads: this is because a read operation in $\mathcal{G}_{\mathrm{ledger}}$ 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 $\mathcal{G}_{\mathrm{ledger}}$ are such that they guarantee the property of *liveness*, then $\mathcal{F}_{\mathrm{updRnd}}$ 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.
- $\mathcal{F}_{\mathrm{leak}}$ does not leak (additional) information as all information is leaked during submitting and reading.

Let $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$ be the protocol stack consisting of $\mathcal{F}_{\mathrm{ledger}}$ with all of its subroutines instantiated as sketched above. Then we can indeed show that $\mathcal{G}_{\mathrm{ledger}}$ (with the wrapper $\mathcal{W}_{\mathrm{ledger}}$) realizes $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$ (cf. Figure 3):
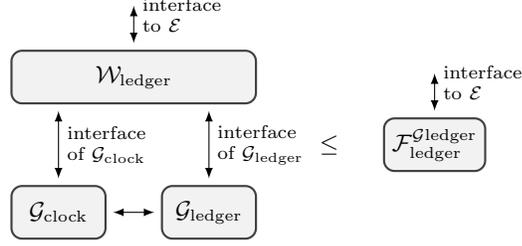
Fig. 3. Realization relation of $\mathcal{G}_{\text{ledger}}$ and $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$ as stated in Theorem 1. The system $\mathcal{E}$ denotes the environment, modeling, as usual in UC setting, arbitrary higher level protocols. All machines are additionally connected to the network adversary.

Theorem 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, including 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 C. Using this theorem, we obtain that existing realizations of $\mathcal{G}_{\text{ledger}}$ also apply to and can be re-used with $\mathcal{F}_{\text{ledger}}$:

Corollary 1 (informal). *Let $\mathcal{P}_{blockchain}$ be a realization of $\mathcal{G}_{\text{ledger}}$, e.g., Bitcoin or Ouroboros Genesis. Furthermore, let $\mathcal{Q}^{\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}}$ be a higher-level protocol using $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$ and let $\mathcal{Q}^{\mathcal{P}}$ be the same protocol as $\mathcal{Q}$ but using $\mathcal{P}_{blockchain}$ (plus the wrapper $\mathcal{W}_{\text{ledger}}$ and $\mathcal{G}_{\text{clock}}$) instead of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$. Then*

$$\mathcal{Q}^{\mathcal{P}} \text{ realizes } \mathcal{Q}^{\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}}$$

Proof. This directly follows from Theorem 1, transitivity of the realization relation, and the composition theorem of the iUC framework. □

The corollary intuitively states that if we have analyzed and proven secure a higher-level protocol $\mathcal{Q}$ based on $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$, then $\mathcal{Q}$ remains secure even if we run it with an actual blockchain $\mathcal{P}_{blockchain}$ that realizes $\mathcal{G}_{\text{ledger}}$. We emphasize again that the higher-level protocol $\mathcal{Q}$ modeled on top of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$ does not gain manual control over, e.g., the mining operations. However, we are not aware of any higher-level protocols that require/make use of such manual control. This includes the lightning network which Kiayias et al. [25] modeled and analyzed as a higher-level protocol on top of $\mathcal{G}_{\text{ledger}}$.

The ideal blockchain functionality $\mathcal{G}_{\text{PL}}$ ***with privacy.*** Kerber et al. proposed a derivative of $\mathcal{G}_{\text{ledger}}$, called $\mathcal{G}_{\text{PL}}$ [24], to analyze and prove security, including transaction privacy, of the proof-of-stake protocol Ouroboros Crypsinous: $\mathcal{G}_{\text{PL}}$ 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 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 $\mathcal{G}_{\text{PL}}$:

Theorem 2 (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 $\mathcal{G}_{\text{PL}}$. Then we have that $\mathcal{G}_{\text{PL}}$ (plus a wrapper) realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$.*

We provide further details regarding $\mathcal{G}_{\text{PL}}$, its relation to $\mathcal{F}_{\text{ledger}}$, and provide a proof sketch in Appendix D. Analogously to Corollary 1, this theorem directly implies that Ouroboros Crypsinous realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}_{\text{PL}}}$.

Other ideal blockchain functionalities. Besides $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{PL}}$, there are also various other ideal functionalities for blockchains [16, 17, 26], all of which are similar to the first ideal functionality introduced by Kiayias et al. [26]. These functionalities have in common that they are relatively simple compared to $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{PL}}$: 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 [3] that "the proposed ledger-functionality (introduced in [26]) 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 [16,17]. 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 $\mathcal{F}_{\mathrm{ledger}}$ appropriately to offer the exact same setup assumptions.

4 Case Study: Security and Privacy of the Corda Ledger

As already mentioned in the introduction, 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 $\mathcal{F}_{\mathrm{ledger}}$.

4.1 Description of the Corda Protocol

There are two types of participants in Corda: *(i) Nodes* or *clients*, which are the clients of Corda, i.e., they 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*,[6] 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 and output 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.[7]

---

[6] Technically, 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.

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

In the following, we call the set of input states, reference states, and smart contracts the *direct dependencies* of a transaction. The *(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 and output 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 they are 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 $\widehat{tx}$ that (directly or also indirectly) depends on tx, i.e., where both clients are allowed to and need to learn tx in order to validate $\widehat{tx}$.

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

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

Submitting transactions. A new transaction is first signed by the initiator, who then forwards the transaction to all signees to collect their signatures. The initiator then sends the transaction together with the signatures to the notary, who adds his own signature to confirm validity of the transaction. The initiator finally informs all signees that the transaction was successful. The initiator is required to know the full dependencies of the transaction such that he can distribute this information to signees and the notary. To obtain this knowledge in the first place, which might include input states known only to, say, one of the signees, clients/signees can proactively send known transactions to other clients. In what follows, we say that a client *pushes* a transaction (see Appendix E.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 [35]. The white paper of Corda [6] 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: 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.
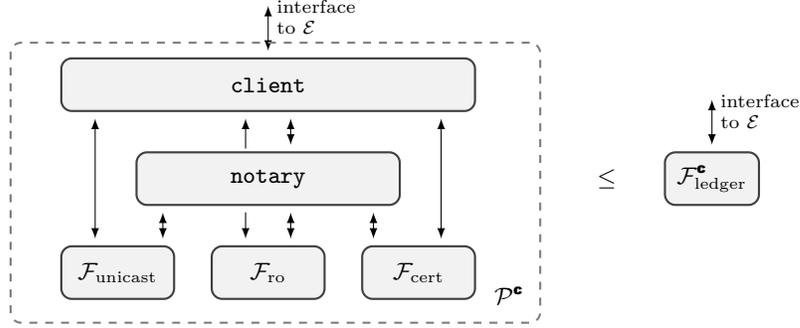
Fig. 4. The Corda protocol $\mathcal{P}^{\mathbf{c}}$ and the realization statement.

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

4.2 Model of Corda in the iUC Framework

Our model $\mathcal{P}^{\mathbf{c}}$ of Corda in the iUC framework closely follows the above description. Formally, $\mathcal{P}^{\mathbf{c}}$ is the protocol (client | notary, $\mathcal{F}_{\mathrm{unicast}}, \mathcal{F}_{\mathrm{cert}}, \mathcal{F}_{\mathrm{ro}}$) consisting of a client machine that is accessible to other (higher-level) protocols/the environment, an internal notary machine, and three ideal subroutines $\mathcal{F}_{\mathrm{unicast}}$, $\mathcal{F}_{\mathrm{cert}}$, and $\mathcal{F}_{\mathrm{ro}}$ 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 the process 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 $\mathcal{F}_{\mathrm{unicast}}$, 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 if they know a transaction that relies on (signatures of) a corrupted notary. This captures 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 they 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. Note that here we use the more general corruption model that we proposed in Section 2 to capture the security assumption of honest notaries in Corda. Using this modeling we do not have to encode this assumption explicitly into $\mathcal{F}_{\text{ledger}}$.

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

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. Furthermore, 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[8] 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 $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$, 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 $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ 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 $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ and discuss why this result implies that Corda itself in fact enjoys the above mentioned properties. Finally, we discuss some other common security properties from the literature and why they are not applicable to Corda.

On a technical level, we define the subroutines of $\mathcal{F}_{\text{ledger}}$ to obtain the instantiation $\mathcal{F}^{\mathbf{c}}_{\text{ledger}} = (\mathcal{F}_{\text{ledger}} \mid \mathcal{F}^{\mathbf{c}}_{\text{submit}}, \mathcal{F}^{\mathbf{c}}_{\text{read}}, \mathcal{F}^{\mathbf{c}}_{\text{update}}, \mathcal{F}^{\mathbf{c}}_{\text{updRnd}}, \mathcal{F}^{\mathbf{c}}_{\text{init}}, \mathcal{F}^{\mathbf{c}}_{\text{leak}}, \mathcal{F}^{\mathbf{c}}_{\text{storage}})$ as follows (cf. Figure 1, the additional subroutine $\mathcal{F}^{\mathbf{c}}_{\text{storage}}$ is explained below). We provide the formal specification of $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ in Appendix E.

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 its buffer and global graph that involve only honest clients and which either directly involve *pid*, or which

---

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

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 $\mathcal{A}$.

- $\mathcal{F}^{\mathbf{c}}_{\mathrm{init}}$ is parameterized by a set of participants. It provides this set to $\mathcal{F}_{\mathrm{ledger}}$.
- $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ handles *(i)* transaction submission, *(ii)* attachment submission, and *(iii)* pushing transactions/sending transactions proactively from one party to another party. In Case *(i)*, $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ ensures that an incoming transaction tx of some party *pid* has the expected format, which includes checking that all parties involved in the transaction are part of the predetermined set of participants. In addition, if *pid* is the initiator of the transaction, then $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ also performs two additional checks: firstly, it checks that *pid*'s current knowledge (provided by $\mathcal{A}$) is a subset of its potential knowledge and that *pid*'s current knowledge contains all inputs used by tx, including inputs for inputs of tx and so on. Secondly, it checks that tx is valid with respect to *pid*'s current knowledge (this state is a subset of the potential knowledge as determined by the adversary).

  If a transaction passes all checks, then $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ generates a new unique ID for that transaction and leaks the metadata of the transaction, including the ID, references to the IDs of other transactions, the set of involved parties, and length of the transaction body (cf. Footnote 8). If any of the involved clients is corrupted, then the full transaction body is leaked as well. Further, if the (honest) initiator of the transaction successfully validated the transaction that involves a corrupted client, the whole transaction subgraph below the submitted transaction is leaked.[9]

  If a party *pid* submits an attachments (Case *(ii)*), $\mathcal{F}_{\mathrm{submit}}$ checks the validity of the attachment. If the attachment passes the validity checks, $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ queues it for *pid*'s state and leaks the attachment ID and the length of the attachment. In Case *(iii)*, if a party pid_a (tries to) push a transaction identified by $txID$ to a party pid_b, $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ enforces that pid_a has access to all dependencies used in $txID$, i.e., $\mathcal{F}_{\mathrm{submit}}$ validates that all dependencies of tx are in the current knowledge of pid_a (and that the current knowledge is a subset of pid_a's potential knowledge, cf. Case *(i)*) and leaks that pid_a shared $txID$ with pid_b.

- $\mathcal{F}^{\mathbf{c}}_{\mathrm{update}}$ mainly handles the following operations: *(i)* expects updates to the global graph (proposed by the adversary $\mathcal{A}$) to contain two sets: a set of IDs of transactions/attachments from honest parties that are in the transaction buffer (i.e., have previously been submitted by the honest party), and a set of transactions/attachments from dishonest parties. Upon receiving this information, $\mathcal{F}^{\mathbf{c}}_{\mathrm{update}}$ checks that transactions/attachments from dishonest clients match the expected format of transactions/attachments, including unique IDs. Furthermore, $\mathcal{F}^{\mathbf{c}}_{\mathrm{update}}$ checks for all transactions in the extension that all honest signees of the transaction have already agreed to the transaction. Finally, $\mathcal{F}^{\mathbf{c}}_{\mathrm{update}}$ also checks that the resulting extended global graph does not contain double spending and is "complete", i.e., for all transactions in the global graph its inputs are also in the global graph. If any of the checks fails, the proposed extension is rejected. Otherwise, it is applied to the global state, with transactions of honest parties that have now been added to the global state being removed from the buffer set. Further, $\mathcal{F}^{\mathbf{c}}_{\mathrm{update}}$ allows to finalize transaction push operations (cf. Case *(iii)* above).

- $\mathcal{F}^{\mathbf{c}}_{\mathrm{read}}$ 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 subgraph g must also be a subgraph of the potential knowledge of *pid*, must be self-consistent, i.e., it must contain at least the previous outputs to read requests of that party, it must be a subset of *pid*'s current knowledge (which needs to be a subset of *pid*'s potential knowledge), and it must be complete (cf. $\mathcal{F}^{\mathbf{c}}_{\mathrm{update}}$ above). Furthermore, if there is a transaction with an honest initiator in the potential knowledge of *pid* which has been submitted at least 2δ time units ago, where δ is a parameter specifying the network delay, then it must be included in g. The graph g is then returned as response to the read request.

  $\mathcal{F}_{\mathrm{read}}$ does not provide any information upon read requests of corrupted parties (this is because all of the information that would be contained in the response to such a request has already been leaked by $\mathcal{F}_{\mathrm{submit}}$ and $\mathcal{F}_{\mathrm{leak}}$).

---

[9] Observe that the initiator allows the involved parties of a transaction to query missing dependencies. Thus, this operation grants a corrupted party access to this information.

- $\mathcal{F}^{\mathbf{c}}_{\mathrm{updRnd}}$ performs the following check upon receiving a time increment request. It looks for a transaction tx in the buffer where all involved clients are honest, which has been submitted by the initiator and acknowledged by all signees, and the last acknowledgment/the initiation (whatever came last) was received at least $\omega(tx)$ time units ago.[10] If such a transaction exists, then the time increment request is denied. Otherwise, it is accepted.
- In $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$, we use the additional (technical) subroutines $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$. $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$ provides an interface for all other $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ subroutines to query the potential knowledge of a party. Further, it is used to process the generation of unique IDs, to store them, to distribute them, and to access transaction/attachments by ID. As explained in Section 2, being able to add additional subroutines such as $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$ is a feature of $\mathcal{F}_{\mathrm{ledger}}$. Here we use this feature to easily synchronize internal state used for bookkeeping purposes across the subroutines of $\mathcal{F}_{\mathrm{ledger}}$. Note that this addition does not require changing $\mathcal{F}_{\mathrm{ledger}}$, which still only uses the fixed set of parameterized subroutines.
- On corruption of a client, $\mathcal{F}^{\mathbf{c}}_{\mathrm{leak}}$ computes its potential knowledge (cf. $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ above) and forwards this information to $\mathcal{A}$.
- To model $\mathcal{P}^{\mathbf{c}}$'s random oracle, the adversary $\mathcal{A}$ is also allowed to query $\mathcal{F}^{\mathbf{c}}_{\mathrm{update}}$ for transaction and attachment IDs. $\mathcal{A}$ obtains the corresponding unique ID (which was potentially freshly generated if the transaction did not exist in $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ yet).
- Due to potential leakage of transaction validity during the Corda protocol, $\mathcal{F}^{\mathbf{c}}_{\mathrm{read}}$ allows the adversary to query the validity of transactions regarding a parties pid current state. $\mathcal{F}^{\mathbf{c}}_{\mathrm{read}}$ enforces that the current state is a subset of the parties potential knowledge.

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

Theorem 3. *Let $\mathcal{P}^{\mathbf{c}}$ and $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ be as described above. Then, $\mathcal{P}^{\mathbf{c}} \leq \mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$.*

Here we provide a proof sketch with the core intuition. The full proof is given in Appendix E.

Proof (sketch). We show that $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ 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 $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ are indeed necessary for a successful simulation since the same information is also leaked by Corda. Hence, $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ 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 6 in Appendix E for a full definition) internally simulates a blinded instance of $\mathcal{P}^{\mathbf{c}}$, in the following called $\overline{\mathcal{P}^{\mathbf{c}}}$. During the simulation, $\mathcal{S}$ uses dummy transactions generated from the submission leakage. The dummy transaction is identified by the original transaction ID, contains all leaked data, as explained above, 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., `SendTransactionFlow` or `GetAttachment` flows 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 $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ synchronous such that read requests lead to the same output in real and ideal world. Note that the output from $\mathcal{S}$ to $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ never fails. $\overline{\mathcal{P}^{\mathbf{c}}}$ ensures that knowledge does not violate the boundaries of $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$, e.g., $\overline{\mathcal{P}^{\mathbf{c}}}$'s build-in network $\mathcal{F}_{\mathrm{unicast}}$ 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

---

[10] $\omega(tx)$ 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.

to all relevant information to answer request/handle operations indistinguishably between $\mathcal{P}^{\mathbf{c}}$ and $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$. This is due to the fact that $\mathcal{S}$ replaces the dummy transaction by 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 $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ 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 $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ for the validity of the transaction according to the honest party's knowledge. We will discuss both cases in more detail in the following discussion. □

We now discuss the implications of Theorem 3 for the security properties of Corda:

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

Liveness. $\mathcal{F}^{\mathbf{c}}_{\text{updRnd}}$ 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, $\mathcal{F}^{\mathbf{c}}_{\text{read}}$ ensures that transactions with honest initiators end up in the local state of all honest signees after another bounded time delay. By Theorem 3, these properties directly translate to Corda. A stronger liveness statement is not possible for Corda. Firstly, 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. Secondly, 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 $\mathcal{F}^{\mathbf{c}}_{\text{read}}$ 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 by someone that already knew the transaction. Furthermore, by definition of $\mathcal{F}^{\mathbf{c}}_{\text{submit}}$, if no dishonest client is involved in a new transaction, then 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 $\mathcal{F}^{\mathbf{c}}_{\text{submit}}$ 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 3 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 $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$, it is not possible to obtain a reasonably stronger privacy statement for meta-information in Corda.

Furthermore, observe that the adversary on $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ 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, $\mathcal{Q}$ using Corda/the ideal ledger such that $\mathcal{Q}$ 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. As already explained in the introduction, Corda does not provide the security property of *consistency* that is typically expected from blockchains. This is because consistency requires all clients to obtain (a prefix of) the full global state of the ledger, whereas clients in Corda generally obtain only a part of that global state.

In addition to *consistency*, the security literature for blockchains also often considers the security properties of *chain-quality* and *chain-growth*. These properties do not make sense for the case of Corda, which does not have the concept of blocks. For example, chain-growth essentially collapses to the property of liveness when interpreted on the level of individual transactions instead of blocks. Hence, we did not formalize and analyze these properties.

5   Acknowledgments

References

1. Accenture: 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 (10 2019), (Accessed on 05/26/2020)
2. Badertscher, C., Gazi, P., Kiayias, A., Russell, A., Zikas, V.: 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. pp. 913–930. ACM (2018)

---

[11] http://www.services-computing.de/?lang=en

3. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: 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, pp. 324–356. Springer (2017)

4. BCG: Digital ecosystems in trade finance. https://image-src.bcg.com/Images/BCG_Digital_Ecosystems_in_Trade_Finance_tcm38-229964.pdf (9 2019), (Accessed on 05/26/2020)

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

6. Brown, R.G.: The Corda platform: An introduction. https://www.r3.com/wp-content/uploads/2019/06/corda-platform-whitepaper.pdf (2020), (Accessed on 28/05/2020)

7. Camenisch, J., Enderlein, R.R., Krenn, S., Küsters, R., Rausch, D.: Universal Composition with Responsive Environments. In: Cheon, J.H., Takagi, T. (eds.) 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, pp. 807–840. Springer (2016), a full version is available at https://eprint.iacr.org/2016/034.

8. Camenisch, J., Krenn, S., Küsters, R., Rausch, D.: 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, pp. 191–221. Springer (2019), the full version is available at http://eprint.iacr.org/2019/1073

9. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally Composable Security with Global Setup. In: Vadhan, S.P. (ed.) Theory of Cryptography, Proceedings of TCC 2007. Lecture Notes in Computer Science, vol. 4392, pp. 61–85. Springer (2007)

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

11. Canetti, R., Jain, A., Scafuro, A.: Practical UC security with a Global Random Oracle. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014. pp. 597–608. ACM (2014)

12. CNBC: Swift announces tie-up with blockchain firm R3 at paris fintech forum. https://www.cnbc.com/2019/01/30/swift-announces-tie-up-with-blockchain-firm-r3-at-paris-fintech-forum.html (1 2019), (Accessed on 05/26/2020)

13. coindesk: 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 (5 2019), (Accessed on 06/02/2020)

14. Daian, P., Pass, R., Shi, E.: Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In: Financial Cryptography and Data Security 2019. LNCS, vol. 11598, pp. 23–41. Springer (2019)

15. David, B., Gazi, P., Kiayias, A., Russell, A.: 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, pp. 66–98. Springer (2018)

16. Dziembowski, S., Eckey, L., Faust, S., Hesse, J., Hostáková, K.: 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, pp. 625–656. Springer (2019)

17. Egger, C., Moreno-Sanchez, P., Maffei, M.: 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. pp. 801–815. ACM (2019)

18. Forbes: NASDAQ partnership with blockchain firm R3 is great for crypto. https://www.forbes.com/sites/benjessel/2020/05/22/why-nasdaqs-partnership-with-r3-is-great-for-digital-asset-adoption/ (5 2020), (Accessed on 05/26/2020)

19. Garay, J.A., Kiayias, A., Leonardos, N.: 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, pp. 281–310. Springer (2015)

20. Graf, M., Küsters, R., Rausch, D.: Accountability in a Permissioned Blockchain: Formal Analysis of Hyperledger Fabric. In: IEEE 5th European Symposium on Security and Privacy (EuroS&P 2020). IEEE Computer Society (2020)

21. Hewlett Packard Enterprise: Blockchain unchained. https://www.hpe.com/us/en/newsroom/blog-post/2018/07/blockchain-unchained.html (7 2018), (Accessed on 05/26/2020)
22. HM Land Registry: HM Land Registry to explore the benefits of blockchain. https://www.gov.uk/government/news/hm-land-registry-to-explore-the-benefits-of-blockchain (10 2018), (Accessed on 05/26/2020)
23. International Business Times: 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 (9 2015), (Accessed on 05/26/2020)
24. Kerber, T., Kiayias, A., Kohlweiss, M., Zikas, V.: Ouroboros Crypsinous: Privacy-Preserving Proof-of-Stake. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019. pp. 157–174. IEEE (2019)
25. Kiayias, A., Litos, O.S.T.: A Composable Security Treatment of the Lightning Network. In: IEEE 33rd Computer Security Foundations Symposium, CSF 2020, 22-25 July, 2020. IEEE Computer Society (2020)
26. Kiayias, A., Zhou, H., Zikas, V.: 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, pp. 705–734. Springer (2016)
27. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: 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. pp. 583–598. IEEE Computer Society (2018)
28. Küsters, R.: Simulation-Based Security with Inexhaustible Interactive Turing Machines. In: Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006). pp. 309–320. IEEE Computer Society (2006), see [30] for a full and revised version.
29. Küsters, R., Rausch, D.: A Framework for Universally Composable Diffie-Hellman Key Exchange. In: IEEE 38th Symposium on Security and Privacy (S&P 2017). pp. 881–900. IEEE Computer Society (2017)
30. Küsters, R., Tuengerthal, M., Rausch, D.: The IITM model: a simple and expressive model for universal composability. Journal of Cryptology **33**(4), 1461–1584 (2020)
31. McKinsey Digital: 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 (6 2018), (Accessed on 05/26/2020)
32. Pass, R., Seeman, L., Shelat, A.: 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, pp. 643–673 (2017)
33. R3: 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/ (11 2017), (Accessed on 06/02/2020)
34. R3: Corda source code. https://github.com/corda/corda (2020), (Accessed on 04/24/2020)
35. R3: R3 Corda master documentation. https://docs.corda.net/docs/corda-os/4.4.html (2020), (Accessed on 04/24/2020)
36. Reuters: 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 (9 2015), (Accessed on 05/26/2020)
37. Team, D.A.C.: Canton: A private, scalable, and composable smart contract platform. https://www.canton.io/publications/canton-whitepaper.pdf (06 2019), (Accessed on 11/27/2019)

Appendix:

A A Brief Introduction to the iUC Framework

This section provides a brief introduction to the iUC framework, which underlies all results in this paper. The iUC framework [8] 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 $\mathcal{Q} = \{M_1, \ldots, M_k\}$ is called a *system*. In a run of $\mathcal{Q}$, there can be one or more instances (copies) of each machine in $\mathcal{Q}$. One instance can send messages to another instance. At any point in a run, only a single instance is active, namely, the one to receive the last message; all other instances wait for input. The active instance becomes inactive once it has sent a message; then the instance that receives the message becomes active instead and can perform arbitrary computations. The first machine to run is the so-called *master*. The master is also triggered if the last active machine did not output a message. In iUC, the environment (see next) takes the role of the master. In the iUC framework a special user-specified **CheckID** algorithm is used to determine which instance of a protocol machine receives a message and whether a new instance is to be created (see below).

To define the universal composability security experiment (cf. [8]), one distinguishes between three types of systems: protocols, environments, and adversaries. As is standard in universal composability models, all of these types of systems have to meet a polynomial runtime notion . Intuitively, the security experiment in any universal composability model compares a protocol $\mathcal{P}$ with another protocol $\mathcal{F}$, where $\mathcal{F}$ is typically an ideal specification of some task, called *ideal protocol* or *ideal functionality*. The idea is that if one cannot distinguish $\mathcal{P}$ from $\mathcal{F}$, then $\mathcal{P}$ must be "as good as" $\mathcal{F}$. More specifically, the protocol $\mathcal{P}$ is considered secure (written $\mathcal{P} \leq \mathcal{F}$) if for all adversaries $\mathcal{A}$ controlling the network of $\mathcal{P}$ there exists an (ideal) adversary $\mathcal{S}$, called *simulator*, controlling the network of $\mathcal{F}$ such that $\{\mathcal{A}, \mathcal{P}\}$ and $\{\mathcal{S}, \mathcal{F}\}$ are indistinguishable for all environments $\mathcal{E}$. Indistinguishability means that the probability of the environment outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\}$ is negligibly close to the probability of outputting 1 in runs of the system $\{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ (written $\{\mathcal{E}, \mathcal{A}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$). The environment can also subsume the role of the network attacker $\mathcal{A}$, which yields an equivalent definition in the iUC framework. We usually show this equivalent but simpler statement in our proofs, i.e., that there exists a simulator $\mathcal{S}$ such that $\{\mathcal{E}, \mathcal{P}\} \equiv \{\mathcal{E}, \mathcal{S}, \mathcal{F}\}$ for all environments.

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

Roles of a protocol can be either public or private. The I/O interfaces of private roles are only accessible by other (entities belonging to) roles of the same protocol, whereas I/O interfaces of public roles can also be accessed by other (potentially unknown) protocols/the environment. Hence, a private role models some internal subroutine that is protected from access outside of the protocol, whereas a public role models some publicly accessible operation that can be used by other protocols. One uses the syntax "($\mathtt{pubrole_1}, \ldots, \mathtt{pubrole_n} \mid \mathtt{privrole_1}, \ldots, \mathtt{privrole_n}$)" to uniquely determine public and private roles of a

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

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

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 manages by the current instance. The first instance that accepts e then gets to process the incoming message. By default, **CheckID** accepts entities of a single party in a single session, which captures a traditional formulation of a real protocol. Other common definitions include accepting all entities from the same session, which captures a traditional formulation of an ideal functionality.
- The special variable $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathsf{role}_{\mathsf{cur}})$ refers to the currently active entity of the current machine instance (that was previously accepted by **CheckID**). If the current activation is due to a message received from another entity, then $(\mathsf{pid}_{\mathsf{call}}, \mathsf{sid}_{\mathsf{call}}, \mathsf{role}_{\mathsf{call}})$ refers to that entity.
- The special macro $\mathbf{corr}(pid_{sub}, sid_{sub}, role_{sub})$ can be used to obtain the current corruption status (i.e., whether this entity is still honest or considers itself to be implicitly/explicitly corrupted) of an entity belonging to a subroutine.
- The iUC framework supports so-called responsive environments and responsive adversaries [7]. 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 2 (Concurrent composition in iUC; informal). *Let $\mathcal{P}$ and $\mathcal{F}$ be two protocols such that $\mathcal{P} \leq \mathcal{F}$. Let $\mathcal{Q}$ be another protocol such that $\mathcal{Q}$ and $\mathcal{F}$ can be connected. Let $\mathcal{R} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{P})$ and let $\mathcal{I} \in \mathsf{Comb}(\mathcal{Q}, \mathcal{F})$ such that $\mathcal{R}$ and $\mathcal{I}$ agree on their public roles. Then $\mathcal{R} \leq \mathcal{I}$.*

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

so-called globally available state. This is unlike most other models for universal composability, such as the UC model, which impose several conditions on the structure of protocols for their composition theorem.

Additional Notation

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

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

B   The Ideal Ledger Functionality $\mathcal{F}_{\text{ledger}}$

In this section, we present the full specification of the ideal ledger functionality $\mathcal{F}_{\text{ledger}}$ in Figure 5 to 7. More technical details regarding iUC are available in Section A and the iUC paper [8].

Further details of $\mathcal{F}_{\text{ledger}}$: Additionally to the presented description in Section 2:

- $\mathcal{F}_{\text{ledger}}$ 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   Full Details: $\mathcal{G}_{\text{ledger}}$ Realizes $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$

In this section, we provide full details for Theorem 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

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

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

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

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

- identities $\subset \{0,1\}^* \times \mathbb{N}$, identities $= \emptyset$ {*The set of participants and the round when they occured first.*

- round $\in \mathbb{N}_{\geq 0}$, round $= 0$ {*Current (network) round in the protocol execution.*

- msglist $\subset \mathbb{N} \times \mathbb{N} \times \{\texttt{tx}, \texttt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, msglist $= \emptyset$. $\begin{cases} \textit{(Totally ordered) sequence of recorded messages} \\ \textit{that is considered as stable/immutable of the form} \\ (id, commitRound, type, msg, submitRound, pid). \textit{ If} \\ type = \texttt{meta}, pid = submitRound = \bot. \end{cases}$

- requestQueue $\subset \mathbb{N} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, requestQueue $= \emptyset$ $\begin{cases} \textit{The list of so far not ordered, honest, incoming "transactions". For-} \\ \textit{mat } (tmpCtr, tx, submittingRound, submittingParty). \end{cases}$

- readQueue $\subset \{0,1\}^* \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^*$, readQueue $= \emptyset$, $\begin{cases} \textit{The queue of read responses that need to be de-} \\ \textit{livered } (pid, responseId, round, msg) \end{cases}$

- readCtr $\in \mathbb{N}$, readCtr $= 0$,   {readCtr *is temporary ID for transactions in the* readQueue.

- reqCtr $\in \mathbb{N}$, reqCtr $= 0$,   {reqCtr *are temporary IDs for transactions in the* requestQueue.

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

We often use the CorruptionSet as specified in [8]. We often write $pid \in$ CorruptionSet instead of $(pid, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}) \in$ CorruptionSet for brevity.

CheckID($pid, sid, role$):
    Accept all messages with the same sid.
Corruption behavior:
    – **LeakedData**($pid, sid, role$):
       if $\exists(pid, registrationRound) \in$ identities, $registrationRound \in \mathbb{N}$:
          identities.remove($pid, registrationRound$)
       send (corrupt, pid, sid, internalState) **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{leak}} : \texttt{leak}$)
       wait for (corrupt, $leakage$)
       return($leakage$) $\begin{cases} \textit{Depending on the desired properties of } \mathcal{F}_{\text{ledger}}, \textit{ output after corruption needs} \\ \textit{to be specified} \end{cases}$
    – **AllowAdvMessage**($pid, sid, role, pid_{\text{receiver}}, sid_{\text{receiver}}, role_{\text{receiver}}, m$): $\mathcal{A}$ is not allowed to call subroutines on behalf of a corrupted party.
Initialization:
    send InitMe **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{init}} : \texttt{init}$)   {$\mathcal{F}_{\text{init}}$ *handels initilaization if necessary.*
    wait for (Init, $identities, msglist, corrupted, leakage$) **s.t.**
       1. $identities \subset \{0,1\}^* \times \{0\}, round \in \mathbb{N}, msglist \subset \mathbb{N} \times \mathbb{N} \times \{\texttt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*\}$,
       2. $corrupted \subset \{0,1\}^* \times \{\text{sid}_{\text{cur}}\} \times \{\texttt{client}\}$,
       3. $msg \in msglist$ are consecutivley enumarted started at 0,
       4. $\exists(\_, \_, \_, \_, a, b) \in msglist$, **s.t.** $a \neq \bot \vee b \neq \bot$
    identities $\leftarrow identities$, msglist $\leftarrow msglist$, CorruptionSet $\leftarrow corrupted$ $\begin{cases} \textit{We enforce correct formats and that} \\ \textit{msglist is a total ordered sequence.} \end{cases}$
    send responsively (Init, $leakage$) **to** NET   {*Send leaked information from initilaization to* $\mathcal{A}$.
    wait for ack **from** NET
MessagePreprocessing:
    recv ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role}_{\text{cur}}, msg$) **from** I/O:
       if ($\text{pid}_{\text{cur}}, \_$) $\notin$ identities $\wedge msg$ starts with Submit or Read:
          {*Register unknown party before its first submit/read operation*

          identities.add(pid_{cur}, round)

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

Main:

recv (Submit, msg) **from** I/O: {*Submission request from a honest identity*
    send (Submit, msg, internalState) **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{submit}} : \texttt{submit}$)   {*Forward request to $\mathcal{F}_{\text{submit}}$*

    wait for (Submit, $response$, $leakage$) **s.t.** $response \in \{\texttt{true}, \texttt{false}\}$
    if $response = \texttt{true}$:
       reqCtr $\leftarrow$ reqCtr $+ 1$
       requestQueue.add(reqCtr, round, $\text{pid}_{\text{cur}}, msg$) $\Big\{$ requestQueue.add($\_$) *equals* requestQueue $\leftarrow$ requestQueue $\cup \{\_\}$.
            Records message, round, identity and its state for "consensus"

    send (Submit, $response$, $leakage$) **to** NET $\Big\{$ *If $\mathcal{F}_{\text{submit}}$ leaks data regarding the submitted transaction, this is forwarded to $\mathcal{A}$.*

recv (Read, msg) **from** I/O: {*Read request from an honest identity*
    send (InitRead, msg, internalState) **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \texttt{read}$)   {*Forward the request to $\mathcal{F}_{\text{read}}$.*

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

    if $local$:
       send responsively (InitRead, $leakage$) **to** NET $(\star)$ {*$\mathcal{F}_{\text{read}}$ leaks data, this is forwarded to $\mathcal{A}$.*
       wait for (InitRead, $suggestedOutput$)   {*$\mathcal{A}$ may influence the read processing*
       send (FinishRead, msg, $suggestedOutput$, internalState) **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \texttt{read}$)
       wait for (FinishRead, $output$, $leakage'$)
       if $output = \bot$:   {*If $\mathcal{A}$'s input for $\mathcal{F}_{\text{read}}$ is not accepted, he is triggered again.*
          Go back to $(\star)$ and repeat the request (local variables $suggestedOutput$, $output$, and $leakage'$ are cleared)
       send responsively (FinishRead, $leakage'$) **to** NET   {*$\mathcal{F}_{\text{read}}$ leaks data, this is forwarded to $\mathcal{A}$.*
       wait for ack
       reply (Read, $output$)
    else:
       readCtr $\leftarrow$ readCtr $+ 1$; readQueue.add(pid, readCtr, round, msg)   {*In case of network read, store request*

       send (Read, readCtr, $leakage$) **to** NET   {*If $\mathcal{F}_{\text{read}}$ leaks data, this is forwarded to $\mathcal{A}$.*

recv (DeliverRead, $readCtr$, $suggestedOutput$) **from** NET **s.t.** $(pid, readCtr, r, msg) \in$ readQueue:
    $\Big\{$ *$\mathcal{A}$ tiggers message delivery per message (this may includes reordering of messages, non-delivery of messages, and manipulation of delivered data - if not enforced by $\mathcal{F}_{\text{updRnd}}$).*
    send (FinishRead, msg, $suggestedOutput$, internalState) **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \texttt{read}$)
    wait for (FinishRead, $output$, $leakage'$)
    if $output \neq \bot$:
       send responsively (FinishRead, $readCtr$, $leakage'$) **to** NET
       wait for ack
       readQueue.remove(pid, $readCtr$, r, msg)   {*Clean up readQueue.*
       send (Read, $output$) **to** (pid, sid_{cur}, I/O)
    else:
       send nack **to** NET   {*Delivery request of $\mathcal{A}$ was denied*

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

    send (CorruptedRead, pid, msg, internalState) **to** (pid, $\text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \texttt{read}$)   {*Forward request to $\mathcal{F}_{\text{read}}$*

    wait for (FinishRead, $leakage$)
    send (Read, pid, $leakage$) **to** NET   {*Forwarded data to $\mathcal{A}$.*

recv (Update, msg) **from** NET: {*Update or maintain request triggered by the adversary.*
    send (Update, msg, internalState) **to** (ϵ, $\text{sid}_{\text{cur}}, \mathcal{F}_{\text{update}} : \texttt{update}$)
    wait for (Update, $msglist$, $updRequestQueue$, $leakage$)
       s.t $msglist \subset \mathbb{N} \times \{\text{round}\} \times \{\texttt{tx}, \texttt{meta}\} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$
          {*$\mathcal{F}_{\text{update}}$ outputs which data to append to msglist and an updated requestQueue.*

    $max \leftarrow \max\{i | (i, \_, \_, \_, \_, \_) \in \text{msglist}\}$ $\Big\{$ *Check that msglist is a totally ordered sequence, extending the existing msglist. If msglist $= \emptyset$ then max defaults to -1*
    $check \leftarrow \text{msglist} \neq \emptyset \vee updRequestQueue \neq \emptyset$
    for $i = max + 1$ **to** $max + |msglist|$ **do**: $\Big\{$ *Check that there exists exactly one entry for every id i in a continous sequence (no gaps)*
       if $\not\exists_1 (i, \_, \_, \_, \_, \_) \in msglist$:
          $check \leftarrow \texttt{false}$
       if $\exists (i, \_, \texttt{meta}, \_, a, b) \in msglist \wedge (a \neq \bot \vee b \neq \bot)$:   {*Check that meta data has correct format*
          $check \leftarrow \texttt{false}$

    if $check$: $\Big\{$ *If update is totally ordered and no new messages where added to requestQueue, we accept the update.*
       msglist.add($msglist$)
       for all $item \in updRequestQueue$ **do**:   {*Remove elements "consumed" elements from requestQueue*
          requestQueue.remove($item$)
    reply (Update, $check$, $leakage$)   {*Inform $\mathcal{A}$ if update was successful and leake data.*

Fig. 6. The ideal ledger functionality $\mathcal{F}_{\text{ledger}}$ (Part 2).

```
Main:

    recv UpdateRound from NET:                                    {A triggers round update if current round satisfies rules of F_updRnd.
        send (UpdateRound, internalState) to (pid_cur, sid_cur, F_updRnd : updRnd)
        wait for (UpdateRound, response, leakage)
        if response = true:
            round ← round + 1
        reply (UpdateRound, response, leakage)

    recv GetCurRound:                                             {A and E are allowed to query the current round.
        reply (GetCurRound, round)

    recv DeRegister from I/O:                                     {De-register honest party
        Remove the unique tuple (pid_cur, r) from identities
        send responsively DeRegister to NET                       {Inform A on the deregistration
        wait for ack
        reply DeRegister
```

Fig. 7. The ideal ledger functionality $\mathcal{F}_{\text{ledger}}$ (Part 3).

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 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).[12]
- 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, the instantiation $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$ is 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

---

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

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 $\mathcal{G}_{\text{ledger}}$.

  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 $\mathcal{F}_{\mathrm{update}}$. 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 $\mathcal{F}_{\mathrm{ledger}}$ compared to $\mathcal{G}_{\mathrm{ledger}}$. 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 $\mathcal{G}_{\mathrm{ledger}}$ take some inputs that are not directly included in $\mathcal{F}_{\mathrm{ledger}}$, 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 $\mathcal{F}_{\mathrm{ledger}}$, i.e., essentially encode the full state and logic of $\mathcal{G}_{\mathrm{ledger}}$ 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. Such security properties can already be defined based on the information that is included in $\mathcal{F}_{\mathrm{ledger}}$ 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 $\mathcal{G}_{\mathrm{ledger}}$ or $\mathcal{F}_{\mathrm{ledger}}$) actually also depends on, say, the number of honest activations. Such a realization can still realize an ideal functionality that requires, e.g., consistency to always hold true independently of the number of honest activations: one can force the environment to always activate a sufficient number of honest parties within each time frame, modeling a setup assumption that is required for security to holds. This is a common technique that has already been used, e.g., for analyzing Bitcoin [3,19], including an analysis based on $\mathcal{G}_{\mathrm{ledger}}$. 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 E).

Hence, in the spirit of abstraction and simplification, we choose not to include further technical details of $\mathcal{G}_{\mathrm{ledger}}$ in $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$ but rather use the following mechanism to deal with any additional inputs to parameterized algorithms such as extendPolicy: Whenever one of the parameterized algorithms from $\mathcal{G}_{\mathrm{ledger}}$ is run within $\mathcal{F}_{\mathrm{ledger}}$, the adversary provides any missing inputs that are not defined in $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$, 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 $\mathcal{G}_{\mathrm{ledger}}$ while $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$ still inherits all properties that are enforced for the global transaction list, buffer list, and/or are related to time.

– The functionality $\mathcal{G}_{\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{ledger}}$, more specifically into $\mathcal{F}_{\mathrm{updRnd}}^{\mathcal{G}\mathrm{ledger}}$, 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 $\mathcal{F}_{\mathrm{ledger}}$.

However, if the parameters of $\mathcal{G}_{\mathrm{ledger}}$ are such that a certain time-related security property of the global transaction/block list is met, then $\mathcal{F}_{\mathrm{updRnd}}^{\mathcal{G}\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{updRnd}}^{\mathcal{G}\mathrm{ledger}}$ that, when they are set, enforce one or both of these security properties, and then show that this can be realized as long as $\mathcal{G}_{\mathrm{ledger}}$ is instantiated in such a way that it also provides these security properties. Clearly the same mechanism can also be used for capturing arbitrary other time-related security properties.

– There are some slight differences in the format of transactions and the global state between $\mathcal{F}_{\mathrm{ledger}}$ and $\mathcal{G}_{\mathrm{ledger}}$, with the key difference being that the global state of $\mathcal{G}_{\mathrm{ledger}}$ is a list of blocks, whereas the global state in $\mathcal{F}_{\mathrm{ledger}}$ 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 define still identify the boundaries of individual blocks. This is necessary for lifting the security properties of *consistency* from $\mathcal{G}_{\text{ledger}}$ to $\mathcal{F}_{\text{ledger}}$, namely, honest users (that have already been registered for a sufficiently long time) are guaranteed obtain a prefix of the global transaction list except for at most the last 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 MaintainLedger operation 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 [25], 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 $\mathcal{A}$ 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 [25] (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 $\mathcal{F}_{\text{ledger}}$.[13] 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 4. *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.

---

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

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 $(\texttt{Submit}, msg)$ 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 $\texttt{Preprocess}$ 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 a $\texttt{finalizePending}$ update message 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 $(\texttt{Submit}, msg)$ 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 $(\texttt{Read}, msg)$ 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 one additional read request from party pid). After this update, $\mathcal{S}$ simulates a read $(\texttt{Read}, msg)$ 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 $\texttt{UpdateRound}$ 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 bounded [14] and *(ii)* $\mathcal{S}$ is a responsive simulator for $\mathcal{I}$, i.e., restricting messages from $\mathcal{I}$ are answered immediately as long as $\{\mathcal{S}, \mathcal{I}\}$ runs with a responsive environment. We now argue that $\mathcal{R}$ and $\{\mathcal{S}, \mathcal{I}\}$ are indeed indistinguishable for any (responsive) environment $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$.

---

[14] This is the polynomial runtime notion employed by the iUC framework.

Now, let $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$ be an arbitrary but fixed environment. First, observe that $\mathcal{F}_{\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{ledger}}$, 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 $\mathcal{F}_{\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{ledger}}$.

Observe that, upon receiving a submission request, $\mathcal{F}_{\mathrm{ledger}}$ behaves just as $\mathcal{G}_{\mathrm{ledger}}$: 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 $\mathcal{G}_{\mathrm{ledger}}$. Any missing information, such as transactions of dishonest parties that are not part of the internal buffer of $\mathcal{F}_{\mathrm{ledger}}$, 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 $\mathcal{F}_{\mathrm{ledger}}$ to run the same deterministic algorithms as $\mathcal{G}_{\mathrm{ledger}}$ in $\mathcal{R}$, resulting in the same block extensions. Hence, the cached global state stored in $\mathcal{F}_{\mathrm{update}}^{\mathcal{G}\mathrm{ledger}}$ is synchronized with $\mathcal{R}$. Afterwards, $\mathcal{F}_{\mathrm{ledger}}$ uses the same deterministic algorithm as $\mathcal{G}_{\mathrm{ledger}}$ to validate the incoming transaction using the cached internal state from $\mathcal{F}_{\mathrm{update}}^{\mathcal{G}\mathrm{ledger}}$. 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 $\mathcal{F}_{\mathrm{ledger}}$ to apply the cached global state from $\mathcal{F}_{\mathrm{update}}^{\mathcal{G}\mathrm{ledger}}$ to the global state in $\mathcal{F}_{\mathrm{ledger}}$. By this, at the end of a submit request both the global state and the buffer of $\mathcal{F}_{\mathrm{ledger}}$ remain synchronized with $\mathcal{R}$.

Read requests: Observe that, upon receiving a read request, $\mathcal{F}_{\mathrm{ledger}}$ gives control to the simulator who then triggers a state update for the global transaction list in $\mathcal{F}_{\mathrm{ledger}}$. This matches the behavior of $\mathcal{G}_{\mathrm{ledger}}$, 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 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 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 $\mathcal{G}_{\mathrm{ledger}}$'s state format via toState). Since this pointer is

determined by the block that is returned in $\mathcal{R}'$, which by definition of $\mathcal{G}_{\mathrm{ledger}}$ is such that it meets all the requirements enforced by $\mathcal{I}$, we have that $\mathcal{F}_{\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{ledger}}$ every time an update to $\mathcal{G}_{\mathrm{clock}}$ in $\mathcal{R}'$ occurs, both worlds always output the same value for the current time. Note that, even if $\mathcal{F}_{\mathrm{updRnd}}^{\mathcal{G}_{\mathrm{ledger}}}$ 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 $\mathcal{G}_{\mathrm{clock}}$ advances. As the global state is synchronized, the same thus also holds true for $\mathcal{F}_{\mathrm{ledger}}$ whenever $\mathcal{S}$ updates the clock.

(De-)Registration: In the ideal world, $\mathcal{F}_{\mathrm{ledger}}$ registers honest entities whenever they first perform a submit or read request. This is identical to the behavior of the wrapper $\mathcal{W}_{\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{ledger}}$ by the simulator. Note in particular that these updates will be accepted by $\mathcal{F}_{\mathrm{ledger}}$ since $\mathcal{G}_{\mathrm{ledger}}$ 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.

<center>Description of the protocol $\mathcal{G}_{\mathrm{ledger}} = (\texttt{client})$:</center>

Participating roles: {client}
Corruption model: *dynamic*
Protocol parameters:

- validate
  $\begin{cases} \textit{Decides on the validity of a transaction with respect to the current state. Used to} \\ \textit{clean the buffer of transactions.} \end{cases}$

- extendPolicy
  $\begin{cases} \textit{The function that specifies the ledger's guarantees in extending the ledger state} \\ \textit{(e.g., speed, content etc.).} \end{cases}$

- Blockify
  $\{\textit{The function to format the ledger state output.}$

- predictTime
  $\{\textit{The function to predict the real-world time advancement.}$

- delay $\in \mathbb{N}$
  $\begin{cases} \textit{A general delay parameter for the time it takes for a newly joining (after the onset of the} \\ \textit{computation) miner to become synchronized.} \end{cases}$

- windowSize $\in \mathbb{N}$
  $\{\textit{The window size (number of blocks) of the sliding window.}$

<center>Description of $M_{\texttt{client}}$:</center>

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

- $\mathsf{P} \subset \{0,1\}^*$, identities $= \emptyset$
  $\{\textit{The (dynamic) set of registered participants.}$

- state $\subset \{0,1\}^*$, state $= \epsilon$
  $\{\textit{The ledger state, i.e., a sequence of blocks containing the content.}$

- nxtBC $\in \{0,1\}^*$, nxtBC $= \epsilon$.
  $\{\textit{Stores the current adversarial suggestion for extending the ledger state.}$

- buffer $\in \mathbb{N} \times \{0,1\}^* \times \mathbb{N} \times \{0,1\}^*$, buffer $= \epsilon$.
  $\begin{cases} \textit{The buffer of submitted input values of the form (txId, txContent,} \\ \textit{submissionTime, submittingParty).} \end{cases}$

- $\tau_L \in \mathbb{N}, \tau_L = 0$.
  $\{\textit{The current time as reported by the clock.}$

- $\tau_{\mathsf{state}} \in \mathbb{N} \times \mathbb{N}, \tau_{\mathsf{state}} = \emptyset$.
  $\begin{cases} \textit{A vector containing for each state block the time when the block added to the ledger} \\ \textit{state.} \end{cases}$

- $\mathsf{I}_{\mathsf{H}}^{\mathsf{T}} \in \{0,1\}^* \times \{0,1\}^* \times \mathbb{N}, \mathsf{I}_{\mathsf{H}}^{\mathsf{T}} = \emptyset$.
  $\begin{cases} \textit{The timed honest-input sequence with data of the form (msg, submittingParty,} \\ \textit{submissionTime).} \end{cases}$

- $\mathsf{PM} \subset \mathsf{P} \times \{\mathsf{d,s,c}\} \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^*, \mathsf{PM} = \emptyset$
  $\begin{cases} \textit{Storage to manage parties. An entry contains the party id, wheter the party} \\ \textit{is d - desynchronized, s - synchronized, or c - corrupted. he registration} \\ \textit{time, the pointer to the current state (or the block number), and its current} \\ \textit{state.} \end{cases}$

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

- **LeakedData**:

    recv (corrupt, $pid, sid, role$) **from** NET:
        if $\exists(pid, ps, \tau, pt, state) \in \mathsf{PM}$:
            $\mathsf{PM}.\texttt{remove}(pid, ps, \tau, pt, state); \mathsf{PM}.\texttt{add}(pid, \mathsf{c}, \tau, pt, state)$ $\{\textit{Record corruption in PM}$
        else:
            $\mathsf{PM}.\texttt{add}(pid, \mathsf{c}, \tau, pt, state)$
        return $\perp$ $\{\textit{We do not leak addtional data during corruption}$

- **AllowAdvMessage**: $\mathcal{A}$ is not allowed to call subroutines on behalf of a corrupted party.

<center>**Fig. 8.** The ideal ledger functionality $\mathcal{G}_{\mathrm{ledger}}$ translated into the iUC model (Part 1).</center>

MessagePreprocessing:

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

Main:

> **recv** Register **from** I/O **s.t.** $pid \notin \mathsf{P}$: $\{$*Registration process for honest parties*
>> $\mathsf{P}.\text{add}(pid); \mathsf{PM}.\text{add}(pid, \mathsf{d}, \tau_L, \epsilon, \epsilon)$
>> **reply** Register
>
> **recv** $(\text{Register}, pid, sid)$ **from** NET **s.t.** $pid \notin \mathsf{P}$: $\{$*Registration process for honest parties*
>> $\mathsf{P}.\text{add}(pid); \mathsf{PM}.\text{add}(pid, \mathsf{c}, \tau_L, \epsilon, \epsilon)$
>> **reply** Register
>
> **recv** DeRegister **from** I/O **s.t.** $(pid, \mathsf{s} \vee \mathsf{d}, \tau, pt, state) \in \mathsf{PM}$: $\{$*Deregistration of honest parties*
>> $\mathsf{P}.\text{remove}(pid); \mathsf{PM}.\text{remove}(pid, ps, \tau, pt, state)$
>> **reply** DeRegister
>
> **recv** $(\text{DeRegister}, pid, sid)$ **from** NET **s.t.** $(pid, \mathsf{c}, \tau, pt, state) \in \mathsf{PM}$: $\{$*Deregistration of corrupted parties*
>> $\mathsf{P}.\text{remove}(pid); \mathsf{PM}.\text{remove}(pid, ps, \tau, pt, state)$
>> **reply** DeRegister
>
> **recv** $(\text{GetRegistered}, sid)$ **from** NET: $\{\mathcal{A}$ *requests the current set of participants*
>> **reply** $(\text{GetRegistered}, sid, \mathsf{P})$
>
> **recv** GetRegistered **from** I/O: $\{\mathcal{E}$ *requests the current set of participants*
>> **reply** $(\text{GetRegistered}, \mathsf{P})$

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

Main:

recv (Submit, tx) **from** I/O **s.t.** $(\text{pid}_{\text{cur}}, \text{s} \vee \text{d}, \_, \_, \_) \in$ PM: $\{$*Honest parties submit a transaction*
    Choose a unique $txId$; $\hat{tx} \leftarrow (\textsf{txId}, tx, \tau_{\textsf{L}}, pid)$
    if $\textsf{validate}(\hat{tx}, \textsf{state}, \textsf{buffer}) = \texttt{true}$:
        $\textsf{buffer.add}(\hat{tx})$ $\{$*Record $\hat{tx}$ as queued for consensus*
        send (Submit, $\hat{tx}$) **to** NET

recv (Submit, pid, tx) **from** NET **s.t.** $(pid, \text{c}, \_, \_, \_) \in$ PM: $\{$*corrupted parties submit a transaction*
    Choose a unique $txId$; $\hat{tx} \leftarrow (\textsf{txId}, tx, \tau_{\textsf{L}}, pid)$
    if $\textsf{validate}(\hat{tx}, \textsf{state}, \textsf{buffer}) = \texttt{true}$:
        $\textsf{buffer.add}(\hat{tx})$ $\{$*Record $\hat{tx}$ as queued for consensus*
        send (Submit, $\hat{tx}$) **to** NET

recv Read **from** I/O **s.t.** $(pid, ps, \tau, pt, state) \in$ PM, $ps \in \{\text{s},\text{d}\}$: $\left\{\begin{array}{l}\textit{Hanlding of read requests from honest}\\\textit{parties}\end{array}\right.$
    $state' \leftarrow \textsf{state}|_{\min\{pt, |state|\}}$
    $\textsf{PM.remove}(pid, ps, \tau, pt, state); \textsf{PM.add}(pid, ps, \tau, pt, state')$
    reply (Read, $state'$)

recv (Read, pid) **from** NET **s.t.** $(pid, \text{d}, \tau, pt, state) \in$ PM:
    reply (Read, state, buffer, $\textsf{I}_{\textsf{H}}^{\textsf{T}}$) $\{$$\mathcal{A}$ *gets "full" knowledge of the status of the chain*

recv MaintainLedger **from** I/O **s.t.** $(pid, \text{s} \vee \text{d}, \_, pt, state) \in$ PM: $\left\{\begin{array}{l}\texttt{MaintainLedger}\textit{ command for honest par-}\\\textit{ties}\end{array}\right.$
    if $\textsf{predictTime}(\textsf{I}_{\textsf{H}}^{\textsf{T}}) > \tau_L$:
        send (Update, sid) **to** $(pid, sid, \mathcal{G}_{\text{clock}} : \texttt{clock})$
    else:
        send (MaintainLedger) **to** NET

recv (NextBlock, $hFlag, txid_1, \ldots, txid_l$) **from** NET: $\{$*Handling of suggested block candidates by $\mathcal{A}$*
    $nxtBC \leftarrow \epsilon$
    for $i = 1$ to l **do:**
        if $\exists (txid_i, tx_i, \tau_i, pid_i) \in$ buffer:
            $nxtBC.\textsf{add}(txid_i, tx_i, \tau_i, pid_i)$
    $nxtBC.\textsf{add}((hFlag, nxtBC)$
    reply (NextBlock, ack)

recv (SetSlack, $(p_1, pt'_1), \ldots, (p_l, pt'_l)$) **from** NET **s.t.** $\forall pid \in \{p_1, \ldots, p_l\} : (pid, \text{s}, \_, \_, \_) \in$ PM:
         $\{$$\mathcal{A}$ *may set the exact "state" of a synchronized party depending on* windowSize
    $u \leftarrow \{(p_1, pt'_1), \ldots, (p, pt'_l)\}$
    if $\exists (p_j, pt'_j) \in u$, **s.t.** $|\textsf{state}| - pt'_j > \textsf{windowSize} \vee pt'_j < |\textsf{state}_\textsf{j}|$: $\left\{\begin{array}{l}\textit{In the case that} \textsf{ windowSize} \textit{ is violated or}\\\textit{a pointer is moved backwards}\end{array}\right.$
        for all $(pid', (\text{s}), \tau', pt', state') \in$ PM **do:** $\{$*Update all synchronized parties to "longest chain"*
            $\textsf{PM.remove}(pid', (\text{s}), \tau', pt', state')$
            $\textsf{PM.add}(pid', (\text{s}), \tau', |\textsf{state}|, \textsf{state})$
    else:
        for $i = 1$ to l **do:**
            $\textsf{PM.remove}(p_\textsf{i}, ps_\textsf{i}, \tau_\textsf{i}, pt_\textsf{i}, state_\textsf{i})$
            $\textsf{PM.remove}(p_\textsf{i}, ps_\textsf{i}, \tau_\textsf{i}, pt'_\textsf{i}, state_\textsf{i})$
    reply (SetSlack, ack)

recv (DesyncState, $(p_1, state'_1), \ldots, (p, state'_l)$) **from** NET: $\left\{\begin{array}{l}\mathcal{A}\textit{ may set the "state" of}\\\textit{de-synchronized parties}\end{array}\right.$
    s.t. $pid \in \text{P} \wedge ps_i = \text{d}, \forall i = 1, \ldots l$, **s.t.** $(p_i, ps_i, \tau_i, pt_i, state_i)\text{PM}$
    for $i = 1$ to l **do:**
        $\textsf{PM.remove}(p_\textsf{i}, ps_\textsf{i}, \tau_\textsf{i}, pt_\textsf{i}, state_\textsf{i})$
        $\textsf{PM.remove}(p_\textsf{i}, ps_\textsf{i}, \tau_\textsf{i}, \epsilon, state'_\textsf{i})$
    reply (DesyncState, ack)

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

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

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

Description of M_{clock}:

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

    – $\mathsf{P} \subset \{0,1\}^* \times \{0,1\}, \mathsf{P} = \emptyset$ *{The set of registered participants of the form (pid, activated).*
    – $\tau \in \mathbb{N}$, initially $\tau = 0$ *{Current time in the $\mathcal{G}_{\mathrm{clock}}$*
    – $\mathsf{F} \in \{0,1\}$, initially $\mathsf{F} = 0$ *{Round update status of $\mathcal{G}_{\mathrm{ledger}}$.*
CheckID($pid, sid, role$):
    Accept all messages for the same sid.
Main:

      recv (Register, pid):   *{Handling registration for parties*
          $\mathsf{P}.\mathsf{add}(pid, 0)$
          reply (Register, pid)

      recv (DeRegister, pid):   *{Handling deregistration for parties*
          $\mathsf{P}.\mathsf{remove}(\mathsf{pid}_{\mathsf{cur}}, \_)$
          reply DeRegister

      recv GetRegistered: *{Output currently registered parties*
          reply (GetRegistered, P)

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

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

      recv ClockRead: *{Handling reads from the clock*
          reply (ClockRead, τ)

Fig. 11. The ideal clock functionality $\mathcal{G}_{\mathrm{clock}}$ from [2] translated into the iUC model.

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

Participating roles: {client}	
Corruption model: *incorruptible*[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: $\mathcal{G}_{\text{ledger}} : \texttt{client}, \mathcal{G}_{\text{clock}} : \texttt{clock}$
CheckID$(pid, sid, role)$:
    Accept all messages with the same sid.
Corruption behavior:
    – **DetermineCorrStatus**$(pid, sid, role)$:
        $corrupted \leftarrow \mathbf{corr}(pid_{\text{cur}}, sid_{\text{cur}}, role_{\text{cur}})$ {*Request corruption status at* $\mathcal{G}_{\text{ledger}}$
        return $corrupted$
Internal state:
    identities $\subset \{0,1\}^* \times \mathbb{N}$, identities $= \emptyset$ {*The set of participants and the round when they occured first.*
MessagePreprocessing:

    recv $(pid_{\text{cur}}, sid_{\text{cur}}, role_{\text{cur}}, msg)$ **from** I/O: {*We ensure for all input via* I/O *that the entity is registered.*

      $corrupted \leftarrow \mathbf{corr}(pid_{\text{cur}}, sid_{\text{cur}}, role_{\text{cur}})$ {*Request corruption status at* $\mathcal{G}_{\text{ledger}}$
      if $corrupted \wedge$ command is dedicated to $\mathcal{G}_{\text{ledger}}$:   {*See defined commands for* $\mathcal{G}_{\text{ledger}}$ *above*
        send msg **to** $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{ledger}} : \texttt{client})$ {*For corrupted parties:* $\mathcal{W}_{\text{ledger}}$ *acts as forwarder*
      if msg starts with Submit or Read: {*Register unknown party before its first submit/read operation*
        send Register **to** $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{clock}} : \texttt{clock})$ {*Register party at* $\mathcal{G}_{\text{clock}}$
        wait for (Register, sid, pid) **from** $\mathcal{G}_{\text{clock}} : \texttt{clock}$
        send Register **to** $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{ledger}} : \texttt{client})$ {*Register party at* $\mathcal{G}_{\text{ledger}}$
        wait for (Register, sid, pid) **from** $\mathcal{G}_{\text{ledger}} : \texttt{client}$
        for $(pid, \_) \in$ identities **do**:
          send CorruptionStatus? **to** $(pid, sid_{\text{cur}}, \mathcal{G}_{\text{ledger}} : \texttt{client})$
          wait for (CorruptionStatus, $corrupted$)
          if $corrupted$:
            identities.remove$(pid, \_)$ {*Remove corrupted identities*
        if $(pid_{\text{cur}}, \_) \notin$ identities: {*Record unknown parties*
          send ClockRead **to** $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{clock}} : \texttt{clock})$
          wait for (ClockRead, $round$)
          identities.add$(pid_{\text{cur}}, round)$
    recv msg **from** $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{ledger}})$: $\left\{\begin{array}{l}\text{In case that } \mathcal{G}_{\text{ledger}} \text{ sends a message via I/O, e.g., in the case that } \mathcal{A} \text{ triggers}\\ \text{sending a message via I/O, the message is forwarded to I/O. Note that this}\\ \text{part is not executed if } \mathcal{W}_{\text{ledger}} \text{ waits for an answer, e.g., during the reading}\\ \text{process due to the specification of iUC.}\end{array}\right.$
      send $(pid_{\text{cur}}, sid_{\text{cur}}, role_{\text{cur}}, msg)$ **to** I/O

Main:

    recv (Submit, msg) **from** I/O: {*Submission of a transaction from an uncorrupted party.*
      send (Submit, msg) **to** $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{ledger}} : \texttt{client})$ {*Forward the request to* $\mathcal{G}_{\text{ledger}}$

    recv (Read, msg) **from** I/O: {*Read from state from an uncorrupted identity.*
      send Read **to** $(pid_{\text{cur}}, sid_{\text{cur}}, \mathcal{G}_{\text{ledger}} : \texttt{client})$ {*Forward the request to* $\mathcal{G}_{\text{ledger}}$.
      wait for (Read, $state$)
      $msglist \leftarrow$ toMsglist$(state)$ {*Translate* $\mathcal{G}_{\text{ledger}}$*'s state format to* $\mathcal{F}_{\text{ledger}}$*'s message list format*
      reply (Read, $msglist$)

    recv (ClockUpdate, pid, sid) **from** NET: {$\mathcal{A}$ *triggers clock updates for honest and corrupted parties*
      send ClockUpdate **to** $(pid, sid, \mathcal{G}_{\text{clock}} : \texttt{clock})$ {*Trigger clock update.*

    recv (MaintainLedger, pid, sid) **from** NET: {$\mathcal{A}$ *triggers* MaintainLedger *for honest and corrupted parties*
      send MaintainLedger **to** $(pid, sid, \mathcal{G}_{\text{ledger}} : \texttt{client})$

    recv (GetCurRound) **from** NET: {$\mathcal{A}$ *requests* GetCurRound
      send ClockRead **to** $(\epsilon, sid_{\text{cur}}, \mathcal{G}_{\text{clock}} : \texttt{clock})$
      wait for (ClockRead, $round$)
      reply (GetCurRound, $round$)

Fig. 12. The wrapper $\mathcal{W}_{\text{ledger}}$ for $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{clock}}$ to prove that they realize $\mathcal{F}_{\text{ledger}}$ (Part 1).

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

Main:

    recv GetRegistered **from** I/O:
        send GetRegistered **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{G}_{\text{ledger}} : \text{ledger}$)
        wait for (GetRegistered, sid, P) **from** $\mathcal{G}_{\text{ledger}} : \text{ledger}$ $\left\{ \begin{array}{l} \mathcal{F}_{\text{ledger}} \text{ outputs only honest parties including the round} \\ \text{they registered} \end{array} \right.$
        $output = \{(pid, round) \in \text{identities} | pid \in P\}$
        reply (GetRegistered, $output$)

    recv DeRegister **from** I/O: $\{\mathcal{E}$ triggers deregistering of $(pid, sid, role)$ from $\mathcal{G}_{\text{clock}}$ and $\mathcal{G}_{\text{ledger}}$
        send DeRegister **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{G}_{\text{clock}} : \texttt{clock}$)
        wait for DeRegister
        send DeRegister **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{G}_{\text{ledger}} : \text{ledger}$)
        wait for DeRegister
        identities.remove($\text{pid}_{\text{cur}}, \_$) $\{$ To mimic behavior of $\mathcal{F}_{\text{ledger}}$
        reply DeRegister

Fig. 13. The wrapper $\mathcal{W}_{\text{ledger}}$ for $\mathcal{G}_{\text{ledger}}$ and $\mathcal{G}_{\text{clock}}$ to prove that they realize $\mathcal{F}_{\text{ledger}}$ (Part 2).

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

Participating roles: {submit}
Corruption model: *incorruptible*
Protocol parameters:
  – validate $\left\{ \begin{array}{l} \text{Validation "algorithm" (from } \mathcal{G}_{\text{ledger}} \text{) that states whether a transaction is valid according to} \\ \text{already ordered messages.} \end{array} \right.$

  – toBTX $\left\{ \begin{array}{l} \text{Algorithm that transforms a input tx to the format expected in } \mathcal{G}_{\text{ledger}}, \text{ i.e.,} \\ (msg', txID, \tau_L, pid). \text{ It includes housekeeping, e.g, txID is unique.} \end{array} \right.$

  – toState $\left\{ \begin{array}{l} \text{Algorithm that transforms } \mathcal{F}_{\text{ledger}} \text{'s message list format to } \mathcal{G}_{\text{ledger}} \text{'s state format. It may include} \\ \text{housekeeping, e.g, generating of unique block ids.} \end{array} \right.$

Description of M_{submit}:

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

    recv (Submit, $msg, internalState^a$) **from** I/O: $\left\{ \begin{array}{l} \text{See Figure 5 for definition of internalState and the local variables it} \\ \text{includes} \end{array} \right.$

        send (Preprocess, $msg, internalState$) **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{update}}^{\mathcal{G}\text{ledger}} : \texttt{update}$) $\left\{ \begin{array}{l} \text{Trigger } \mathcal{G}_{\text{ledger}} \text{'s prepro-} \\ \text{cessing at } \mathcal{F}_{\text{update}}^{\mathcal{G}\text{ledger}} \end{array} \right.$

        wait for (Preprocess, $msglist', buffer, leakage$) $\left\{ \begin{array}{l} \mathcal{F}_{\text{update}}^{\mathcal{G}\text{ledger}} \text{ provides the up-to-date/validated data} \\ \text{to } \mathcal{F}_{\text{submit}}^{\mathcal{G}\text{ledger}} \end{array} \right.$

        if validate(toBTX(msg), toState($msglist'$), $buffer$) = true: $\{$Emulate $\mathcal{G}_{\text{ledger}}$'s validation behavior
            reply (validationProcessed, true, [$msg, leakage$]) $\{\mathcal{A}$ receives all details on the transaction msg
        else:
            reply (validationProcessed, false, [$msg, leakage$]) $\{\mathcal{A}$ receives all details on msg

---
[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*.

Fig. 14. The write/submit functionality $\mathcal{F}_{\text{submit}}^{\mathcal{G}\text{ledger}}$ of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$

Participating roles: {read}
Corruption model: *incorruptible*
Protocol parameters:
- 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 : $\{0,1\}^* \to \mathbb{N} \cup \{\bot\}$ *{Mapping from identities to last transaction in their state; initially $\bot$*

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 *transcript* (included in the incoming message, resp. internalState), set pointer[pid] $\leftarrow \bot$.

Main:

       recv (InitRead, msg, *internalState*[a]) **from** I/O: *{See Figure 5 for definition of internalState and the local variables it includes*

          reply (InitRead, false, msg)   *{Reads in $\mathcal{G}_{\text{ledger}}$ may include state updates, thus we trigger $\mathcal{A}$ and leak the read request to $\mathcal{A}$.*

       recv (FinishRead, msg, $outID$, *internalState*) **from** I/O: *{$outID$ is the suggestedOutput from $\mathcal{A}$. For de-synced parties, it is directly the read output, for synced parties, it is a pointer to the output*

          Let (pid$_{\text{cur}}$, *roundRegistered*) for some *roundRegistered* $\in \mathbb{N}$ be the unique tuple in identities.

          send ProcessingOpen? **to** (pid$_{\text{cur}}$, 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 *response*:   *{If there are pending updates, they need to be propagated to $\mathcal{F}_{\text{ledger}}$*

             reply (FinishRead, $\bot$, ϵ)
          if *roundRegistered* $+ \delta \leq round$:   *{Party is synced*
             if $outID \notin \mathbb{N} \vee$ (pointer[pid$_{\text{cur}}$] $\neq \bot \wedge outID <$ pointer[pid$_{\text{cur}}$]):   *{$\mathcal{A}$ is supposed to send the same or later pointer*
                reply (FinishRead, $\bot$, ϵ)
             Let ($outID$, *committingRound*, tx, msg, _, _) be the unique transaction from $msglist$ with id $outID$.
             if ($outID$, *committingRound*, tx, msg', _, _) does not exist in $msglist$:   *{$\mathcal{A}$ is supposed to send an existing pointer*
                reply (FinishRead, $\bot$, ϵ)
             Let $b_{outID} \in \mathbb{N}$ the block number of the block that contains the transaction with id $outID$ (according to (meta, cut) messages in $msglist$ starting from block 1).
             Let $b_{\text{current}} \in \mathbb{N}$ the block number of the current/most recent block in $msglist$.
             if $b_{outID} +$ windowSize $< b_{\text{current}}$:
                reply (FinishRead, $\bot$, ϵ)
             pointer[pid$_{\text{cur}}$] $\leftarrow outID$ *{Update last 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)   *{$\mathcal{A}$ decides the output. $outID$ is suggestedOutput.*
       recv (CorruptedRead, pid, msg, *internalState*) **from** I/O:
          reply (FinishRead, ϵ)   *{$\mathcal{A}$ already knows the full chain*

---

[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*.

Fig. 15. The read functionality $\mathcal{F}_{\text{read}}^{\mathcal{G}\text{ledger}}$ of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$

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

Participating roles: {update}
Corruption model: *incorruptible*
Protocol parameters:

- validate $\qquad\qquad$ { *Validation "algorithm" (from $\mathcal{G}_{\mathrm{ledger}}$) that states whether a transaction is valid according to already ordered messages.*

- extendPolicy $\qquad\qquad$ { extendPolicy *from $\mathcal{G}_{\mathrm{ledger}}$. Defines how state "evolves".*

- toMsglist $\qquad\qquad$ { *"Algorithm" that transforms output of* extendPolicy *from $\mathcal{G}_{\mathrm{ledger}}$ to msglist format (including metadata).* extendPolicy *may include format/data transformation.*

- toState $\qquad\qquad$ { *Algorithm that transforms $\mathcal{F}_{\mathrm{ledger}}$'s message list format to $\mathcal{G}_{\mathrm{ledger}}$'s state format. It may include housekeeping, e.g, generating of unique block ids.*

- ToBuffer $\qquad\qquad$ { *Algorithm that transforms $\mathcal{F}_{\mathrm{ledger}}$'s* requestQueue *format to $\mathcal{G}_{\mathrm{ledger}}$'s* buffer *format. It may include housekeeping, e.g, generating of unique ids.*

- toBTX $\qquad\qquad$ { *Algorithm that transforms a input tx to the format expected in $\mathcal{G}_{\mathrm{ledger}}$, i.e., $(msg', txID, \tau_L, pid)$. It includes housekeeping, e.g, txID is unique.*

Description of M_{update}:

Implemented role(s): {update}
CheckID(pid, sid, $role$):
$\qquad$ Accept all messages with the same sid.
Internal state:

- recordedActivations : $\{0,1\}^*$ $\qquad\qquad$ { *Emulated honest activations of $\mathcal{G}_{\mathrm{ledger}}$*
- msgListAppend$^{\mathsf{p}}$: $\{0,1\}^*$ $\qquad\qquad$ { *Cache for pending update msglist*
- UpdRequestQueue$^{\mathsf{p}}$: $\{0,1\}^*$ $\qquad\qquad$ { *Cache for pending update requestQueue*

Main:

$\qquad$ **recv** (Update, [pending&new, msg], $internalState^a$) **from** I/O: $\qquad$ { *See Figure 5 for definition of internalState and the local variables it includes*

$\qquad\qquad$ **if** $msg \neq (I^T_H, nxtBC, corrBuffer, \tau_{state})$, **s.t.** $I^T_H, corrBuffer, \tau_{state}$ match the format from $\mathcal{G}_{\mathrm{ledger}}$, $nxtBC$ is:
$\qquad\qquad$ a sequence of $nxtBC$'s in the format of $\mathcal{G}_{\mathrm{ledger}}$
$\qquad\qquad\qquad\qquad$ { *Check message format*
$\qquad\qquad\qquad$ **reply** (Update, $\emptyset, \emptyset, \epsilon$) $\qquad\qquad$ { *Processing aborted*
$\qquad\qquad$ **if** $\exists$ a transaction of an uncorrupted pid in $corrBuffer$:
$\qquad\qquad\qquad$ **reply** (Update, $\emptyset, \emptyset, \epsilon$) $\qquad\qquad$ { *Processing aborted*
$\qquad\qquad$ recordedActivations.add(I^T_H) $\qquad\qquad$ { *Update recordedActivations*

$\qquad\qquad$ ProcessUpdate() $\qquad$ { *See definition of* ProcessUpdate() *below. The procedure gets the complete internal state of $\mathcal{F}^{\mathcal{G}\mathrm{ledger}}_{\mathrm{update}}$ and all currently used local variables as input. It may write to local and global variables or create new local variables.*

$\qquad\qquad$ msgListAppend$^{\mathsf{p}} \leftarrow \emptyset$; UpdRequestQueue$^{\mathsf{p}} \leftarrow \emptyset$ $\qquad$ { *Clear variables for pending upgrade*
$\qquad\qquad$ **reply** (Update, $msgListAppend$, $updRequestQueue$, $leakage$) $\qquad$ { *Return updates.*

$\qquad$ **recv** (Update, [finalizePending], $internalState$) **from** I/O: $\qquad$ { *$\mathcal{A}$ may finalize an update without triggering a new one*

$\qquad\qquad$ $msgListAppend \leftarrow$ msgListAppend$^{\mathsf{p}}$; $updRequestQueue \leftarrow$ UpdRequestQueue$^{\mathsf{p}}$
$\qquad\qquad$ $leakage \leftarrow (msgListAppend, updRequestQueue)$
$\qquad\qquad$ msgListAppend$^{\mathsf{p}} \leftarrow \emptyset$; UpdRequestQueue$^{\mathsf{p}} \leftarrow \emptyset$ $\qquad$ { *Clear variables for pending upgrade*
$\qquad\qquad$ **reply** (Update, $msgListAppend$, $updRequestQueue$, $leakage$) $\qquad$ { *Return updates.*

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

Fig. 16. The update functionality $\mathcal{F}^{\mathcal{G}\mathrm{ledger}}_{\mathrm{update}}$ of $\mathcal{F}^{\mathcal{G}\mathrm{ledger}}_{\mathrm{ledger}}$ (Part 1)

Description of M_{update} (continued):

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 for- **do**
      mat of $\mathcal{G}_{\text{ledger}}$,
     3. $\nexists$ a transaction of an uncorrupted pid in
      $corrBuffer$,
   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 $\mathcal{F}_{\text{submit}}$*

  $\mathsf{msgListAppend}^{\mathsf{p}} \leftarrow (msgListAppend); \mathsf{UpdRequestQueue}^{\mathsf{p}} \leftarrow (updRequestQueue)$ $\left\{ \begin{array}{l} Record~pending~up-\\ dates \end{array} \right.$

  reply (Preprocess, $state$, ToBuffer($updRequestQueue$) $\cup$ $corrBuffer$, $leakage$) $\left\{ \begin{array}{l} Return~~~~~data~~~~to\\ \mathcal{F}_{\text{submit}}. \end{array} \right.$

 recv ProcessingOpen? **from** I/O: {I/O *round can query whether there are pending state updates*
  if $\mathsf{msgListAppend}^{\mathsf{p}} \neq \emptyset \vee \mathsf{UpdRequestQueue}^{\mathsf{p}} \neq \emptyset$:
   reply (ProcessingOpen?, true)
  else:
   reply (ProcessingOpen?, true)

Procedures and Functions:

 procedure ProcessUpdate() :
  if $\mathsf{msgListAppend}^{\mathsf{p}} \neq \emptyset \vee \mathsf{UpdRequestQueue}^{\mathsf{p}} \neq \emptyset$:
   $msglist$.add($\mathsf{msgListAppend}^{\mathsf{p}}$); $requestQueue \leftarrow requestQueue^{\mathsf{c}}$ {*Include pending updates in update process*
  $updRequestQueue \leftarrow \emptyset$

  $N \leftarrow$ extendPolicy(I_H^T, toState($msglist$), $nxtBC$, ToBuffer($requestQueue$) $\cup$ $corrBuffer$, τ_{state})
                    {*Emulate $\mathcal{G}_{\text{ledger}}$ behavior during update*
  $msgListAppend \leftarrow \mathsf{msgListAppend}^{\mathsf{p}}; updRequestQueue \leftarrow \mathsf{UpdRequestQueue}^{\mathsf{p}}$
  if $N \neq \epsilon$:   {*Process update, if* extendPolicy *produces an update*
   $msgListAppend$.add(toMsglist(N, $msglist$))   {*Transform output of* extendPolicy *to* msglist *format*
  for all tx in $requestQueue$ **do:**
   if $\neg$validate(toBTX(tx), toState($msglist$), ToBuffer($updRequestQueue$) $\cup$ $corrBuffer$):
                  {*Emulate $\mathcal{G}_{\text{ledger}}$'s update behavior*

    Remove entry of tx from $requestQueue$, add it to $updRequestQueue$
  $leakage \leftarrow (msgListAppend, updRequestQueue)$

Fig. 17. The update functionality $\mathcal{F}_{\text{update}}^{\mathcal{G}\text{ledger}}$ of $\mathcal{F}_{\text{ledger}}^{\mathcal{G}\text{ledger}}$ (Part 2)

<div align="center">Description of the subroutine $\mathcal{F}_{\mathrm{updRnd}}^{\mathcal{G}\mathrm{ledger}} = (\mathrm{updRnd})$:</div>

Participating roles: {updRnd}
Subroutines: $\mathcal{F}_{\mathrm{update}}^{\mathcal{G}\mathrm{ledger}}$: update
Corruption model: *incorruptible*
Protocol parameters:
  - $\rho \in \mathbb{N}$ {*The upper bound in rounds after which a honest tx should be in the state.*
  - growthWindow $\in \mathbb{N}$ {*The number of rounds the* growthRate *is related to*
  - growthRate $\in \mathbb{N}$ {*The number of "blocks" that should be added at in any* growthWindow *consecutive rounds*
  - ensureLiveness $\in$ {true, false}   {*If* true, *liveness is ensured*
  - ensureGrowth $\in$ {true, false}   {*If* true, *chain-growth is ensured*

<div align="center">Description of M_{updRnd}:</div>

Implemented role(s): {updRnd}
Subroutines: $\mathcal{F}_{\mathrm{update}}^{\mathcal{G}\mathrm{ledger}}$: update, $\mathcal{F}_{\mathrm{state}}$: state
CheckID(pid, sid, $role$):
     Accept all messages with the same sid.
Main:

    recv (UpdateRound, msg, $internalState^a$) **from** I/O: {*See Figure 5 for definition of internalState and the local variables it includes*

        send (ProcessingOpen?) **to** ($\mathrm{pid_{cur}}$, $\mathrm{sid_{cur}}$, $\mathcal{F}_{\mathrm{update}}$: update)   {*Check whether there are pending state updated*
        wait for (ProcessingOpen?, $response$)
        if $response$:   {*Round may not perceed if there are pending updates*
            reply (UpdateRound, false, ϵ)
        if ensureLiveness $\wedge \exists(\_, r, \_) \in$ requestQueue, **s.t.** $r \in \mathbb{N} \wedge r <$ round $- \rho$:
            reply (UpdateRound, false, ϵ)
        if ensureGrowth $\wedge$ round $\geq$ growthWindow:
            if $\#(\_, \_, \mathrm{meta}, \mathrm{cut}, \_, \_)$ in the last growthWindow committing rounds in $msglist$ is smaller than growthRate:
                reply (UpdateRound, false, ϵ)
        if $\exists(pid, responseID, round, msg) \in readQueue \wedge (pid, \_) \in$ identities: {*Check that all honest read requests in this round are processed*
            reply (UpdateRound, false, ϵ)
        else:
            reply (UpdateRound, true, ϵ)

---

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

<div align="center">**Fig. 18.** The round update/time update functionality $\mathcal{F}_{\mathrm{updRnd}}^{\mathcal{G}\mathrm{ledger}}$ of $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$</div>

<div align="center">Description of the subroutine $\mathcal{F}_{\mathrm{leak}}^{\mathcal{G}\mathrm{ledger}} = (\mathrm{leak})$:</div>

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

<div align="center">Description of M_{leak}:</div>

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

    recv (Corrupt, pid, sid, $internalState$) **from** I/O: {*See Figure 5 for definition of internalState and the local variables it includes*

        reply (Corrupt, ϵ)   {$\mathcal{A}$ *already had full overview of the state*

<div align="center">**Fig. 19.** The leakage subroutine $\mathcal{F}_{\mathrm{leak}}^{\mathcal{G}\mathrm{ledger}}$ of $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$</div>

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

Participating roles: {init}
Corruption model: *incorruptible*
Protocol parameters:
    − extendPolicy   { *extendPolicy from* $\mathcal{G}_{\mathrm{ledger}}$. *Defines how state "evolves".*
    − toMsglist   { *Algorithm that transferst output from* extendPolicy *to the format of* msglist *as specified in* $\mathcal{F}_{\mathrm{ledger}}$ *(Figure 5)*.

Description of M_{init}:

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

        recv Init **from** I/O: { $\mathcal{F}_{\mathrm{init}}^{\mathcal{G}\mathrm{ledger}}$ *runs* extendPolicy *wich may produce a genesis block*
            $N \leftarrow$ extendPolicy($\emptyset, \emptyset, \epsilon, \emptyset, \epsilon$)
            $msglist \leftarrow$ toMsglist(N)
            reply (Init, $\emptyset$, $msglist$, $\emptyset$, $msglist$)   { *The initial data is leaked to* $\mathcal{A}$

Fig. 20. The initialization functionality $\mathcal{F}_{\mathrm{init}}^{\mathcal{G}\mathrm{ledger}}$ of $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$

D $\mathcal{G}_{\mathbf{PL}}$ Realizes $\mathcal{F}_{\mathbf{ledger}}$ (Sketch)

In this section, we sketch how $\mathcal{F}_{\mathrm{ledger}}$ can be instantiated such that Kerber et al.'s ideal functionality for privacy in blockchains $\mathcal{G}_{\mathrm{PL}}$ [24] also realizes that instantiation of $\mathcal{F}_{\mathrm{ledger}}$. Since $\mathcal{G}_{\mathrm{PL}}$ is a variant of $\mathcal{G}_{\mathrm{ledger}}$, we first briefly describe the key differences between $\mathcal{G}_{\mathrm{PL}}$ and $\mathcal{G}_{\mathrm{ledger}}$ and how $\mathcal{G}_{\mathrm{PL}}$ models privacy properties.[15] Then, we explain how the subroutines of $\mathcal{F}_{\mathrm{ledger}}$ need to be instantiated – we call the resulting instantiation $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$ in what follows – in order to prove that $\mathcal{G}_{\mathrm{PL}}$ realizes $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$.

D.1   The Private Ledger Functionality $\mathcal{G}_{\mathbf{PL}}$

The functionality $\mathcal{G}_{\mathrm{PL}}$ inherits most of its construction from $\mathcal{G}_{\mathrm{ledger}}$ while mainly adding two additional features: *(i)* $\mathcal{G}_{\mathrm{PL}}$ allows for modeling privacy properties of blockchains, and *(ii)* $\mathcal{G}_{\mathrm{PL}}$ includes (initial) stake distributions. This is to enable capturing the Proof-of-Stake-based blockchain Ouroboros Crypsinous [24].

    On a technical level, Kerber et al. add the following new parameters to $\mathcal{G}_{\mathrm{PL}}$ 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 $\mathcal{G}_{\mathrm{PL}}$, each party gets a blind'ed response to its read requests. That is, it cannot directly access the full global state of $\mathcal{G}_{\mathrm{PL}}$. Similarly, $\mathcal{A}$ only gets access to a blind'ed version of $\mathcal{G}_{\mathrm{PL}}$'s state and all potential leakages, such as the leakage during transaction submission, are blinded via one of the additional parameters.

    Furthermore, $\mathcal{G}_{\mathrm{PL}}$ includes an explicit (coin) ID generation interface Generate that allows to generate private IDs, e.g., private coin IDs which is necessary for Ouroboros Crypsinous to, e.g., handle privacy during "mining". Finally, Kerber et al. also change the behavior of some parameters compared to $\mathcal{G}_{\mathrm{ledger}}$. In particular, *(i)* the activation list $\mathsf{I}_{\mathsf{H}}^{\mathsf{T}}$ 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.

D.2 $\mathcal{G}_{\mathbf{PL}}$ Realizes $\mathcal{F}_{\mathbf{ledger}}^{\mathcal{G}_{\mathbf{PL}}}$

Now, we sketch the instantiation $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$ which is realized by $\mathcal{G}_{\mathrm{PL}}$. This instantiation is similar to the instantiation used for $\mathcal{G}_{\mathrm{ledger}}$ in Appendix C and uses the same high-level idea, namely, running the parameters of $\mathcal{G}_{\mathrm{PL}}$ inside the subroutines of $\mathcal{F}_{\mathrm{ledger}}$. There is, however, one conceptual difference between $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$ and $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$: In $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$, we were able to abstract from some of the (not security relevant) technical details of

---

[15] For a detailed presentation of $\mathcal{G}_{\mathrm{PL}}$ we refer to the original paper [24].

$\mathcal{G}_{\mathrm{ledger}}$, 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 $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$. However, this is not possible for $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$: The information that $\mathcal{G}_{\mathrm{PL}}$ 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 $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$ in Appendix C, namely, we encode the full logic of $\mathcal{G}_{\mathrm{PL}}$ with all technical details within $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$. On a technical level, $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$ differs from $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}\mathrm{ledger}}$ mainly in the following points: *(i)* $\mathcal{F}_{\mathrm{init}}^{\mathcal{G}_{\mathrm{PL}}}$ needs to distribute the initial stake distribution of $\mathcal{G}_{\mathrm{PL}}$, *(ii)* $\mathcal{F}_{\mathrm{read}}^{\mathcal{G}_{\mathrm{PL}}}$ needs a special handling for the `Generate` command which matches $\mathcal{G}_{\mathrm{PL}}$, *(iii)* handling of leakage needs to be enhanced by $\mathcal{G}_{\mathrm{PL}}$'s leakage parameters, and *(iv)* `Read` needs to `blind` outputs before they are delivered to the requestor.

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

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

Theorem 5. *(informal) Let $\mathcal{G}_{\mathrm{PL}}$ be the ideal blockchain functionality with arbitrary parameters such that all parameterized algorithms are deterministic. Let $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$ be the instantiation of $\mathcal{F}_{\mathrm{ledger}}$ as described above, where the internal subroutines use the same parameters as $\mathcal{G}_{\mathrm{PL}}$. Furthermore, if $\mathcal{G}_{\mathrm{PL}}$ is parameterized such that it provides* liveness *and/or* chain-growth*, then let the parameters of the subroutine $\mathcal{F}_{\mathrm{updRnd}}^{\mathcal{G}_{\mathrm{PL}}}$ in $\mathcal{F}_{\mathrm{ledger}}^{\mathcal{G}_{\mathrm{PL}}}$ be set such that it also enforces the same properties. Then:*

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

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

E Full Details: Corda Realizes $\mathcal{F}_{\mathbf{ledger}}^{\mathbf{c}}$

In the following, we illustrate a flow between different parties in Corda using the submission of a transaction as example. This is succeeded by full details for Theorem 3, including formal specifications of all machines and a formal proof of the theorem.

E.1 Example Run

Figure 21 illustrates the process of submitting a new transaction to Corda. The process is started by an initiator who submits, validates, and signs the whole 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 notary responsible for that transaction (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.

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
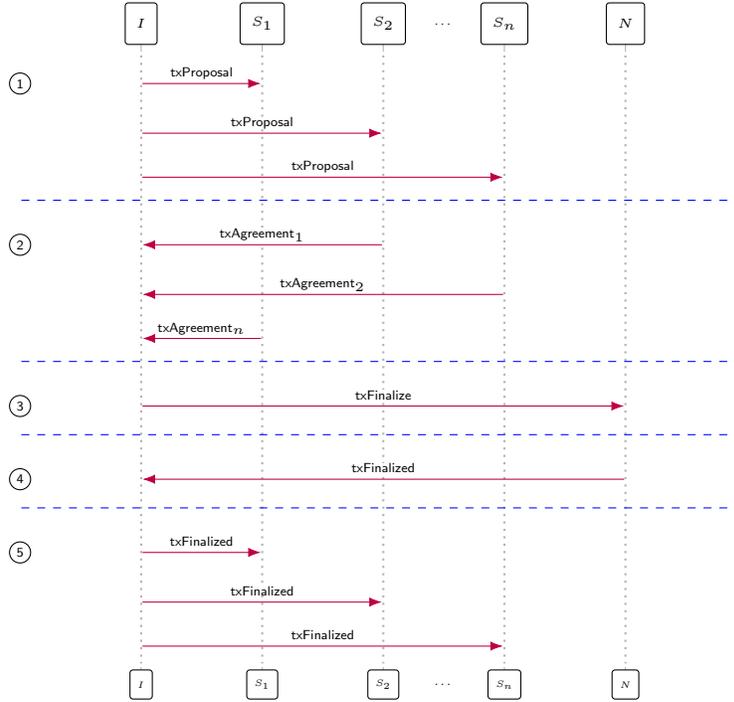
Fig. 21. Overview of a common transaction in Corda with an initiator of the transaction I, transaction signees $S_1, \ldots, S_n$, and a notary service N.

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.

E.2 The Corda Model $\mathcal{P}^{\mathbf{c}}$

In this section, we provide additional details and highlights regarding our Corda model $\mathcal{P}^{\mathbf{c}}$ introduced in Section 4.2). We provide the formal definitions of $\mathcal{P}^{\mathbf{c}}$ in Figures 24 to 35 at the end of this section.

Remark: To simplify presentation, we introduced $\mathcal{P}^{\mathbf{c}}$ as protocol ($\texttt{client} \mid \texttt{notary}, \mathcal{F}_{\mathrm{unicast}}, \mathcal{F}_{\mathrm{cert}}, \mathcal{F}_{\mathrm{ro}}$) in Section 4.2. Formally, the Corda protocol is defined as $\mathcal{P}^{\mathbf{c}} = (\mathcal{P}^{\mathbf{c}}_{\mathrm{ledger}} : \texttt{client} \mid \mathcal{P}^{\mathbf{c}}_{\mathrm{ledger}} : \texttt{notary}, \mathcal{F}_{\mathrm{unicast}}, \mathcal{F}_{\mathrm{cert}}, \mathcal{F}_{\mathrm{ro}}$) (more details below).

The core of the Corda model which is mainly specified in the machines for clients and notaries consists of the resolution and validation methods depicted in Figure 23.[16] as well as the sequence of stages a transaction passes from being first introduced to a client until it is accepted into a party's local state. These stages are encoded in $\mathsf{buffer}_{\mathsf{TxSig}}$. The largest part of the protocol is then the bookkeeping needed to implement the state machine. We discuss both the validation function and the sequence of stages in the following.

---

[16] Figure 23 shows the specification excerpt for clients. Notaries have very slight modifications which we describe in this section as well.
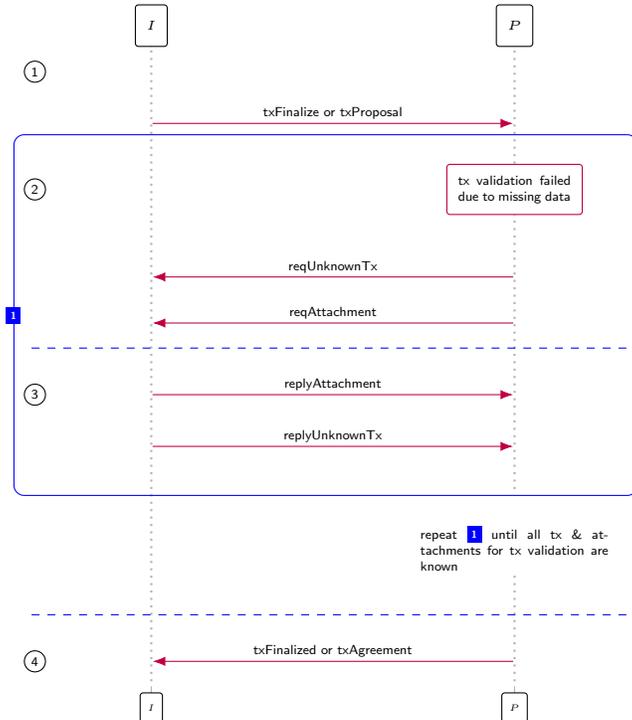
Fig. 22. A notary or another client P requests related transactions and attachments from an initiator I if P is not aware of them.

Path to transaction validity Clients learn about transaction proposals from three paths. They may get the transaction from the network if a peer wishes to send it to them or from the environment via a call to submit where they are either initiator or signee of that transaction. Depending on how a client first learned about a transaction a different sequence of stages are passed until the transaction is finally accepted as valid.

In what follows, we describe the stages of a transaction in Corda more detailed. The simplest case concerns transactions that are send 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 pro-actively. Transactions in `requested` stage directly pass to the set of valid transactions verifiedTx once successfully validated.

If a transaction is submitted to a client acting as a signee, 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, validated and – if successfully validated – marked as `sign`. In this stage a signature is created and send to the initiator and the transaction proceeds to the stage `expNotarization`. Finally, once a notary signature is received from the initiator, the transaction is accepted into verifiedTx.

On initiator side, submitted transactions are marked as `expColSigs` and validated. Once validated they enter into `reqSigs` and the signees are asked for signatures. The transaction stays in this stage until all necessary signatures are received at which point it is marked as `reqNotarize` and send to the notary for notarization. Once the response from the notary is received and validated, the transaction is accepted into verifiedTx and forwarded to all signees to notify them about the notary signature.

If in the previously mentioned processes a transaction in the buffer can not be resolved but needs to for validation the status is set to `reqDeps` to prevent asking for the dependencies of this particular transaction several times.

Main (excerpt):

function validate(tx, Σ, pid) :
    if verifiedTx.contains($\_, tx, \_, \Sigma', \_$):
        if $|\Sigma'| == m + 2$:
            return alreadyNotarised
    else:
        if isDoubleSpend(tx, verifiedTx):
            return doubleSpend
    if $\neg [\forall\ i \in [m] : \text{isValidId}(signee_i, \mathtt{client}) == \mathtt{true}\ \wedge \text{isValidId}(initiator, \mathtt{client}) == \mathtt{true}\ \wedge$
      $\text{isValidId}(formerNotary, \mathtt{notary}) == \mathtt{true}\ \wedge \text{isValidId}(notary, \mathtt{notary}) == \mathtt{true})]$:
      return invalid
    if $\neg [(|\Sigma| \le m + 2\ \wedge\ \exists i \in [m] : signee_i == pid_{\mathsf{cur}} \vee initiator == pid_{\mathsf{cur}})$
      $\vee (\forall\ i \in [m] \exists_1 (\sigma', signee_i) \in \Sigma \wedge (|\Sigma| == m + 1 \vee (|\Sigma| == m + 2\ \wedge\ \exists(\sigma', formerNotary) \in \Sigma)))]$:
      return invalid
    for all $(\sigma, pid_\sigma) \in \Sigma$ **do**:
      send (tx, σ) **to** $(pid_\sigma, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{cert}} : \mathtt{verifier})$
      wait for (VerResult, res)   *{Checks all provided signatures*
      if $res == \mathtt{false}$:
        return invalid
    if isResolvable(tx) == false $\wedge\ \text{buffer}_{\mathsf{TxSig}}[tx][4]! = \mathtt{reqDeps}$:
      $msg \leftarrow \text{resolveTx}(tx, pid)$
      send (Message, msg) **to** $(pid_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{unicast}} : \mathtt{unicast})$
    else if isResolvable(tx):
      $txDependencies \leftarrow \epsilon$
      for all $(txId', outputIndex')$ **do**:
        if verifiedTx.contains($(\_, tx, \_, \_, \_)$):
          $txDependencies.\text{add}(\text{verifiedTx}[txId'][1])$
        else:
          $txDependencies.\text{add}(\text{buffer}_{\mathsf{TxSig}}[txId'][1])$
      for all $attachmentId \in proposal$ **do**:
        $txAttachments.\text{add}(attachments[attachmentId])$
      $res \leftarrow \text{executeValidation}(tx, txDependencies, txAttachments)$
      return res

function resolveTx($tx, pids$) (msg):
    $\text{buffer}_{\mathsf{TxSig}}[tx][4] \leftarrow \mathtt{reqDeps}$
    $msg \leftarrow \epsilon$
    for all $(txId, outputIndex) \in tx_{inputStates} \cup tx_{referenceStates}$ **do**:
      if $\nexists(\_, tx', \_) \in \text{verifiedTx} \wedge (txId, outputIndex) \in tx'_{outputStates}$:
        $\text{buffer}_{\mathsf{TxSig}}.\text{add}((txId, \bot, \bot, pids, \mathtt{requested}))$
        $msg.\text{add}((pids, (\mathtt{SendTransactionFlow}, (txId, outputIndex))))$
    for all $attachmentId \in proposal$ **do**:
      if $(attachmentId, \_) \notin attachments$:
        $msg.\text{add}((pids, (\mathtt{GetAttachment}, (attachmentId))))$
    return msg

function validateAllLocal() :
    $identities \leftarrow \epsilon, msg \leftarrow \epsilon, resolved \leftarrow \mathtt{true}$
    while $resolved == \mathtt{true}$ **do**
      $resolved \leftarrow \mathtt{false}, res \leftarrow \epsilon$
      for all $(\_, tx, \Sigma, pid, status) \in \text{buffer}_{\mathsf{TxSig}} \mid status \notin \{\bot, \mathtt{unrequested}, \mathtt{expNotarization}, \mathtt{reqNotarization}\}$ **do**:
        if isResolvable(tx):
          $res \leftarrow \text{validate}(tx, \Sigma, \epsilon)$
          if $res == \mathtt{valid}\ \wedge (status == \mathtt{sign} \vee status == \mathtt{expColSigs})$:
            send (Sign, tx) **to** $(pid_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{cert}} : \mathtt{signer})$
            wait for (Signature, σ)
          else if $res == \mathtt{valid}\ \wedge \text{isNotarised}(tx)$:
            verifiedTx.add($(txId, tx, \emptyset)$)
          if $status == \mathtt{sign}$:
            $msg.\text{add}((pid_{\mathsf{cur}}, (\mathtt{Signature}, (tx, \sigma))))$
          else if $status = \mathtt{expColSigs}$:
            $msg.\text{add}((pid_{\mathsf{cur}}, \mathtt{CollectSignaturesFlow}(tx)))$
          $resolved \leftarrow \mathtt{true}$
          $\text{buffer}_{\mathsf{TxSig}}.\text{remove}(\_, tx, \Sigma, pid, \_)$
    send (Message, msg) **to** $(pid_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{unicast}})$

Fig. 23. The Corda model $\mathcal{P}^{\mathbf{c}}$, excerpt of the specification of the Corda client $M_{\mathtt{client}}$

Validity testing During the Corda protocol, clients and notaries execute several validations. As part of this process, the function `resolveTx()` is responsible for collecting all information necessary to complete validation of a transaction. Concretely, to validate a transaction all attachments need to be known and all transactions providing input and reference states need to be known and verified. To resolve unknown data,

`resolveTx()` creates the `GetAttachment` and `SendTransactionFlow` messages to be forwarded to the unicast interface.

`validateAllLocal()` acts as a housekeeping method. Every time new information is received this may have consequences for other transactions: If collection of their dependencies is completed they may be validated, if the signatures have arrived a transaction might be ready for notarization or a notary signature can settle the stage of a transaction as valid. Concretely the function implements a fixed point iteration where all resolved transactions (those with enough verified information) are validated and moved to verifiedTx until no more transaction can be processed. `validateAllLocal()` may send messages to the unicast channel in case clients acting as initiator or signee are satisfied with their validity.

The most complex function is `validate()` which is responsible for assessing the validity of a specific transaction. It optionally takes a *pid* in which case it will use `resolveTx()` to ask for missing dependencies. Alternatively, if no *pid* is provided, it only assesses validity if all necessary information is present. In a first step, `validate` checks whether the transaction has already been accepted (and is therefore in verifiedTx) or qualifies as double spend. 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 notaries are allowed to be mentioned in the notary field). Further, `validate` checks if the correct number of signatures exists. We define m to be the number of signees. If $\mathsf{pid_{cur}}$ is signee or initiator of the transaction the number of signatures might be smaller then $m + 2$ as this transaction might still be in signing process. Otherwise either there have to be signatures for every signee and the initiator or additionally for the notary. Finally, if dependencies or attachments are missing, they are optionally requested which ends the activation. If all checks verify, all inputs to the contract (or attachment) verification algorithm `executeValidation` are collected and the contract validated. Contract validation receives all (direct) input and reference states for the current transaction as well as their attachments but has no access to the full dependency tree or other transactions in the client's state.

Differences for notaries Notaries act similar to clients in that they receive transactions which pass several stages in $\mathsf{buffer_{TxSig}}$ until they get accepted into verifiedTx. However there are several noteworthy differences: Notaries never receive dependencies pro-actively and never take the active role of an initiator but act like a signee. However notaries always "accept" transactions: They receive transactions for notarization from the initiator instead of first receiving them as approved transactions from the environment and forwarding them upon request by the initiator. Finally once they have signed a transaction they can directly add them to verifiedTx whereas signees need to wait for a notary signature (e. g., to handle double spending). For this reasons the stages for the notary are only `requested` and `sign`, which conceptually correspond to the states with the same name for the client. Besides this a notary only needs to check in `validate` if there exists exactly one signature for every signee and the initiator as otherwise the transaction is invalid for notarisation.

Note that we model Corda transactions as $tx = (initiator, [signee_1, \ldots, signee_m], notary, formerNotary, proposal)$ (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.

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

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

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

Having explained both the ideal protocol $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ (cf. Section 4.3) and its intended realization $\mathcal{P}^{\mathbf{c}}$, we can now formally state the main result of this section (cf. Figure 4 in Section 4.3).

Before going into details, we first define different knowledge notations and the notion of synchronized states we use later on. Jumping slightly ahead, we use following abbreviations from Theorem 6: $\mathcal{R}$ denotes

the real protocol, i.e., $\mathcal{P}^{\boldsymbol{c}}$, $\mathcal{I}$ denotes the ideal protocol $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{ledger}}$, 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 to the induction hypothesis that the "states" of $\mathcal{R}'$ 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 in time 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 $\mathcal{I}$.[17] 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 $txIDs^{\mathcal{R}}_{buffered}$ the transaction IDs extracted from $\mathsf{buffer}_{\mathsf{TxSig}}$ from the entities state in $\mathcal{R}'$. Let $txIDs^{\mathcal{I}}$ be the set of transaction IDs such that

- $txIDs^{\mathcal{I}}$ is the set of transaction IDs extracted from $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{ledger}}$'s message list in $\mathcal{I}$ as follows: *(i)* For all transactions in msglist where *entity* is initiator, signee, or notary of the transaction, add the transaction ID to $txIDs^{\mathcal{I}}$. *(ii)* For all push operations where $entity = (pid, sid, role)$ is the receiver of the operation, i.e., for all message list entities that contain a message of the form $(txID, \mathsf{pids}, pid)$, add $txID$ to $txIDs^{\mathcal{I}}$. *(iii)* For all transactions in *requestQueue* where *entity* is initiator or signee (or notary) of the transaction. If there exists a submit transaction of this transaction submitted by the transactions initiator in *requestQueue*, then add the transaction to $txIDs^{\mathcal{I}}$. *(iv)* Do recursively: for all transaction IDs in $txIDs^{\mathcal{I}}$, add their input transaction IDs from msglist to $txIDs^{\mathcal{I}}$. *(v)* All leaked transactions/transactions generate by $\mathcal{A}$ are in $txIDs^{\mathcal{I}}$ as they are not private.
- $attachmentIDs^{\mathcal{I}}$ is the set of all attachment IDs extracted from $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{ledger}}$'s msglist and *requestQueue* such that $(pid, sid, role)$ submitted the attachment to $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{ledger}}$. Additionally, for all transactions in $txIDs^{\mathcal{I}}$, extract the attachment IDs the transactions and add them to $attachmentIDs^{\mathcal{I}}$.
- $txIDs^{\mathcal{I}}_{buffered}$ is the set of transaction IDs extracted from $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{ledger}}$'s *requestQueue*.

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:

$$txIDs^{\mathcal{R}} \subset txIDs^{\mathcal{I}} \,\wedge$$

$$attachmentIDs^{\mathcal{R}} \subset attachmentIDs^{\mathcal{I}}$$

To allow better referencing in what follows: the following sets $txIDs^{\mathcal{R}}(entity), attachmentIDs^{\mathcal{R}}(entity)$, resp. $txIDs^{\mathcal{I}}(entity), attachmentIDs^{\mathcal{I}}(entity)$ are defined as above but explicitly referencing the owner *entity*.

We call the $txIDs^{\mathcal{R}}(entity)$ together with $attachmentIDs^{\mathcal{R}}(entity)$ the *real knowledge* of *entity*, resp. the *potential knowledge* of *entity* in $\mathcal{R}'$, resp. $\mathcal{I}$ (at a point during the run of $\{\mathcal{S}, \mathcal{I}\}$).

We call the intersecting set of a set of known transactions of *entity* from $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{read}}$ ($knowTransactions(entity) = \{txID \mid (pid, txID) \in \mathsf{knownTransaction}\}$) and the intersecting set of an *entity*'s set of known attachments from $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{read}}$ ($knowAttachments(entity) = \{id^a \mid (pid, id^a) \in \mathsf{knownAttachments}\}$) as *active* knowledge of *entity* in $\mathcal{I}$.

We call the set of all transactions and attachment id that an entity had access to before and including the current activation the *current knowledge* of the entity at this point during the run.

Theorem 6. *Let $\eta \in \mathbb{N}$ be the security parameter and $\Sigma = (\mathsf{gen}(1^{\eta}), \mathsf{sig}, \mathsf{ver})$ be an EUF-CMA secure signature scheme. Let $\mathcal{P}^{\boldsymbol{c}}$ be the Corda protocol that uses the signature scheme Σ, using parameters/further algorithms as defined in Figure 24 and further parameters selected arbitrarily such that all parameterized algorithms are deterministic and in polynomial time. Let $\mathcal{F}^{\boldsymbol{c}}_{\mathrm{ledger}}$ be the instantiation of $\mathcal{F}_{\mathrm{ledger}}$ as described above, where the internal subroutines use the same parameters as $\mathcal{P}^{\boldsymbol{c}}$. Then:*

$$\mathcal{P}^{\boldsymbol{c}} \leq \mathcal{F}^{\boldsymbol{c}}_{\mathrm{ledger}}$$

---

[17] We do not expect equality of both state as $\mathcal{I}$ abstracts network delay during communication and might have access to data which still needs to be delivered in $\mathcal{R}'$.

Proof. As part of the proof, we firstly define a responsive simulator $\mathcal{S}$ such that the real world running the protocol $\mathcal{R} := \mathcal{P}^{\mathbf{c}}$ is indistinguishable from the ideal world running $\{\mathcal{S}, \mathcal{I}\}$, with the protocol $\mathcal{I} := \mathcal{F}^{\mathbf{c}}_{\text{ledger}}$, 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 (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 give 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/overview regarding submitted transactions and read operations of honest entities. Nevertheless, $\mathcal{S}$ follows the same approach as in Theorem 4: $\mathcal{S}$ internally simulates $\mathcal{R}$. In contrast to Theorem 4, $\mathcal{S}$ has to use the data leaked from $\mathcal{F}_{\text{ledger}}$ to simulate a "blinded" version of $\mathcal{R}$. $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ leaks many details of a transaction (except the transaction's full plain text) including *(i)* the transaction's id, *(ii)* the length of the transaction, *(iii)* initiator entity, signee entities, and (former) notary entity, *(iv)* transaction inputs and outputs (only ids), and *(v)* used attachments (only ids). During the simulation, $\mathcal{S}$ replaces the original (blinded) transaction in the simulation by a unique dummy transaction (identified by the original transaction ID) in the expected format enriched with data from $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$'s leakage. The dummy transactions includes all the leaked data such that $\mathcal{R}'$ can directly execute format checks on the dummy transaction. The unknown part of the dummy transaction is filled with random bit strings such that the length of the dummy transaction matches the length of the original transaction. For attachments, $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ 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 this unique dummy transaction.

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

Firstly, after corruption of an entity, $\mathcal{S}$ has to provide the appropriate leakage to an adversary $\mathcal{A}$. This is mainly done by keeping corruption in $\mathcal{R}'$ and $\mathcal{I}$ in sync (see below). When corrupting an entity *entity* in $\mathcal{I}$, $\mathcal{S}$ gets full access to the entity's internal state which includes *(i)* the attachments known by *entity*, *(ii)* transaction subgraphs in which *entity* is involved in (from the transaction *entity* is directly involved in 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 an other entity (also following the direction of the input transactions in the subgraph).[18] As soon as $\mathcal{F}_{\text{ledger}}$ leaks details regarding attachments or transactions $\mathcal{S}$ already is aware of (as he knows the ids), he replaces the dummy transaction in $\mathcal{R}'$ by the actual transaction/attachment data.[19] Then, $\mathcal{S}$ extracts the leaked data for the corrupted party from $\mathcal{R}'$.

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

Thirdly, if $\mathcal{R}'$, resp. a corrupted entity, uses the random oracle $\mathcal{F}_{\text{ro}}$ to generate a fresh transaction or attachment id to retrieve an id, $\mathcal{S}$ checks whether the query is a transaction or an attachment. If it is one of both, $\mathcal{S}$ forwards the request to $\mathcal{F}_{\text{ro}}$ as Update[GetID] message to $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$. $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ will request $\mathcal{S}$ for a unique id if $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ is not aware of the requested bit string. Otherwise and after setting the id, $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ will provide the id to $\mathcal{S}$. The simulator forwards the id as answer of $\mathcal{F}_{\text{ro}}$ to $\mathcal{R}'$ or to the corrupted entity which queried

---

[18] The leaked data is an over-approximation of the corrupted entity's knowledge where instant delivery of dependent transactions and attachments is assumed.

[19] Replacing blinded data by original data in particular includes the regeneration and replacement of all signatures in $\mathcal{R}'$ which were generated based on the blinded data.

$\mathcal{F}_{\mathrm{ro}}$. If the request to $\mathcal{F}_{\mathrm{ro}}$ 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 $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$) 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 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 $\mathcal{E}$.

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 of $\mathcal{F}_{\mathrm{ledger}}$ in $\mathcal{I}$ before continuing its 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, $\mathcal{S}$ extracts the leakage during corruption after it applied the leakage from $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ to $\mathcal{R}'$.
- Incoming Messages from corrupted entities of $\mathcal{F}_{\mathrm{ledger}}$ in $\mathcal{I}$ are forwarded on the network to the environment in the name of the corresponding entity in $\mathcal{R}'$. Conversely, whenever a corrupted entity of $\mathcal{R}'$ wants to output a message to a higher-level protocol, $\mathcal{S}$ instructs the corresponding entity of $\mathcal{F}_{\mathrm{ledger}}$ to output the same message to the higher-level protocol.
- Messages from corrupted parties to $\mathcal{F}_{\mathrm{ro}}$ are mapped to Update with "flavor" GetID] commands if the corrupted party queries $\mathcal{F}_{\mathrm{ro}}$ for transaction or attachment IDs. $\mathcal{S}$ forwards them to $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$. Responses of $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ are mapped to the answer format of $\mathcal{F}_{\mathrm{ro}}$, and returned to the corrupted party (see above). Otherwise, the requests are handled by $\mathcal{S}$ as explained above.
- Messages from corrupted parties to $\mathcal{F}_{\mathrm{unicast}}$ are forwarded to the simulation $\mathcal{R}'$.

Current state queries to $\mathcal{S}$

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

Transaction submission Whenever an honest entity $entity = (pid, sid, role)$ receives a request $(\texttt{Submit}, msg)$ to submit a new transaction msg, the subroutine $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ processes the three possible message formats *(i)* a "common" transaction proposal, *(ii)* submission of an attachment, and *(iii)* transmission of an existing transaction from one party to another party and leaks appropriate data for $\mathcal{S}$ to distinguish the above operations.

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 in a submission request to $\mathcal{R}'$. In the case that the submit was triggered by one of the involved entities except the initiator and one of the involved entities is corrupted $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ leaks the transaction to $\mathcal{S}$.[20] In the case that the submit was triggered by initiator (and one of the involved entities is corrupted), $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ 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 $\mathcal{F}_{\mathrm{unicast}}$. See below for details.

---

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

Case (ii): $\mathcal{S}$ triggers that the attachment is stored in the submitting entities state by sending the appropriate message $\texttt{Update}[\texttt{attachment}]$ to $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$. 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, 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 $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$ leaks data during Case (iii), it updates $\mathcal{R}'$ as explained above.

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

Read requests

Whenever an honest entity *entity* receives a request $(\texttt{Read}, msg)$ 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 to $\mathcal{R}'$. Note that the request is independent of the input within the read request. $\mathcal{S}$ extracts the transaction ids from the simulated read request and forwards the suggested output to $\mathcal{F}^{\mathbf{c}}_{\text{ledger}}$.

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

Further details

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

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

Now, let $\mathcal{E} \in \mathsf{Env}(\mathcal{R})$ 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 to distinguish $\mathcal{R}$ and $\mathcal{I}$ based on generated IDs and signatures.

Collisions of IDs: $\mathcal{R}$ uses a random oracle $\mathcal{F}_{\text{ro}}$ 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, singer, and signature pairs and verifying then at $\mathcal{F}_{\text{cert}}$. However, due to Σ's EUF-CMA security, the possibility to guess such a triple is negligible in η. Thus, the probability that $\mathcal{E}$ may distinguish between $\mathcal{R}$ and $\mathcal{I}$ based on a 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 was interactions on the network, resp. I/O, interface yet. Thus, the base case holds true. In the following, assume that all network, resp. I/O, interactions to far have resulted in the same behavior visible towards the environment in both the real and ideal world.

---

[21] As all algorithms are in polynomial time and $\texttt{executeValidation}$ ensures that the execution of attachments finishes in polynomial time.

Interaction with honest entities via I/O: Firstly, we show that the I/O behavior simulated by $\mathcal{S}$ towards the environment is indistinguishable from the I/O behavior of $\mathcal{R}$. For brevity and readability, we often use expressions like "$\mathcal{S}$ pushes the attachment/the 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 into an input to the environment but they might influence future read requests. Thus, the main goal here is to prove that $\mathcal{R}'$ and $\mathcal{I}$ stay synchronized after a submission of a transaction. Note that $\mathcal{S}$ can distinguish the three following cases du to the format of the leakage provided by $\mathcal{I}$.

Submission of a transaction 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.[22] 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 $txIDs_{buffered}^{\mathcal{R}}(entity)$) 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 $\mathcal{S}$). Note that – if the transactions initiators submission of the transaction is already processed and accepted – this step may add the transaction, its subgraph, and the used attachments to $txIDs^{\mathcal{I}}(entity)$, resp. $attachmentIDs^{\mathcal{I}}(entity)$. 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 the query with the current knowledge of the entity (in form of IDs) as explained above. As $\mathcal{R}'$ and $\mathcal{I}$ were in sync before the operation they stay synchronous after the operation as well (we will step over the different Corda flows below, which will 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 monotonically increasing and $\mathcal{S}$ keeps track of declined transactions and attachments: the current state delivered by $\mathcal{S}$ to $\mathcal{I}$ is always a super set of the previous current state. Thus, $\mathcal{I}$ will accept $\mathcal{S}$'s update. Further, the simulated entity only accepts pushed transactions from a corrupted entity if the corrupted entity can provide the full (valid) transaction subgraph below the transaction. That is, these transactions need to generated by corrupted parties or leaked. Thus, this check also does not fail.

As the knowledge of *entity* in $\mathcal{R}'$ is by induction hypothesis a subset of $\mathcal{I}$ knowledge before the activation we can thus conclude that $\mathcal{R}'$ ans $\mathcal{I}$ 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 $\mathcal{I}$ are still synchronized in this case (as real knowledge and potential knowledge does not change). Observe that the entity in $\mathcal{R}'$ process may push several transactions to other entities (via the `CollectSignaturesFlow`, see below). 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 at the end of the simulation in $\mathcal{S}$. Observe that the update received from $\mathcal{S}$ lead to the fact that real knowledge in $\mathcal{R}'$ is now equal to the active knowledge in $\mathcal{I}$.

2. In the case that $\mathcal{I}$ accepts the transaction. Observe that the entities potential knowledge in $\mathcal{I}$ does not change at this point in time. 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 equal 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 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.

---

[22] $\mathcal{S}$ can determine the role of the submitting entity as $\mathcal{I}$'s leakage includes the roles of the involved parties

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 $\mathcal{R}'$.

Submission of an attachment The leakage of $\mathcal{I}$ indicates whether a submitted attachment is valid. In case that 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 $\mathcal{R}'$ and $\mathcal{I}$ use the same (deterministic) algorithm to validate the attachments, $\mathcal{R}'$ will accept the attachment if $\mathcal{I}$ accepted it. $\mathcal{R}'$ adds the attachment immediately to the entity's real knowledge $attachmentsIDs^{\mathcal{R}}(entity)$. In this case, $\mathcal{S}$ pushes 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 $\mathcal{R}'$.

One entity shares a transaction with another entity Before $\mathcal{I}$' leakage indicates that entity $entity_a$ 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 now equal to the active knowledge in $\mathcal{I}$.

Observe that $\mathcal{I}$ leaks indicates that it accepted the push operation, the 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 same algorithms are running in the same input (as real knowledge and active knowledge are equal). 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 $\mathcal{R}'$. In this case, $\mathcal{S}$ immediately pushes the transaction (ID) via the command `Update[txExchange]` to the entity's potential knowledge in $\mathcal{I}$. Observe, that this operation adds all transaction from *msglist* which are in the transaction subgraph below the transaction $txId$ and all attachments mentioned in this subgraph to the entities potential knowledge. Thus, $\mathcal{R}'$ and $\mathcal{I}$ are synchronized after this operation.

Simulation of internal Corda protocol steps and $\mathcal{F}_{\text{unicast}}$: 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 $\mathcal{P}^{\mathbf{c}}$ in connection with the simulated network operation via $\mathcal{F}_{\text{unicast}}$ to distinguish between $\mathcal{R}$ and $\mathcal{I}$. By construction of $\mathcal{R}$ and $\mathcal{I}$, the simulation of $\mathcal{F}_{\text{unicast}}$ does not directly result into an input to the environment via I/O but they 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 $\mathcal{F}_{\text{unicast}}$ 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 $\mathcal{F}_{\text{unicast}}$ does not allow to send messages on behalf of other entities, i.e., $\mathcal{F}_{\text{unicast}}$ ensures an authenticated secure channel for communication.

Note that the leakage to the network generated by simulation of $\mathcal{F}_{\text{unicast}}$ is indistinguishable between $\mathcal{R}$ and $\mathcal{I}$ as $\mathcal{S}$'s dummy transactions have the same length as 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 on 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 operation To trigger the simulation of the Corda flow processing in $\mathcal{R}'$, $\mathcal{S}$ waits for the `Deliver` command from network. The command informs $\mathcal{S}$ by an $\mathcal{F}_{\text{unicast}}$ message id which message is to be delivered in $\mathcal{R}'$. Let $entity_s$ be the sender and $entity_r$ be the recipient of the message.

`SignTransactionFlow`: 1. The delivered message in the simulation was send 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 $txIDs^{\mathcal{R}}_{buffered}(entity_s)/txIDs^{\mathcal{I}}_{buffered}(entity_s)$. If there exists a message from $entity_r$ in

$txIDs^{\mathcal{R}}_{buffered}(entity_r)/txIDs^{\mathcal{I}}_{buffered}(entity_r)$ indicating that $entity_s$ approve the message (and 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 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 an execution of executeValidation is replaced by true if the transaction already passed the validation by $entity_s$ (otherwise false). In every case, $txIDs^{\mathcal{R}}_{buffered}(entity_s)$ does not change as either the transaction was already in the set or is declined and does not enter the set. In case of an 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 send back to $entity_s$.

   b) due to missing dependencies, $entity_r$ generates `SendTransactionFlow`, resp. `GetAttachment` requests which is send via $\mathcal{F}_{\mathrm{unicast}}$ back to $entity_r$.

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

2. The delivered message in the simulation was send by a corrupted entity $entity_s$ (such that the entity 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 usually possible as the corrupted party need 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.[23]

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` : 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 transaction mentioned in the message (again - note that $\mathcal{S}$ has access to this information due to leaked data).

   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, $\mathcal{S}$ has access to all necessary data to perform the simulation, executeValidation is replaced by the validity information leaked by $\mathcal{I}$.

   b) If the message from $entity_s$ indicates that it declines the transaction, $entity_r$ removes the transaction from her buffer $txIDs^{\mathcal{R}}_{buffered}(entity_r)$.

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` : Note that the 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 $entity_s$ (a is the initiator of the transaction), it processes the messages as explained in the cases above (executeValidation is replaced by leakage from $\mathcal{I}$, $\mathcal{S}$ has access to all necessary data as explained above). The outcome of the simulation may trigger *(i)* `SendTransactionFlow`, resp. `GetAttachment` : requests to $entity_s$ (via $\mathcal{F}_{\mathrm{unicast}}$ - see below) *(ii)* or $entity_r$ notarizes the transaction and `NotariseRes` message to $entity_s$ (if all checks and signatures in the simulation verify).

---

[23] Note 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.

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

- $\mathcal{S}$ immediately pushes attachments or transaction push messages to $\mathcal{I}$ msglist.
- transactions send to notarisation from honest initiators in $\mathcal{R}'$ are checked with a subset of the rules from the update process in $\mathcal{I}$, such as format, role checks, and double spend 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 due to the fact that 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 an 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: All involved entities in $\mathcal{I}$ 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 $entity_s$ - note that $\mathcal{S}$ has full access to all transaction necessary to execute the notarisation or she may request the output of executeValidation at $\mathcal{I}$. Thus, the simulation works as in the case above. Note that corrupted entities cannot forge signatures of other entities. Thus, we can follow the explanation from above. $\mathcal{S}$ 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 : 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 notarisation message. In the case that the simulation accepts the notarisation, the FinalityFlow is triggered. (Note that $\mathcal{S}$ 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 before 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, $\mathcal{S}$ 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 : 1. In the case that $entity_r$ receives the message from an uncorrupted entity $entity_s$ (the transaction initiator), there are all information available to simulate this process. If the entity $entity_r$ accepts the RecvFinalityFlow message, it adds the transaction to the entity's real knowledge. As the transaction was already before 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 $entity_s$[24] (and $entity_r$ is involved in the transaction): Note that $\mathcal{S}$ 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, $\mathcal{S}$ already pushed the transaction 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 & RecvTransactionFlow : According to the explanations above, $\mathcal{S}$ is able to 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, in the case that a corrupted party sends a transaction (graph) and the entity accepts it, the entity becomes corrupted (as this includes that a notary in this subgraph is corrupted). Thus, $\mathcal{R}'$ and $\mathcal{I}$ still stay synchronized (as the entity is not in the set of honest entities any longer).

---

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

`GetAttachment` & `RecvAttachment` : According to the explanations above, $\mathcal{S}$ is able to simulate this in any case. We emphasize here, that $\mathcal{S}$ is able to call executeValidation at $\mathcal{I}$ in the context of the receiving parties current state.[25] to validate the transaction in the context of the receiving party. 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 already a submission added the transaction and its 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 a honest entity *entity* receives a request $(\texttt{Read}, msg)$ to read from its restricted view on the global state , $\mathcal{I}$, resp. $\mathcal{F}_{\text{ledger}}$ forwards this (local) read request to $\mathcal{S}$ and waits to receive a suggested output. $\mathcal{S}$ simulates the read request internally. She extracts the transaction and attachment ids from the output and forwards them in the appropriate `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 as 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 subset (due to the specification of Corda).
2. The read output is always a subset of the entities current knowledge in $\mathcal{R}'$ (due to the specification of Corda).
3. The party is uncorrupted and the output in $\mathcal{R}'$ only consists of transaction notarised by uncorrupted notaries. Thus, all transactions from 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 transaction (subgraphs), potential additional transactions added to the entity's read output are already in leaked (see explanation during flow explanations).
5. Observe that $\mathcal{R}'$, resp. $\mathcal{F}_{\text{unicast}}$, enforces the upper bound of δ rounds for eventual message delivery. Further note that all involved entities for the mentioned transactions are honest, we can conclude that the 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: Let $subgraph(tx)$ the subgraph of the exchanged transaction (such that all input references are part of the graph, down to the issuance transactions). The receiving entity queries the full subgraph of the transaction before adding the transaction to its verifiedTx and thus outputs it during reading. As in the worst case, these are $|subgraph(tx)|$ objects to query (if the graph is a chain), 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 $txIDs^{\mathcal{I}}$, resp. $attachmentIDs^{\mathcal{I}}$ as the entity is then corrupted.

Corrupting parties: In the following, we argue that *(i)* $\mathcal{S}$ is always able to keep corrupting in $\mathcal{R}'$ and $\mathcal{R}$ 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 novel 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. $\mathcal{S}$ get access to the complete potential knowledge of an entity as soon it is corrupted. $\mathcal{S}$ replaces the dummy transactions in $\mathcal{R}'$ by the original transaction (including regeneration of signatures). In the case

---

[25] Note 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.

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

Interaction via network: First, observe that $\mathcal{I}$ provides $\mathcal{S}$ with 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 request. Thus, $\mathcal{S}$ 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 via network or to provide data in plain in case that corrupted parties are involved in a transaction or $\mathcal{S}$ can call $\mathcal{I}$ to validate transaction in the context of an entity. 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 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.

Current time requests: As the simulator updates the internal clock of $\mathcal{F}_{\text{ledger}}$ 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 $\mathcal{F}^{\mathbf{c}}_{\text{updRnd}}$:

Observe that $\mathcal{F}_{\text{unicast}}$ enforces that all messages are delivered in at most δ rounds in $\mathcal{R}$. $\mathcal{F}^{\mathbf{c}}_{\text{updRnd}}$ guarantees that a transaction tx in $requestQueue$, where all involved parties are honest, are part of the msglist after at most $(3 + 4 \cdot |subgraph(tx)| \cdot \delta)$ rounds (where $subgraph(tx)$ is the transaction subgraph below tx as explained above). As in the worst case scenario, the simulated notary does not know any dependencies of tx and the subgraph is a chain, we conclude that the bound from $\mathcal{F}^{\mathbf{c}}_{\text{updRnd}}$ 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 rounds to deliver the `CollectSignaturesFlow` messages from initiators to signees.
2. signees may request at most $|subgraph(tx)|$ dependencies from the initiator. Thus, delivery/processing needs at most $2 \cdot |subgraph(tx)| \cdot \delta$ 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 $|subgraph(tx)|$ dependencies from the initiator. Thus, delivery/processing needs at most $2 \cdot |subgraph(tx)| \cdot \delta$ 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 $(3 + 4 \cdot |subgraph(tx)| \cdot \delta)$ rounds after the last agreement was recorded. Thus, $\mathcal{I}$ will accept all time updates of $\mathcal{S}$.

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

Description of parameters/algorithms used in the protocols $M_{\texttt{client}}$ and $M_{\texttt{notary}}$:

---

executeValidation(tx, $txDependencies$, $txAttachments$) Among other things,
- it interprets attachments and checks if the transaction is valid with respect to these attachments,
- it ensures that attachments are determenestic and it enforces that the overall runtime of all attachments (and further validations) is in polynomial time
- it checks wether the transaction is contractually valid, i. e., fulfills the requirements given by the contracts for each state and for the transaction, but not valid signatures
- it checks whether all inputs have the same notary
- it checks whether all outputs have the same notary as the input notary or if the transaction is a notary change transaction.
- it checks that the owner of the transaction's *input* states are a subset of the transactions participants.
- it checks that the owner of the transaction's *output* states are a subset of the transactions participants.

validateAttachment($attachment$) This algorithm checks whether a single attachments 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 buffer$_{\texttt{TxSig}}$.

isValidId(pid, $role$) This algorithm checks whether the given pid belongs to a machine of the given $role$.

isDoubleSpend(tx) This algorithm checks whether the given tx is a double spend compared to transactions in the internal state verifiedTx, i. e., if there exists a transaction in verifiedTx that is either not the same transaction without notary signature but uses one of the input states of tx as well or spends a reference state.

isNotarised(tx) This algorithm checks whether either tx itself is notarised or it is contained in the dependency chain of a notarized transaction.

Fig. 24. Algorithms and parameters used by $M_{\texttt{client}}$ and $M_{\texttt{notary}}$

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

Participating roles: $\{\text{client}, \text{notary}\}$[a]
Corruption model: *Dynamic corruption without secure erasures*
Protocol parameters:
- executeValidation — $\{$*Algorithm for verifying a transaction.*
- networkmap — $\{$*Set of identities in network as* $(type, id)$
- validateAttachment — $\{$*Algorithm for validating attachments according to certain rules*

---

[a] To simplify presentation, we introduced $\mathcal{P}^{\mathbf{c}}$ as protocol $(\text{client} \mid \text{notary}, \mathcal{F}_{\text{unicast}}, \mathcal{F}_{\text{ro}})$ in Section 4.2. Formally, the Corda protocol is defined as $\mathcal{P}^{\mathbf{c}} = (\mathcal{P}^{\mathbf{c}}_{\text{ledger}} : \text{client} \mid \mathcal{P}^{\mathbf{c}}_{\text{ledger}} : \text{notary}, \mathcal{F}_{\text{unicast}}, \mathcal{F}_{\text{ro}})$.

Description of M_{client}:

Implemented role(s): $\{\text{client}\}$
Subroutines: $\mathcal{F}_{\text{cert}} : \text{cert}, \mathcal{F}_{\text{unicast}} : \text{unicast}, \mathcal{F}_{\text{ro}} : \text{randomOracle}, \mathcal{P}^{\mathbf{c}}_{\text{ledger}} : \text{notary}$,
Internal state:
- verifiedTx $\subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*$, verifiedTx $= \emptyset$ $\{$*List of verified transactions, as* $(txId, tx, \sigma)$

- attachments $\subset \{0,1\}^* \times \{0,1\}^*$, attachments $= \emptyset$ $\begin{cases} \textit{Set of known attachments and identifiers as} \\ (attachmentId, attachment) \end{cases}$

- buffer$_{\text{TxSig}} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \times status$, buffer$_{\text{TxSig}} = \emptyset$ $\begin{cases} \textit{Transaction currently in process as} \\ (txId, tx, \Sigma, pid, status). \ \Sigma \textit{ is a set of tuples of the} \\ \textit{form } (pid, \sigma), \ status \in \{\bot, \text{unrequested}, \text{requested},\} \\ \text{approved}, \text{expColSigs}, \text{reqSigs}, \text{reqDeps}, \text{sign}, \\ \text{expNotarization}, \text{reqNotarization}\} \end{cases}$

- buffer$_{\text{ReqValid}} \subset \{0,1\}^* \times \{0,1\}^*$, buffer$_{\text{ReqValid}} = \emptyset$ $\begin{cases} \textit{Set of transaction and attachments allowed to be queried as} \\ (txId/attachmentId, allowedRequesters). \textit{ Thereby } \text{buffer}_{\text{ReqValid}} \textit{ is a} \\ \textit{ordered set.} \end{cases}$

CheckID$(pid, sid, role)$:
    Accept all messages with the same sid.
Corruption behavior:
    DetermineCorrStatus$(pid, sid, role)$:
       if corr $==$ true: $\{$*Checks whether client itself is corrupted.*
          return true
       $corrRes \leftarrow \mathbf{corr}(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{signer})$ $\{$*Request corruption status at* $\mathcal{F}_{\text{cert}}$
       if $corrRes ==$ true: $\{$*Checks whether* $\mathcal{F}_{\text{cert}}$ *instance is corrupted*
          return true
       for all $(\_, tx, \_) \in$ verifiedTx $\cup (\_, tx, \_, \_, \text{reqNotarization}) \in$ buffer$_{\text{TxSig}} \cup (\_, tx, \_, \_, \text{expNotarization}) \in$ buffer$_{\text{TxSig}}$ **do:**
          $tx \leftarrow (\_, \_, \_, formerNotary, \_)$ $\begin{cases} \textit{Checks if at least one of all notaries used for notarisation in a tx either for transactions in} \\ \text{verifiedTx } \textit{or transactions expected to be notarized in } \text{buffer}_{\text{TxSig}} \textit{ is corrupted} \end{cases}$
          $corrRes \leftarrow \mathbf{corr}(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{signer})$
          if $corrRes ==$ true:
             return true

See Appendix A for notation details. We expect the ITM to enforce the transaction format $tx = (initiator, [signee_1, \ldots, signee_m],$ $notary, formerNotary, proposal)$ where in $proposal$ input states as $tx_{inputStates} = (txId, outputIndex)$ and output states as $tx_{outputStates}$. We denote by m the number of signees in a transaction. For tuples, lists, etc. we start index counting at 0.

Main:

    recv (Read, msg) **from** I/O: $\{$*Reading from Corda's transaction graph*
       $shortendVerifiedTx \leftarrow \emptyset$
       for all $(txId, tx, \sigma) \in$ verifiedTx **do:**
          $shortendVerifiedTx.\text{add}((txId, tx))$
       reply (Read, attachments, $shortendVerifiedTx$) $\{$*Output the transaction graph of pid excluding "internal" signatures*

    recv (Submit, msg) **from** I/O **s.t.** $msg = (\text{attachment}, attachment)$: $\{$*Client gets a new attachment via I/O*

       send $attachment$ **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ro}} : \text{randomOracle})$ $\begin{cases} \textit{Generates an attachment id and stores the new attach-} \\ \textit{ments needed for verification of tx} \end{cases}$
       wait for $attachmentId'$
       if buffer$_{\text{ReqValid}}.\text{contains}(\text{ATT}.\text{add}(attachmentId')) \wedge \text{validateAttachment}(attachment) ==$ true:

          attachments.add$(\text{ATT}.\text{add}(attachmentId'), attachment)$ $\begin{cases} \textit{Add (labeled) attachment and attachment} \\ \textit{id to } \text{attachments} \end{cases}$

Fig. 25. The Corda model $\mathcal{P}^{\mathbf{c}}_{\text{ledger}}$, specification of the Corda client M_{client} (Part 1)

Main:

   recv (Submit, msg) **from** I/O **s.t.** $msg = (\text{tx}, tx)$:
     send tx **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ro}} : \text{randomOracle})$
     wait for $(txId')$ *{Generate tx ID*

     $txId \leftarrow \text{TX.add}(txId')$ *{TX.add$(txId')$ concatenates $txId'$ with a label make the id identifiable as a transaction id.*
     if $initiator == \text{pid}_{\text{cur}}$:   *{initiator starts Corda's flow processing*

        if $\forall (txId', outputIndex') \in tx_{inputStates}$:   *{All direct dependencies are known*
          verifiedTx.contains$(txId, tx', \_) \mid outputIndex' \in tx'_{outputStates}$:
           $\text{buffer}_{\text{TxSig}}.\text{add}((txId, tx, \emptyset, \epsilon, \text{expColSigs}))$
           if validate$(tx, \emptyset, \epsilon) == \text{valid}$:   *{For validate definition see below*
             for all $(txId', outputIndex') \in tx_{inputStates}$ **do:**
               $\text{buffer}_{\text{ReqValid}}[txId'][1].\text{add}(\{signee_1, \ldots, signee_m, formerNotary\})$
             for all $attachmentId \in proposal$ **do:**
               $\text{buffer}_{\text{ReqValid}}[attachmentId][1].\text{add}(\{signee_1, \ldots, signee_m, formerNotary\})$
               {States and attachments are only marked valid for request if tx itself is valid. Rekursive defintion of objects allowed
               {to request in SendTransactionFlow.
             $msg \leftarrow \text{CollectSignaturesFlow}(tx)$ *{Definition of CollectSignaturesFlow in Figure 28*
             send (Message, msg) **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}} : \text{unicast})$
           else:
             $\text{buffer}_{\text{TxSig}}.\text{remove}((\_, tx, \_, \_, \_))$ *{Clean up declined transaction*
             validateAllLocal()   *{Check whether there are open task, e. g., finalizing a transaction*
     else if $\exists i \in [m] : signee_i == \text{pid}_{\text{cur}}$:   *{As signee: record tx as "will approve CollectSignaturesFlow"*
        $\text{buffer}_{\text{TxSig}}.\text{add}((txId, tx, \emptyset, \epsilon, \text{approved}))$

   recv (Submit, msg) **from** I/O **s.t.** $msg = (\text{tx}, txId, pid_{recv})$:   *{$\text{pid}_{\text{cur}}$ pushes/shares a verified transaction*
     $msg' \leftarrow \epsilon$ *{to/with pid_{recv}*
     if verifiedTx.contains$((txId, \_, \_))$:
       $msg' \leftarrow (\text{verifiedTx}[txId][1], \text{verifiedTx}[txId].[2])$ *{$msg' = (tx, \Sigma)$*
     else if $\text{buffer}_{\text{TxSig}}.\text{contains}((txId, \_, \_, \_, status)) \mid status \notin \{\perp, \text{unrequested}, \text{requested}, \text{reqDeps}\}$:
       $msg' \leftarrow (\text{buffer}_{\text{TxSig}}[txId][1], \text{buffer}_{\text{TxSig}}[txId][2])$ *{$msg' = (tx, \Sigma)$*
     if $msg' \neq \epsilon$:
       send (Message, $\{(pid_{recv}, (\text{RecvTransactionFlow}, msg'))\}$) **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}} : \text{unicast})$

   recv GetCurRound: *{$\mathcal{A}$ and $\mathcal{E}$ are allowed to query the clock*
                                                {Forward requests about current round to the uni-
     send GetCurRound **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}} : \text{unicast})$ *{cast channel*
     wait for (GetCurRound, $round$)
     reply (GetCurRound, $round$)

   recv (SignTransactionFlow, msg) **from** $(\text{pids}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}} : \text{unicast})$ **s.t.** $msg = (tx, \sigma)$:
   On request of a transaction initiator who started a CollectSignaturesFlow, a client signs a transaction if it agrees on the tx.
     if $\text{buffer}_{\text{TxSig}}[tx].\text{contains}(\_, tx, \emptyset, \epsilon, \text{approved}) \wedge initiator == \text{pids}$:
       send (Verify, tx, σ) **to** $(initiator, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{cert}} : \text{verifier})$
       wait for (VerResult, res)   *{Check initiators signature*

       if $res == \text{valid}$:   *{Transaction tx is already stored in $\text{buffer}_{\text{TxSig}}$. It is now marked as requested*
         $\text{buffer}_{\text{TxSig}}[tx][3] \leftarrow initiator$ *{for signature and the request (=initiator) is stored for possibly later answer*

         $\text{buffer}_{\text{TxSig}}[tx][4] \leftarrow \text{sign}$ *{Update processing status*

         $(txId, tx, \Sigma, pid, \text{sign}) \leftarrow \text{buffer}_{\text{TxSig}}[tx]$
         if validate$(tx, \Sigma, pid) == \text{valid}$:   *{For definition of validate, see Figure 28*
            send (Sign, tx) **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{cert}} : \text{signer})$
           wait for (Signature, σ)   *{Generate signature for initiator*
            $\text{buffer}_{\text{TxSig}}[tx][4] \leftarrow \text{expNotarization}$ *{Update processing status*
            reply (Message, $\{(initiator, (\text{Signature}, (tx, \sigma)))\}$)
       else:
         reply (Message, $\{(initiator, (\text{Signature}, (tx, \perp)))\}$)

   recv (Signature, msg) **from** $(\text{pids}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}} : \text{unicast})$ **s.t.** $msg = (tx, \sigma)$:
   Collect signed responses from the CollectSignaturesFlow. tx format needs to be as defined above

     if $\text{buffer}_{\text{TxSig}}.\text{contains}(\_, tx, \_, \_, \text{reqSigs}) \wedge$ *{Only requested signatures, i. e., for transactions in*
       $\text{pids} \in signee_{[m]} \wedge (\text{pids}, \_) \notin \text{buffer}_{\text{TxSig}}[tx][2]$:   *{$\text{buffer}_{\text{TxSig}}$ where this client is initiator, are accepted.*
       send (Verify, tx, σ) **to** $(\text{pids}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{cert}} : \text{verifier})$
       wait for (VerResult, $verified$)   *{Check signee signature*
       if $verified == \text{true}$:
         $\text{buffer}_{\text{TxSig}}[txId][2].\text{add}((\sigma, \text{pids}))$
         if $|\text{buffer}_{\text{TxSig}}[txId][2]| == m + 1$:   *{Every signature is only stored once if valid, thus if numbers match*
                                         {all needed signatures are present.
           FinalityFlow(tx) *{Start FinalityFlow if all signees confirmed tx*
       else:
         $\text{buffer}_{\text{TxSig}}.\text{remove}((txId, tx, \_, \_))$ *{Remove rejected tx from buffer*

Fig. 26. The Corda model $\mathcal{P}_{\text{ledger}}^{\mathbf{c}}$, specification of the Corda client M_{client} (Part 2)

Main:

recv (NotariseRes, msg) **from** (pids, sid$_{\text{cur}}$, $\mathcal{F}_{\text{unicast}}$: unicast) **s.t.** $msg = (\text{pids}, (tx, \sigma))$:

Handle tx finalization from notary and distribute result among tx signees
    if (buffer$_{\text{TxSig}}[tx]$.contains($\_, tx, \_, formerNotary$, reqNotarization) $\Big\{$ *Only expected signatures from a*
       | pids = $formerNotary$):   *notary are distributed*

        buffer$_{\text{TxSig}}[tx][2]$.add($formerNotary, \sigma$) $\Big\{$ *This will go through as an honest client will only trigger this if* pid$_{\text{cur}}$
        $res \leftarrow$ validate(tx, buffer$_{\text{TxSig}}[tx][2], \epsilon$)   *signed and therefore validated tx. For definition of* validate *see Fig-*
        if $res ==$ valid: *ure 28*
           verifiedTx.add(buffer$_{\text{TxSig}}[tx][0]$, buffer$_{\text{TxSig}}[tx][1]$, buffer$_{\text{TxSig}}[tx][2]$)
        buffer$_{\text{TxSig}}$.remove(buffer$_{\text{TxSig}}[tx]$)
        for all $i \in [m]$ **do:**
           msg.add(($signee_i$, (RecvFinalityFlow, (tx, Σ.add($formerNotary, \sigma$))))) $\{$*Distribute notarisation to signees*
        reply (Message, msg)

recv (RecvFinalityFlow, msg) **from** (pids, sid$_{\text{cur}}$, $\mathcal{F}_{\text{unicast}}$: unicast) **s.t.** $msg = (\text{pids}, (tx, \Sigma))$:
Called from initiator pids *after* FinalityFlow *is finished. Stores tx with signing result from notary*
    if buffer$_{\text{TxSig}}[tx]$.contains($\_, tx, \_$, pids, expNotarization):
        if $|\Sigma| = m + 2 \wedge$ validate(tx, Σ, ϵ) == valid: $\Big\{$*Assure still everything is valid and enough signatures are*
           verifiedTx.add(($txId, tx, \Sigma$))   *provided*
        buffer$_{\text{TxSig}}$.remove(buffer$_{\text{TxSig}}[txId]$)

recv (SendTransactionFlow, msg) **from** (pids, sid$_{\text{cur}}$, $\mathcal{F}_{\text{unicast}}$: unicast) **s.t.** $msg = (txId)$:
Client processes/forwards a requested transaction
    if buffer$_{\text{ReqValid}}[txId][1]$.contains(pids): $\{$*Checks whether pid is allowed to request transaction tx*
        buffer$_{\text{ReqValid}}[txId][1]$.remove(pids) $\{$*Decline further access*
        if verifiedTx.contains($txId, \_, \_$):
           $tx \leftarrow$ verifiedTx$[txId][1]$ $\{$verifiedTx$[.][1] = tx$ *for answer*
        else:
           $tx \leftarrow$ buffer$_{\text{TxSig}}[txId][1]$ $\{$buffer$_{\text{TxSig}}[.][1] = tx$ *for answer*
        for all $(txId', outputIndex') \in tx_{inputStates}$ **do:** $\{$*Grant pids access to dependencies of tx*
           buffer$_{\text{ReqValid}}[txId']$.add(pids)
        for all $attachmentId' \in tx[4]$ **do:**
           buffer$_{\text{ReqValid}}[attachmentId']$.add(pids)
        $msg' = $ (RecvTransactionFlow, verifiedTx$[txId]$) $\{$*Forward tx details to* pids
        send (Message, $\{(\text{pids}, msg')\}$) **to** (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{unicast}}$: unicast)
    else:
        send (Message, $\{(\text{pids}, \bot)\}$) **to** (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{unicast}}$: unicast)

recv (GetAttachment, msg) **from** (pids, sid$_{\text{cur}}$, $\mathcal{F}_{\text{unicast}}$: unicast) **s.t.** $msg = (\text{pids}, attachmentId)$:
Answers request from notary for attachment needed for verification
    if buffer$_{\text{ReqValid}}[attachmentId]$.contains(pids):
        $msg' = $ (RecvAttachment, pids, attachments$[attachmentId]$)
        reply (Message, $\{\text{pids}, msg'\}$)
    else:
        reply (Message, $\{\text{pids}, \bot\}$)

recv (RecvTransactionFlow, (pids, (tx, Σ))) **from** (pids, sid$_{\text{cur}}$, $\mathcal{F}_{\text{unicast}}$: unicast):
Entity processes a requested transaction
    send tx **to** (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{ro}}$: randomOracle) $\{$*Get transaction id*
    wait for $(txId')$
    $txId \leftarrow$ TX.add($txId'$)
    if ($\neg$(buffer$_{\text{TxSig}}$.contains(($txId, \_, \_$, pids, $\_$)) $\wedge$ verifiedTx$[txId]$.contains(($txId, \_, \_$)))):
        buffer$_{\text{TxSig}}$.add(($txId, tx, \Sigma$, pids, unrequested))
    else if buffer$_{\text{TxSig}}$.contains(($txId, \bot, \bot$, pids, requested)):
        $res \leftarrow$ validate(tx, Σ, pids) $\{$*See definition of* validate *below*
        validateAllLocal()

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

recv DeRegister **from** I/O: $\{$ *"Dummy" interface as deregistration is not part of the Corda model*
    wait for ack $\{$DeRegister *always returns ack*
    reply DeRegister

Fig. 27. The Corda model $\mathcal{P}^{\mathbf{c}}_{\text{ledger}}$, specification of the Corda client M_{client} (Part 3)

Procedures and Functions:

function CollectSignaturesFlow(tx) $(tx = (\_, [signee_1, \dots, signee_m], \_, \_, \_))$:
  Collects signatures for a transaction from all necessary participants

  $\mathsf{buffer}_{\mathsf{TxSig}}[tx][4] \leftarrow \texttt{reqSigs}$ $\qquad \left\{ \begin{array}{l} \textit{Transaction is marked in buffer as } \texttt{CollectSignaturesFlow } \textit{has been started an sig-} \\ \textit{natures form co-signees are expected} \end{array} \right.$

  send $(\texttt{Sign}, tx)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathsf{cert}} : \texttt{signer})$ $\qquad \{ \textit{Initiator signs transaction proposal}$
  wait for σ
  $\mathsf{buffer}_{\mathsf{TxSig}}[tx][3].\mathsf{add}(\mathsf{pid}_{\mathsf{cur}}, \sigma)$ $\qquad \{ \textit{Record initiator signature}$
  $msg \leftarrow \epsilon$ $\qquad \{ \textit{Prepare message to } \mathcal{F}_{\mathsf{unicast}}$
  for all $i \in [m]$ **do:**
    $msg.\mathsf{add}((signee_i, (\texttt{SignTransactionFlow}, (tx, \sigma))))$
  return msg

function FinalityFlow(tx, Σ) $(tx = (\_, \_, \_, formerNotary, \_))$: $\qquad \left\{ \begin{array}{l} \textit{Corresponds with notary to get signature and distribute} \\ \textit{signature to participants} \end{array} \right.$

  $\mathsf{buffer}_{\mathsf{TxSig}}[tx][4] \leftarrow \texttt{reqNotarization}$ $\qquad \{ \textit{Transaction is marked as waiting for notarisation in } \mathsf{buffer}_{\mathsf{TxSig}}$
  $msg \leftarrow \texttt{validateAllLocal}()$

function validate(tx, Σ, pid) : $\qquad \left\{ \begin{array}{l} \textit{Validates a transaction and produces message for requesting transactions} \\ \textit{and attachments if necessary.} \end{array} \right.$
  if $\mathsf{verifiedTx}.\mathsf{contains}(\_, tx, \_, \Sigma', \_)$:
    if $|\Sigma'| == m + 2$:
      return alreadyNotarised
  else:
    if isDoubleSpend$(tx, \mathsf{verifiedTx})$: $\qquad \left\{ \begin{array}{l} \textit{Checks whether one of the input states is already used in another tx} \\ \textit{in } \mathsf{verifiedTx} \end{array} \right.$
      return doubleSpend
  if $\neg\big[\forall\, i \in [m] : \texttt{isValidId}(signee_i, \texttt{client}) == \texttt{true} \wedge$
    $\texttt{isValidId}(initiator, \texttt{client}) == \texttt{true} \wedge$ $\qquad \left\{ \begin{array}{l} \texttt{isValidId } \textit{checks whether the given pid is regis-} \\ \textit{tered with the particular role in } \mathsf{networkmap}. \end{array} \right.$
    $\texttt{isValidId}(formerNotary, \texttt{notary}) == \texttt{true} \wedge$
    $\texttt{isValidId}(notary, \texttt{notary}) == \texttt{true}\big]$:
      return invalid
  if $\neg\big[\big(|\Sigma| \leq m + 2 \wedge \exists i \in [m] : signee_i == \mathsf{pid}_{\mathsf{cur}} \vee$ $\qquad \left\{ \begin{array}{l} \textit{Checks whether for all singees there exists a sig-} \\ \textit{nature. If it is notarized it has to be the correct} \\ \textit{notary. If less signatures are given it is expected} \\ \textit{that this client participates and signing is not} \\ \textit{done.} \end{array} \right.$
    $initiator == \mathsf{pid}_{\mathsf{cur}}\big) \vee \big(\forall\, i \in [m]\exists_1(\sigma', signee_i) \in \Sigma \wedge$
    $(|\Sigma| == m + 1 \vee (|\Sigma| == m + 2 \wedge \exists(\sigma', formerNotary) \in \Sigma))\big)\big]$:
      return invalid
  for all $(\sigma, pid_\sigma) \in \Sigma$ **do:**
    send (tx, σ) **to** $(pid_\sigma, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathsf{cert}} : \texttt{verifier})$
    wait for $(\texttt{VerResult}, res)$ $\qquad \{ \textit{Checks all provided signatures}$
    if $res == \texttt{false}$:
      return invalid
  if isResolvable$(tx) == \texttt{false} \wedge \mathsf{buffer}_{\mathsf{TxSig}}[tx][4]! = \texttt{reqDeps}$: $\qquad \left\{ \begin{array}{l} \textit{Checks whether all dependencies are available in} \\ \mathsf{internalState} \textit{ of the entity} \end{array} \right.$
    $msg \leftarrow \texttt{resolveTx}(tx, pid)$ $\qquad \left\{ \begin{array}{l} \textit{If transactions or attachments are needed, request is sent to the initiator} \\ \textit{of the transaction.} \end{array} \right.$

    send $(\texttt{Message}, msg)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathsf{unicast}} : \texttt{unicast})$
  else if isResolvable(tx): $\qquad \{ \textit{Party has all necessary information for processing the tx in } \mathsf{internalState}$
    $txDependencies \leftarrow \epsilon$ $\qquad \{ \textit{Collect dependencies for upcoming validation}$
    for all $(txId', outputIndex')$ **do:**
      if $\mathsf{verifiedTx}.\mathsf{contains}((\_, tx, \_, \_, \_))$:
        $txDependencies.\mathsf{add}(\mathsf{verifiedTx}[txId'][1])$
      else:
        $txDependencies.\mathsf{add}(\mathsf{buffer}_{\mathsf{TxSig}}[txId'][1])$
    for all $attachmentId \in proposal$ **do:**
      $txAttachments.\mathsf{add}(attachments[attachmentId])$
    $res \leftarrow \texttt{executeValidation}(tx, txDependencies, txAttachments)$ $\qquad \left\{ \begin{array}{l} \textit{Validate the transaction regarding the} \\ \textit{given context} \end{array} \right.$
    return res

function resolveTx(tx, pids) (msg):
  $\mathsf{buffer}_{\mathsf{TxSig}}[tx][4] \leftarrow \texttt{reqDeps}$
  $msg \leftarrow \epsilon$
  for all $(txId, outputIndex) \in tx_{inputStates}$ **do:** $\qquad \{ \textit{For definition of } tx_{inputStates} \textit{ see above}$
    if $\nexists(\_, tx', \_) \in \mathsf{verifiedTx} \wedge (txId, outputIndex) \in tx'_{outputStates}$: $\qquad \left\{ \begin{array}{l} \textit{Accumulates all transaction necessary for verification but not} \\ \textit{currently in } \mathsf{verifiedTx} \end{array} \right.$
      $\mathsf{buffer}_{\mathsf{TxSig}}.\mathsf{add}((txId, \bot, \bot, \mathsf{pids}, \texttt{requested}))$
      $msg.\mathsf{add}((\mathsf{pids}, (\texttt{SendTransactionFlow}, (txId, outputIndex))))$
  for all $attachmentId \in proposal$ **do:**
    if $(attachmentId, \_) \notin attachments$: $\qquad \left\{ \begin{array}{l} \textit{Accumulates all attachments necessary for verifica-} \\ \textit{tion but not currently in } \mathsf{attachments} \end{array} \right.$
      $msg.\mathsf{add}((\mathsf{pids}, (\texttt{GetAttachment}, (attachmentId))))$
  return msg

Fig. 28. The Corda model $\mathcal{P}^{\mathbf{c}}_{\text{ledger}}$, specification of the Corda client $M_{\texttt{client}}$ (Part 4)

Description of M_{client} (cont.):

Procedures and Functions:

function validateAllLocal() : $\qquad$ $\begin{cases}$ *Checks for all stored transaction if the inputs and attachments can* $\\$ *be resolved and the transaction therefore validated.* $\end{cases}$
$\qquad$ $identities \leftarrow \epsilon, msg \leftarrow \epsilon, resolved \leftarrow$ true
$\qquad$ **while** $resolved ==$ true **do**
$\qquad\qquad$ $resolved \leftarrow$ false, $res \leftarrow \epsilon$
$\qquad\qquad$ **for all** $(\_, tx, \Sigma, pid, status) \in \text{buffer}_{\text{TxSig}} \mid status \notin \{\bot, \text{unrequested}, \text{expNotarization}, \text{reqNotarization}\}$ **do:**

$\qquad\qquad\qquad$ **if** isResolvable(tx): $\qquad$ $\begin{cases}$ resolvable *checks whether all dependencies are verified (i. e., in* $\\$ verifiedTx) *and all attachments available.* $\end{cases}$
$\qquad\qquad\qquad\qquad$ $res \leftarrow$ validate(tx, Σ, ϵ)
$\qquad\qquad\qquad\qquad$ **if** $res == $ valid $\wedge$ $(status == $ sign $\vee$ $status == $ expColSigs$)$:
$\qquad\qquad\qquad\qquad\qquad$ **send** (Sign, tx) **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{cert}} : \text{signer})$
$\qquad\qquad\qquad\qquad\qquad$ **wait for** (Signature, σ)
$\qquad\qquad\qquad\qquad$ **else if** $res == $ valid $\wedge$ isNotarised(tx):
$\qquad\qquad\qquad\qquad\qquad$ verifiedTx.add$((txId, tx, \emptyset))$

$\qquad\qquad\qquad\qquad$ **if** $status == $ sign: $\qquad$ $\begin{cases}$ *If transaction is requested for signing and transaction* $\\$ *has been resolved return an answer.* $\end{cases}$
$\qquad\qquad\qquad\qquad\qquad$ msg.add$((\text{pid}_{\text{cur}}, (\text{Signature}, (tx, \sigma))))$

$\qquad\qquad\qquad\qquad$ **else if** $status = $ expColSigs: $\qquad$ $\begin{cases}$ *If* pid_{cur} *is initiator and transaction is resolvable it* $\\$ *should start the* CollectSignaturesFlow. $\end{cases}$

$\qquad\qquad\qquad\qquad\qquad$ msg.add$((\text{pid}_{\text{cur}}, \text{CollectSignaturesFlow}(tx)))$
$\qquad\qquad\qquad\qquad$ $resolved \leftarrow$ true
$\qquad\qquad\qquad\qquad$ $\text{buffer}_{\text{TxSig}}$.remove$(\_, tx, \Sigma, pid, \_)$
$\qquad\qquad$ **send** (Message, msg) **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}})$

Fig. 29. The Corda model $\mathcal{P}^{\mathbf{c}}_{\text{ledger}}$, specification of the Corda client M_{client} (Part 5)

Description of the protocol $M_{\texttt{notary}} = \{\texttt{notary}\}$:

Participating roles: {notary}
Corruption model: *Dynamic corruption with secure erasures*
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 $M_{\texttt{notary}}$:

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

  − $\mathsf{verifiedTx} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*, \mathsf{verifiedTx} = \emptyset$ {*List of verified transactions, either by this notary or others, as* $(txId, tx, \sigma)$

  − $\mathsf{attachments} \subset \{0,1\}^* \times \{0,1\}^*, \mathsf{attachments} = \emptyset$ {*Set of known attachments and identifiers as* $(attachmentId, attachment)$

  − $\mathsf{buffer}_{\mathsf{TxSig}} \subset \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \times status, \mathsf{buffer}_{\mathsf{TxSig}} = \emptyset$ {*Transaction currently in process as* $(txId, tx, \Sigma, pid, status)$. Σ *is a set of tuples of the form* (pid, σ), $status = \{\texttt{requested}, \texttt{sign}\}$

CheckID(pid, sid, $role$)**:**
    Accept all messages with the same sid.
Corruption behavior:
    DetermineCorrStatus(pid, sid, $role$)**:**
        if corr == true: {*Checks whether notary itself is corrupted.*
            return true
        $corrRes \leftarrow \textbf{corr}(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \texttt{signer})$ {*Checks whether* $\mathcal{F}_{\text{cert}}$ *instance is corrupted.*
        if $corrRes$ == true:
            return true

  See Appendix A for notation details. We expect the ITM to enforce the transaction format $tx = (initiator, [signee_1, \ldots, signee_m],$ $notary, formerNotary, proposal)$ where in $proposal$ input states as $tx_{inputStates} = (txId, outputIndex)$ and output states as $tx_{outputStates}$. We denote by m the number of signees in a transaction. For tuples, lists, etc. we start index counting at 0.

Main:

    recv (Notarise, (tx, Σ)) **from** (pids, $\mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{unicast}} : \texttt{unicast}$) **s.t.** $initiator = \mathsf{pids}$:
    Notarises transaction tx if valid and no double spend. tx format specified above
        if $formerNotary \neq \mathsf{pid}_{\mathsf{cur}}$:   {*Requested notary is defined in formerNotary, abort processing if wrongly addressed*
            send (Message, $\{(initiator, (\text{NotariseRes}, (tx, \texttt{misdirected})))\}$) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{unicast}})$
        else if $\mathsf{verifiedTx}.\texttt{contains}(\_, tx, \_)$:   {*If tx already validated and signed, then return stored signature*
            $(txId, tx, \sigma) \leftarrow \mathsf{verifiedTx}[tx]$
            if $\sigma = \bot$:   {*Generate notary signature if tx is already validated but signature is missing*
                send (Sign, tx) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{cert}} : \texttt{signer})$
                wait for (Signature, σ')
                $\mathsf{verifiedTx}[tx][2] \leftarrow \sigma'$
            send (Message, $\{(initiator, (\text{NotariseRes}, (tx, \sigma)))\}$) **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{unicast}} : \texttt{unicast})$
        else: {*Notary only signs if specified in tx.*
            send tx **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\text{ro}} : \texttt{randomOracle})$
            wait for ($txId$)
            $\mathsf{buffer}_{\mathsf{TxSig}}.\texttt{add}((\texttt{TX}.\texttt{add}(txId), tx, \Sigma, \mathsf{pids}, \texttt{sign}))$ {*Record tx, processing status, and additional data in* $\mathsf{buffer}_{\mathsf{TxSig}}$

            $res \leftarrow \texttt{validate}(tx, \Sigma, initiator)$ {*See definition of function* validate *below*
            $msg \leftarrow \texttt{validateAllLocal}()$

Fig. 30. The Corda model $\mathcal{P}^{\mathbf{c}}_{\text{ledger}}$, specification of the Corda notary $M_{\texttt{notary}}$ (Part 1)

Main:

    recv (RecvTransactionFlow, (pids, (tx, Σ))) **from** (pids, $\text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}} : \text{unicast}$):
      Entity processes a requested transaction
        send tx **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ro}} : \text{randomOracle}$) $\{$*Get transaction id*
        wait for $(txId')$
        $txId \leftarrow \text{TX.add}(txId')$
        if $\text{buffer}_{\text{TxSig}}.\text{contains}((idtx, \bot, \bot, \text{pids}, \text{requested}))$:
           $res \leftarrow \text{validate}(tx, \Sigma, \text{pids})$ $\{$*See definition of* validate *below*
           $\text{validateAllLocal}()$

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

Procedures and Functions:

    function $\text{validate}(tx, \Sigma, pid)$: $\{$*Validates a transaction and produces message for requesting transactions*
                                                         and attachments if necessary.
        if $(\_, tx, \sigma) \in \text{verifiedTx}$: $\{$*If tx already finalized, output existing signature*
           return σ
        if $\text{verifiedTx.contains}(\_, tx, \_, \Sigma', \_)$:
           if $|\Sigma'| == m + 2$:
               return alreadyNotarised
        else:
           if $\text{isDoubleSpend}(tx, \text{verifiedTx})$: $\{$*Checks whether one of the input states is already used in another tx*
                                                        in verifiedTx
              return doubleSpend
        if $\neg[\forall\, i \in [m] : \text{isValidId}(signee_i, \text{client}) == \text{true} \wedge$
          $\text{isValidId}(initiator, \text{client}) == \text{true} \wedge$ $\{$isValidId *checks whether the given pid is regis-*
          $\text{isValidId}(formerNotary, \text{notary}) == \text{true} \wedge$ *tered with the particular role in* networkmap.
          $\text{isValidId}(notary, \text{notary}) == \text{true})]$:
           return invalid
        if $\neg(\forall\, i \in [m] \exists_1 (\sigma', signee_i) \in \Sigma)$: $\{$*Checks whether for all singees there exists a signature.*
           return invalid   *If it is notarized it has to be the correct notary.*
        for all $(\sigma, pid_\sigma) \in \Sigma$ **do:**
           send (tx, σ) **to** ($pid_\sigma, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{cert}} : \text{verifier}$)
           wait for $(\text{VerResult}, res)$ $\{$*Checks all provided signatures*
           if $res == \text{false}$:
               return invalid
        if $formerNotary \neq notary$:
           send (tx, σ) **to** ($formerNotary, sid, \mathcal{F}_{\text{cert}} : \text{verifier}$)
           wait for $(\text{VerResult}, res)$ $\{$*Notary change transaction have always to be signed.*
           if $res == \text{false}$:
               return invalid
        if $\text{isResolvable}(tx) == \text{false} \wedge \text{buffer}_{\text{TxSig}}[tx][4]! = \text{reqDeps}$: $\{$*Checks whether all dependencies are available in*
                                                    internalState *of the entity*
           $msg \leftarrow \text{resolveTx}(tx, pid)$ $\{$*If transactions or attachments are needed, request is sent to the initiator*
                                                   of the transaction.
           send (Message, msg) **to** ($\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}} : \text{unicast}$)
        else if $\text{isResolvable}(tx)$: $\{$*Party has all necessary information for processing the tx in* internalState
           $txDependencies \leftarrow \epsilon$ $\{$*Collect dependencies for upcoming validation*
           for all $(txId', outputIndex')$ **do:**
               if $\text{verifiedTx.contains}((\_, tx, \_, \_, \_))$:
                  $txDependencies.\text{add}(\text{verifiedTx}[txId'][1])$
              else:
                  $txDependencies.\text{add}(\text{buffer}_{\text{TxSig}}[txId'][1])$
           for all $attachmentId \in proposal$ **do:**
               $txAttachments.\text{add}(\text{attachments}[attachmentId])$
           $res \leftarrow \text{executeValidation}(tx, txDependencies, txAttachments)$ $\{$*Validate the transaction regarding the*
           return res *given context*

    function $\text{resolveTx}(tx, pids)$ (msg):
        $\text{buffer}_{\text{TxSig}}[tx][4] \leftarrow \text{reqDeps}$
        $msg \leftarrow \epsilon$
        for all $(txId, outputIndex) \in tx_{inputStates}$ **do:** $\{$*For definition of* $tx_{inputStates}$ *see above*
           if $\nexists(\_, tx', \_) \in \text{verifiedTx} \wedge (txId, outputIndex) \in tx'_{outputStates}$:   *Accumulates all transaction necessary for verification but not*
                                                      currently in verifiedTx
              $\text{buffer}_{\text{TxSig}}.\text{add}((txId, \bot, \bot, \text{pids}, \text{requested}))$
              $msg.\text{add}((\text{pids}, (\text{SendTransactionFlow}, (txId, outputIndex))))$
         for all $attachmentId \in proposal$ **do:**
           if $(attachmentId, \_) \notin \text{attachments}$: $\{$*Accumulates all attachments necessary for verifica-*
               $msg.\text{add}((\text{pids}, (\text{GetAttachment}, (attachmentId))))$ *tion but not currently in* attachments
         return msg

Fig. 31. The Corda model $\mathcal{P}_{\text{ledger}}^{\mathbf{c}}$, specification of the Corda notary M_{notary} (Part 2)

Procedures and Functions (cont.):

 function validateAllLocal() : $\begin{cases}\textit{Checks for all stored transaction if the inputs and attachments can}\\\textit{be resolved and the transaction therefore validated.}\end{cases}$

  $identities \leftarrow \epsilon, msg \leftarrow \epsilon, resolved \leftarrow \texttt{true}$

  while $resolved == \texttt{true}$ **do**

   $resolved \leftarrow \texttt{false}, res \leftarrow \epsilon$

   for all $(\_, tx, \Sigma, pid, status) \in \text{buffer}_{\text{TxSig}} \mid status \notin \{\bot, \texttt{unrequested}\}$ **do:**

    if isResolvable(tx): $\begin{cases}\texttt{resolvable}\ \textit{checks whether all dependencies are verified (i.e., in}\\\texttt{verifiedTx})\ \textit{and all attachments available.}\end{cases}$

     $res \leftarrow \text{validate}(tx, \Sigma, \epsilon)$

     if $res == \texttt{valid} \wedge status == \texttt{sign}$:

      send $(\texttt{Sign}, tx)$ **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{cert}} : \texttt{signer})$

      wait for $(\texttt{Signature}, \sigma)$

      verifiedTx.add$(txId, tx, \sigma)$

     else if $res == \texttt{valid}$:

      verifiedTx.add$((txId, tx, \bot))$

     $\sigma \leftarrow res$

     if $status == \texttt{sign}$: $\begin{cases}\textit{If transaction is requested for notarisation and trans-}\\\textit{action has been resolved return an answer.}\end{cases}$

      msg.add$((\text{pid}_{\text{cur}}, (\texttt{NotariseRes}, (tx, \sigma))))$

     $resolved \leftarrow \texttt{true}$

     $\text{buffer}_{\text{TxSig}}$.remove$(\_, tx, \Sigma, pid, \_)$

  send $(\texttt{Message}, msg)$ **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{unicast}})$

Fig. 32. The Corda model $\mathcal{P}_{\text{ledger}}^{\mathbf{c}}$, specification of the Corda notary M_{notary} (Part 3)

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

Participating roles: $\{\texttt{unicast}\}$
Corruption model: *incorruptible*
Protocol parameters:
 – $\delta \in \mathbb{N}$ $\begin{cases}\textit{A general delay parameter for the time the adversary is allowed to prevent messages}\\\textit{from being delivered.}\end{cases}$

Fig. 33. The unicast functionality $\mathcal{F}_{\text{unicast}}$ that models an ideal network for the Corda model $\mathcal{P}_{\text{ledger}}^{\mathbf{c}}$

Description of M_{unicast}:

Implemented role(s): $\{\texttt{unicast}\}$
Subroutines: $\mathcal{P}_{\text{ledger}}^{\mathbf{c}} : \texttt{client}, \mathcal{P}_{\text{ledger}}^{\mathbf{c}} : \texttt{notary}$ $\{\mathcal{F}_{\text{unicast}}\ \textit{can access the corruption status in}\ \mathcal{P}_{\text{ledger}}^{\mathbf{c}}$
Internal state:

 – $\text{buffer}_{msg} \subset \mathbb{N} \times \mathbb{N} \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^*, \text{buffer}_{msg} = \emptyset$ $\begin{cases}\textit{Buffer for messages consisting of tuples}\\(label, round, \texttt{sender}, receiver, content)\end{cases}$

 – $\text{labels} \in \mathbb{N}, \text{labels} = \emptyset$ $\{\textit{Already used identifiers}$
 – $\text{round} \in \mathbb{N}, \text{round} = 0$ $\{\textit{Current round/time unit}$

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

Main:

 recv $(\texttt{Message}, msg)$ **from** $\texttt{I/O}$ **s.t.** $msg \subset \{(receiver, content)\mid receiver, content \in \{0,1\}^*\}$: $\begin{cases}\textit{Request to deliver a several different mes-}\\\textit{sages to different receivers}\end{cases}$

  $leakage \leftarrow \epsilon$

  for all $(r, m) \in msg$ **do:**

   $label \xleftarrow{\$} \mathbb{N}$ **s.t.** $label \notin \text{labels}$ $\{\textit{Generate an unique lable per message that needs to be delivered}$

   labels.add$(label)$

   $leakage$.add$(label, \text{pid}_{\text{cur}}, r, |m|)$ $\{\textit{Record receiver, round, and content length as leakage}$

   buffer_{msg}.add$(label, round, \text{pid}_{\text{cur}}, r, m)$ $\{\textit{Queue message for delivery}$

  send $(\texttt{MultiMessage}, leakage)$ **to** NET $\{\textit{Forward leakage to}\ \mathcal{A}$

 recv $(\texttt{Deliver}, label)$ **from** NET: $\{\mathcal{A}\ \textit{triggers delivery of messages}$

  parse $(label, s, r, m)$ **from** $\text{buffer}_{msg}[label]$ $\{\textit{Fetch data from message queue}$

  parse $(\texttt{Command}, content)$ **from** m

  buffer_{msg}.remove$(label, s, r, m)$ $\{\textit{Remove message from delivery queue}$

  send $(\texttt{Command}, (s, content))$ **to** $(r, sid, \texttt{I/O})$ $\{\textit{Deliver message to receiver}$

 recv $(\texttt{UpdateRound})$ **from** NET: $\{\mathcal{A}\ \textit{triggers round update if all messages older than}\ \delta\ \textit{rounds are delivered}$

  if $\exists (\_, r', \_, \_, \_) \in \text{buffer}_{msg} : r' < \text{round} - \delta$: $\{\textit{Ensure all messages are delivered within}\ \delta\ \textit{rounds}$

   reply $(\texttt{UpdateRound}, \texttt{false}, \epsilon)$

  else: $\{\textit{Round update accepted}$

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

   reply $(\texttt{UpdateRound}, \texttt{true}, \epsilon)$

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

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

Fig. 33. The unicast functionality $\mathcal{F}_{\text{unicast}}$ that models an ideal network for the Corda model $\mathcal{P}_{\text{ledger}}^{\mathbf{c}}$

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

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

Implemented role(s): {signer, verifier}
Internal state:
  – $(\mathsf{pk}, \mathsf{sk}) \in (\{0,1\}^* \cup \{\bot\})^2 = (\bot, \bot)$.   *{Key pair.*
  – pidowner $\in \{0,1\}^* \cup \{\bot\} = \bot$.   *{Party ID of the key owner.*
  – msglist $\subset \{0,1\}^* = \emptyset$.   *{Set of recorded messages.*
  – corr $\in \{\texttt{true}, \texttt{false}\} = \texttt{false}$.   *{Is signature key corrupted?*
CheckID($pid, sid, role$):
  Check that $sid = (pid', sid')$:
  If this check fails, output reject.   *{A single instance manages all parties and roles in a single session. A session models one signature key pair belonging to party pid'.*
  Otherwise, accept all entities with the same SID.
Corruption behavior:
  – **DetermineCorrStatus**($pid, sid, role$): Return corr.
Initialization:
  $(\mathsf{pk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{Gen}(1^\eta)$ *{Generate public/secret key pair*
  Parse $\mathsf{sid}_{\mathrm{cur}}$ as (pid, sid).
  pidowner $\leftarrow pid$.
Main:

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

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

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

Fig. 34. The ideal signature functionality $\mathcal{F}_{\mathrm{cert}}$.

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

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

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

Implemented role(s): {randomOracle}
Internal state:
  – hashHistory $\subseteq \{0,1\}^* \times \{0,1\}^\eta$, initially hashHistory $= \emptyset$ *{The set of recorded value/hash pairs*
CheckID($pid, sid, role$):
  Accept all messages with the same sid.
Main:

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

Fig. 35. The random oracle $\mathcal{F}_{\mathrm{ro}}$ (cf. [11])

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

Participating roles: {submit}
Corruption model: *incorruptible*
Protocol parameters:
  − executeValidation   {*Validation algorithm that verfies whether it is ok to add a transaction to the current corda state*

  − validateAttachment   {*Algorithm that validates attachments*

Description of $M^{\mathbf{c}}_{\mathrm{submit}}$:

Implemented role(s): {submit}
Subroutines: $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$: storage, $\mathcal{F}^{\mathbf{c}}_{\mathrm{read}}$: read
CheckID(pid, sid, $role$):
   Accept all messages with the same sid.
  In the following, we will use the abberiation (1) for the expression $msg = [\mathtt{tx}, (initiator, [signee_1, \ldots, signee_m], notary, formerNotary, proposal)]$, (2) for $msg = (\mathtt{attachment}, attachment)$, and (3) for $msg = (\mathtt{tx}, txId, pid_{recv})$.

Main:

   recv (Submit, msg, $internalState$) **from** I/O: {*See Figure 5 for definition of internalState and the local variables it includes*

      if $\neg[(1) \lor (2) \lor (3)]$:
         reply ($\mathtt{validationProcessed}, \mathtt{false}, \epsilon$)   {*Submitted transactions/attachments need to have the expected data format*
      if $(1) \land \mathsf{pid}_{\mathsf{cur}} \notin \{initiator, signee_1, \ldots, signee_m\}$:

         reply ($\mathtt{validationProcessed}, \mathtt{false}, \epsilon$)   {*Clients can only submit transactions they are involved in*

      if $(1) \land \neg[initiator, signee_1, \ldots, signee_m$ are clients ($role = \mathtt{client}$, pid prefixed by $\mathtt{client}$)$\land notary, formerNotary$ are notaries: ($role = \mathtt{notary}$, pid prefixed by $\mathtt{notary}$) $\land$
      $\forall pid \in \{initiator, signee_1, \ldots, signee_m, notary, formerNotary\} : pid \in identities]$

                                                      {*Check correct identities and roles*
         reply ($\mathtt{validationProcessed}, \mathtt{false}, \epsilon$)   {*Clients can submit transactions they are involved in transaction, all parties need to be registered*

Fig. 36. The write/submit functionality $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ of $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ (Part 1)

Main:

    if (1):
        send $(\mathsf{GetID}, \mathsf{tx}, (initiator, [signee_1, \ldots, signee_m], notary, formerNotary, proposal))$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\mathbf{c}}_{\mathrm{storage}})$ $\{$*In case that a transaction was submitted* $\{$*Request txID*
        wait for $(\mathsf{GetID}, \mathsf{tx}, id)$ $\{\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$ *returns an id or* false
        $attachments \leftarrow \epsilon$
        parse $(id^a_1, \ldots, id^a_l)$ **from** $msg.\mathsf{remove}(\mathsf{tx})$ $\{$*We leak the ids of the attachment ids involved in a transaction*
        for all $id^a \in \{id^a_1, \ldots, id^a_l\}$ **do:** $\{$*Connect attachments to transactions*

            send $(\mathsf{GetContent}, \mathsf{attachment}, id^a)$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\mathbf{c}}_{\mathrm{storage}} : \mathsf{storage})$ $\begin{cases}Get\ attachment\ from\\ \mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}\end{cases}$
            wait for $(\mathsf{GetContent}, \mathsf{attachment}, attachment)$
            $attachments.\mathsf{add}(id^a, attachment)$
        parse $[(inTxId_1, idx_1), \ldots, (inTxId_h, idx_h)](out_1, \ldots, out_e)$ **from** $proposal$
            $\begin{cases}Extract\ input\ tx\ ids\ and\ input\ index\ and\ output\ index\ from\ proposal.\ This\ needs\ to\ be\ leaked.\ If\ there\ are\ no\\ inputs,\ this\ returns\ \perp\end{cases}$
        $leakage \leftarrow [(id^a_1, \ldots, id^a_l), ((inTxId_1, idx_1), \ldots, (inTxId_h, idx_h))(out_1, \ldots, out_e))]$
        if $initiator = \mathsf{pid}_{\mathsf{cur}}$: $\{$*Initiators need to know dependend tx and used attachments*
        send $\mathsf{getCurrentKnowledge}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{read}} : \mathsf{read})$ $\{$*Query* $\mathsf{pid}_{\mathsf{cur}}$*'s current knowledge*
        wait for $(\mathsf{getCurrentKnowledge}, transactions^c_{\mathsf{pid}_{\mathsf{cur}}}, attachments^c_{\mathsf{pid}_{\mathsf{cur}}})$

        send $\mathsf{getKnowledge}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}^{\mathbf{c}}_{\mathrm{storage}} : \mathsf{storage})$ $\begin{cases}Requests\quad \mathsf{pid}_{\mathsf{cur}}\text{'s}\quad current\\ (maximal)\quad knowledge\quad at\\ \mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}\end{cases}$
        wait for $(\mathsf{getKnowledge}, txGraph_{\mathsf{pid}_{\mathsf{cur}}}, attachments_{\mathsf{pid}_{\mathsf{cur}}})$
        if $\mathsf{missingDependency}(msg, transactions^c_{\mathsf{pid}_{\mathsf{cur}}}, attachments^c_{\mathsf{pid}_{\mathsf{cur}}})$:

            $\begin{cases}\mathsf{missingDependency}(msg, transactions^c_{\mathsf{pid}_{\mathsf{cur}}}, attachments^c_{\mathsf{pid}_{\mathsf{cur}}})\ returns\ \mathsf{false}\ if\ all\ input\ transactions\ to\ msg\ and\\ their\ dependencies\ down\ to\ the\ issuance\ transactions\ are\ available\ in\ transactions^c_{\mathsf{pid}_{\mathsf{cur}}}\ as\ well\ as\ all\ attach-\\ ments\ used\ in\ the\ subgraph(s)\ below\ msg\ are\ available\ in\ attachments^c_{\mathsf{pid}_{\mathsf{cur}}},\ otherwise\ \mathsf{true}\end{cases}$
            reply $(\mathsf{validationProcessed}, \mathsf{false}, \epsilon)$ $\begin{cases}Decline\ submission\ if\ \mathsf{pid}_{\mathsf{cur}}\ has\ no\ access\\ to\ data\ to\ validate\ msg\end{cases}$

        if $\neg\mathsf{executeValidation}(msg, transactions^c_{\mathsf{pid}_{\mathsf{cur}}}[inTxId_1, \ldots, inTxId_h], attachments^c_{\mathsf{pid}_{\mathsf{cur}}}[id^a_1, \ldots, id^a_l])$: $\begin{cases}Check\quad validity\quad of\\ transaction\quad regard-\\ ing\ its\ direct\ inputs\end{cases}$
            reply $(\mathsf{validationProcessed}, \mathsf{false}, \epsilon)$
        if $\mathsf{isDoubleSpend}msg$: $\{$*Check double spending*
            reply $(\mathsf{validationProcessed}, \mathsf{false}, \epsilon)$ $\{$*Decline submission*

      $leakage.\mathsf{add}[id, initiator, signee_1, \ldots, signee_m, formerNotary])$ $\begin{cases}\mathcal{A}\ receives\ all\ party\ expected\ notary\ that\\ are\ involved\ in\ transaction\ id\end{cases}$

      if one of the parties $initiator, signee_1, \ldots, signee_m, formerNotary$ is in $CorruptionSet$:
          $leakage.\mathsf{add}(proposal)$ $\begin{cases}If\ an\ involved\ party\ is\ corrupted,\ \mathcal{A}\ gets\ all\ details\ of\ the\ tx\ and\ the\ connected\ subgraph\ as\ he\\ is\ allowed\ to\ query\ the\ subgraph\ for\ verification\end{cases}$

          if $\mathsf{pid}_{\mathsf{cur}} = initiator$: $\begin{cases}If\ the\ initiator\ has\ access\ to\ all\ necessary\ transactions,\ the\ full\ sub-\\ graph\ below\ the\ tx\ leaks\end{cases}$

            $txGraph_{\mathsf{pid}_{\mathsf{cur}}}.\mathsf{addToTxGraph}(id, msg.\mathsf{remove}(\mathsf{tx}), \emptyset)$ $\begin{cases}\mathsf{addToTxGraph}\ adds\ the\ tx\ extracted\ from\ \mathsf{tx}\ to\\ txGraph - \mathsf{pid}_{\mathsf{cur}}\end{cases}$

            $txSubGraph \leftarrow \mathsf{getConnectedSubGraph}(\mathsf{tx}, txGraph_{\mathsf{pid}_{\mathsf{cur}}})$

            $\begin{cases}\mathsf{getConnectedSubGraph}(msg, txGraph_{\mathsf{pid}_{\mathsf{cur}}})\ outputs\ the\ (maximal)\ connected\ subgraph\ of\ txGraph_{\mathsf{pid}_{\mathsf{cur}}}\ such\ that\\ msg\text{'s}\ outputs\ are\ all\ edges\ of\ this\ subgraph\ (\text{``maximal''}\ means\ \nexists\ a\ connected\ subgraph\ txGraph\ containing\\ msg\text{'s}\ output\ and\ that\ is\ a\ superset\ of\ \mathsf{getConnectedSubGraph}(msg, txGraphs))\end{cases}$

            for all clients $(txID, tx, attachments_{tx})$ in $txSubGraph$ **s.t.** $txId$ is an input to msg **do:**
                $leakage.\mathsf{add}(txID, tx, attachments_{tx})$ $\{$*Leak tx id, content, and connected attachments*
      else:
          $leakage.\mathsf{add}(|proposal|)$ $\begin{cases}If\ all\ involved\ party\ are\ honest,\ \mathcal{A}\ gets\ only\ length\ of\ the\ proposal\\ data\end{cases}$
      reply $(\mathsf{validationProcessed}, \mathsf{true}, leakage)$

Fig. 37. The write/submit functionality $\mathcal{F}^{\mathbf{c}}_{\mathrm{submit}}$ of $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$ (Part 2)

Main:

 if (2):
  send (GetID, attachment, $attachment$) **to** (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$) $\{$ *In case that an attachment was submitted*
                             $\{$ *Request id of the attachment*

  wait for (GetID, attachmentid) $\{\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ *returns an id or* false

  $leakage.\text{add}(id, |attachment|)$ $\begin{cases} \mathcal{A} \text{ receives the length of the attach-} \\ \text{ment as leakage} \end{cases}$
  reply (validationProcessed, validateAttachment($attachment$), $leakage$)

 if (3) $\land$ pid$_{\text{cur}}$ is a client ($role = $ client and pid$_{\text{cur}}$ prefixed by client): $\begin{cases} A \text{ client pushes a transac-} \\ \text{tion to another entity} \end{cases}$

  send getCurrentKnowledge **to** (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{read}}$: read) $\{$ *Query* pid$_{\text{cur}}$'s *current knowledge*
  wait for (getCurrentKnowledge, $transactions_{\text{pid}_{\text{cur}}}^{C}$, $attachments_{\text{pid}_{\text{cur}}}^{C}$)
  send (GetContent, tx, $txID$) **to** $\{$ *Request tx content*
  wait for (GetContent, tx, tx) $\{\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ *returns an id or* false
  send getKnowledge **to** (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage)
  wait for (getKnowledge, $txGraph_{\text{pid}_{\text{cur}}}$, $attachments_{\text{pid}_{\text{cur}}}$)
  if $tx \in transactions_{\text{pid}_{\text{cur}}}^{C} \land \neg\text{missingDependency}(tx, transactions_{\text{pid}_{\text{cur}}}^{C}, attachments_{\text{pid}_{\text{cur}}}^{C})$:
                    $\{$pid$_{\text{cur}}$ *has access to tx and can forward it*

   if pid_{recv} in *CorruptionSet*: $\{$ *tx is pushed to a corrupted party*

    send (getTxGraph, $internalState$, incBuffer) **to** (pid$_{\text{cur}}$, sid$_{\text{cur}}$, $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$: storage)
          $\{$ *Generate transaction graph including tx from requestQueue, located in* $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$
   wait for (getTxGraph, $txGraph$)

   $txSubGraph \leftarrow \text{getConnectedSubGraph}(tx, txGraph)$

       $\begin{cases} \text{getConnectedSubGraph}(msg, txGraph) \text{ outputs the (maximal) connected subgraph of } txGraph \text{ such that } msg\text{'s} \\ \text{outputs are all edges of this subgraph ("maximal" means } \nexists \text{ a connected subgraph } txGraph \text{ containing } msg\text{'s} \\ \text{output and that is a superset of } \text{getConnectedSubGraph}(msg, txGraphs)) \end{cases}$

   for all clients $(txID, tx, attachments_{tx})$ in $txSubGraph$ **s.t.** $txId$ is an input to msg **do:**
    $leakage.\text{add}(txID, tx, attachments_{tx})$ $\{$ *Leak tx id, content, and connected attachments*
   else:
    $leakage \leftarrow (txId, \text{pid}_{\text{cur}}, pid_{recv})$ $\begin{cases} \mathcal{A} \text{ is informed that party pushed tx} \\ \text{to entity} \end{cases}$
    reply (validationProcessed, true, $leakage$)
  else:
   reply (validationProcessed, false, ϵ) $\{$ *Input rejected*
 reply (validationProcessed, false, ϵ) $\{$ *Input rejected*

Fig. 38. The write/submit functionality $\mathcal{F}_{\text{submit}}^{\mathbf{c}}$ of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ (Part 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 : $\{0,1\}^* \times \{0,1\}^*$ $\{$ *The set of known transactions for a party. Entries of form* $(pid, txID)$

 – knownAttachments : $\{0,1\}^* \times \{0,1\}^*$ $\{$ *The set of known attachments for a party. Entries of form* (pid, id^a)

 – knownTransactions$^{\mathsf{r}}$: $\{0,1\}^* \times \{0,1\}^*$ $\begin{cases} \text{The set of known transactions for a party outputted during reading.} \\ \text{Entries of form } (pid, txID) \end{cases}$

 – knownAttachments$^{\mathsf{r}}$: $\{0,1\}^* \times \{0,1\}^*$ $\begin{cases} \text{The set of known attachments for a party outputted during reading.} \\ \text{Entries of form } (pid, id^a) \end{cases}$

CheckID(pid, sid, $role$)**:**
 Accept all messages with same sid.
 In the following, we will use the abberivation (1) for the expression $tx = [\text{tx}, (initiator, [signee_1, \ldots, signee_m], notary, formerNotary, proposal)]$.
Main:
 recv (InitRead, msg, $internalState$) **from** I/O: $\begin{cases} \text{See Figure 5 for definition of } internalState \text{ and the local variables it} \\ \text{includes} \end{cases}$

  reply (InitRead, true, ϵ) $\{$ *Reads in Corda are always local, we do not leak anything to* $\mathcal{A}$.

Fig. 39. The read functionality $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ (Part 1)

Main:

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

      s.t. $input = ([txID_1, \ldots, txID_h], [id_1^a, \ldots, id_o^a])$ *{Read outputs the tx subgraph $\mathsf{pid}_{\text{cur}}$ is aware of*

      Execute code from `getCurrentKnowledge` here. Write the output into $txGraph_{\mathsf{pid}_{\text{cur}}}$ and $attachments_{\mathsf{pid}_{\text{cur}}}$.

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

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

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

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

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

         send getKnowledge **to** $(\mathsf{pid}_{\text{cur}}, \mathsf{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}} : \mathsf{storage})$

         wait for (getKnowledge, $transactions_{\mathsf{pid}_{\text{cur}}}^P$, $attachments_{\mathsf{pid}_{\text{cur}}}^P$)

         $hasAccess \leftarrow \texttt{true}$

         for all $txID' \in knownTX^r$ **do:**

            if $txID' \notin transactions_{\mathsf{pid}_{\text{cur}}}^P$:   *{Check that the adversary only distributes knowledge she has access to*

               send (GetContent, tx, $txID'$) **to** $(\mathsf{pid}_{\text{cur}}, \mathsf{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}} : \mathsf{storage})$

               wait for (GetContent, tx, tx')

               if $\nexists$ corrupted party in tx:

                  $hasAccess \leftarrow \texttt{false}$

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

            if $id' \notin attachments_{\mathsf{pid}_{\text{cur}}}^P$:   *{Check that the adversary only distributes knowledge she has access to*

               send (GetContent, attachment, id') **to** $(\mathsf{pid}_{\text{cur}}, \mathsf{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}} : \mathsf{storage})$

               wait for (GetContent, attachment, $attachment'$)

               if $attachment'$ in $requestQueue$ or $msglist \wedge \nexists$ a leakage of $attachment$ in transcript:

                  $hasAccess \leftarrow \texttt{false}$

         if $hasAccess$:

            for all $txID' \in \{txID_1, \ldots, txID_h\} \setminus knownTX^r$ **do:**

               knownTransactions.add$(\mathsf{pid}_{\text{cur}}, txID')$

            for all $id' \in \{id_1^a, \ldots, id_o^a\} \setminus knownAtt^r$ **do:**

               knownAttachments.add$(\mathsf{pid}_{\text{cur}}, id')$

         else:

            reply (FinishRead, $\perp, \epsilon$)   *{$\mathsf{pid}_{\text{cur}}$ does not have access to suggested state*

      else:

         reply (FinishRead, $\perp, \epsilon$)   *{$\mathsf{pid}_{\text{cur}}$ does not have access to suggested state*

      $verifiedTx \leftarrow msglist$ *{Remove entries that are not*

      for all $(txID', tx', \_) \in verifiedTx$ **s.t.** $txID' \notin \{txID_1, \ldots, txID_h\}$ **do:**   *in the suggested state from*

         $verfiedTx.\text{remove}(txID', tx', \_)$ *$\mathcal{A}$*

      if $\exists tx'$ in $msglist$, **s.t.** tx' is not in $verifiedTx \wedge \mathsf{pid}_{\text{cur}}$ 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 have it after at most δ rounds after notarization in their*

                                                  states

      if $\exists tx'$ in $msglist$ **s.t.** tx' not in $verifiedTx \wedge \mathsf{pid}_{\text{cur}}$ 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 have it after at most 2δ rounds after notarization in their*

                                                  states

      if $\exists tx'$ as push operation to $\mathsf{pid}_{\text{cur}}$ in $msglist$ **s.t.** tx' not in $verifiedTx \wedge$ all parties involved in tx' are not corrupted $\wedge round$ is

        greater than committing round of $tx' + |subGraph|(tx')\delta + 1$ **s.t.** $subGraph(tx')$ is the subgraph below tx':

         reply (FinishRead, $\perp$)   *{Push of transaction should be finished after δ rounds*

      $cleanedVerifiedTx \leftarrow \emptyset, attachments \leftarrow \emptyset$ *{Separate attachment data from tx data*

      for all $(txID, tx, \_) \in verifiedTx$ **do:**

         $cleanedVerifiedTx.\text{add}(txID, tx)$

      for all $id^a \in \{id_1^a, \ldots, id_o^a\}$ **do:**   *{Output suggested attachments*

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

         $attachments.\text{add}(id^a, attachment)$

      if there is a transaction in $cleanedVerifiedTx$ such 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 $\mathsf{knownTransactions}^r$, resp. $\mathsf{knownAttachments}^r$

      reply (FinishRead, $(attachments, cleanedVerifiedTx), \epsilon$)

Fig. 40. The read functionality $\mathcal{F}_{\text{read}}^{\mathbf{c}}$ of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ (Part 2)

Main:

    recv (CorruptedRead, pid, msg, $internalState$) **from** I/O: $\{\mathcal{A}$ *receives all information via leakage*

        reply (FinishRead, ϵ)

    recv getCurrentKnowledge **from** I/O: $\{$*Other subroutines may ask* $\mathcal{F}_{\text{read}}^{\textbf{c}}$ *for an entities current state*

        send getKnowledge **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\textbf{c}} : \text{storage})$ $\{$*Request* pid_{cur} *'s knowledge at* $\mathcal{F}_{\text{storage}}^{\textbf{c}}$

        wait for (getKnowledge, $txGraph_{\text{pid}_{\text{cur}}}$, $attachments_{\text{pid}_{\text{cur}}}$)

        send responsively (getKnowledge, pid_{cur}) **to** NET $\{$*Query current state of* pid_{cur} *at* $\mathcal{A}$

        wait for (getKnowledge, pid_{cur}, $(txID_1, \ldots, txID_h)$, $(id_1^a, \ldots, id_o^a)$)

        $knownTX^r \leftarrow \{txID' \,|(\text{pid}_{\text{cur}}, txID') \in \text{knownTransactions}\}$ $\{$*Extract previous state of* pid_{cur}

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

                                      $\left\{\begin{array}{l}\text{\textit{Update} } \text{pid}_{\text{cur}} \text{ \textit{'s knowledge if this is compliant with the view of} } \mathcal{F}_{\text{ledger}}^{\textbf{c}}\end{array}\right.$

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

        if $\nexists txID' \in \{txID_1, \ldots, txID_h\}$ **s.t.** $txID' \notin txGraph_{\text{pid}_{\text{cur}}} \wedge \nexists id' \in \{id_1^a, \ldots, id_o^a\}$ **s.t.** $id' \notin attachments_{\text{pid}_{\text{cur}}}$:

            send getKnowledge **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\textbf{c}} : \text{storage})$

            wait for (getKnowledge, $transactions_{\text{pid}_{\text{cur}}}^p$, $attachments_{\text{pid}_{\text{cur}}}^p$)

            $hasAccess \leftarrow \text{true}$

            for all $txID' \in knownTX^r$ **do:**

                if $txID' \notin transactions_{\text{pid}_{\text{cur}}}^p$: $\{$*Check that the adversary only distributes knowledge she has access to*

                    send (GetContent, tx, $txID'$) **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}} : \text{storage})$

                    wait for (GetContent, tx, tx')

                    if $\nexists$ corrupted party in tx:

                        $hasAccess \leftarrow \text{false}$

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

                if $id' \notin attachments_{\text{pid}_{\text{cur}}}^p$: $\{$*Check that the adversary only distributes knowledge she has access to*

                    send (GetContent, attachment, id') **to** $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\textbf{c}} : \text{storage})$

                    wait for (GetContent, attachment, $attachment'$)

                    if $attachment'$ in $requestQueue$ or $msglist \wedge \nexists$ a leakage of $attachment$ in transcript:

                        $hasAccess \leftarrow \text{false}$

        if $hasAccess$:

            for all $txID' \in \{txID_1, \ldots, txID_h\} \setminus knownTX^r$ **do:**

                knownTransactions.add($\text{pid}_{\text{cur}}, txID'$)

            for all $id' \in \{id_1^a, \ldots, id_o^a\} \setminus knownAtt^r$ **do:**

                knownAttachments.add($\text{pid}_{\text{cur}}, id'$)

    $knownTransactions_{\text{pid}_{\text{cur}}} \leftarrow \{txID \,|(\text{pid}_{\text{cur}}, txID) \in \text{knownTransactions}\}$ $\left\{\begin{array}{l}\textit{Extract previous state of}\\ \text{pid}_{\text{cur}}\end{array}\right.$

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

    reply (getCurrentKnowledge, $transactions_{\text{pid}_{\text{cur}}}$, $attachments_{\text{pid}_{\text{cur}}}$)

Fig. 41. The read functionality $\mathcal{F}_{\text{read}}^{\textbf{c}}$ for $\mathcal{F}_{\text{ledger}}$ to represent Corda (Part 3).

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

Participating roles: {update}
Corruption model: *incorruptible*
Protocol parameters:
   – executeValidation   *{ Validation algorithm that verfies whether it is ok to add a transaction to the current corda state*
   – validateAttachment   *{ Algorithm that validates attachments*

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

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

       recv (Update, [normal, msg], $internalState$) **from** I/O: *{ See Figure 5 for definition of internalState and the local variables it includes*

           executeBasicChecks()   *{ Definition of executeBasicChecks() below. The procedure gets all local and global variables of $M_{\text{update}}^{\mathbf{c}}$ as input and may update them and create new variables*

          $msgListAppend \leftarrow \emptyset$, $updRequestQueue \leftarrow \emptyset$, $ctr \leftarrow \max\{i | (i, \_, \_, \_, \_) \in msglist\}$ *{ Build extension of msglist. If msglist $= \emptyset$ then max defaults to -1*
        for all $(id^a, attachment, pid') \in \{(id_1^a, attachment_1, pid_1), \ldots, (id_m^a, attachment_m, pid_m)\}$ **do:**
           if $\neg$validateAttachment($attachment$):   *{ Reject invalid attachments from $\mathcal{A}$*
               reply (Update, $\emptyset, \emptyset, \epsilon$)   *{ Processing aborted*
           $ctr \leftarrow ctr + 1$
           $msgListAppend$.add(ctr, $round$, tx, $attachment$, $round$, pid')   *{ Add attachments from Adv*
        for all $(id', pid') \in \{(id_1^a, pid_i^a), \ldots, (id_1^a, pid_i^a)\}$ **do:**   *{ Add attachments from honest parties*
           send (GetContent, attachment, id') **to** $(\_, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}} : \text{storage})$ *{ Get attachment content from $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$*
           wait for (GetContent, attachment, $attachment'$)
           $ctr \leftarrow ctr + 1$
           $msgListAppend$.add(ctr, $round$, tx, $attachment'$, $round$, pid')   *{ Add attachments from Adv*
           Let $(ctr_{tmp}, (\text{attachment}, attachment'), r, pid')$ the matching entry in $requestQueue$
           $updRequestQueue$.add(ctr_{tmp}, (attachment, $attachment'$), r, pid')
        if $\exists txID \in \{txID_1, \ldots, txID_l\}$ **s.t.** $txID$ is not in $reqQueue$ or $\{txID_{i_1}, \ldots, txID_{i_h}\}$:   *{ Check that input data exists*
           reply (Update, $\emptyset, \emptyset, \epsilon$)   *{ Processing aborted*
        $msglist' \leftarrow msglist$ *{ Used for processing update*
        for all $txID \in \{txID_1, \ldots, txID_l\}$ **do:**   *{ Check that all involved honest parties agreed on the tx*
           send (GetContent, tx, $txID$) **to** $(\_, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{storage}}^{\mathbf{c}} : \text{storage})$ *{ Get tx content from $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$*
           wait for (GetContent, tx, tx)
           parse $(initiator, signee_1, \ldots, signee_h)$ **from** tx *{ checkAgreements outputs true if for all honest signees, resp. honest initiator there exists a submit request of tx in requestQueue*
           $txAgreed \leftarrow$ checkAgreements(tx, $requestQueue$, $CorruptionSet$)

           if $\neg txAgreed$:
                reply (Update, $\emptyset, \emptyset, \epsilon$)   *{ Processing aborted*
           $id = |msglist'| + 1$ *{ Start preparing the update*
           Let r be the highest round in $reqQueue$ and $\{(txID_{i_1}, tx_{i_1}, pid_{i_1}, r_{i_1}), \ldots, (txID_{i_h}, tx_{i_h}, pid_{i_h}, r_{i_h})\}$ for $txID$
           $msglist'$.add(id, $round$, tx, [tx, tx, true], r, $initiator$)
           if isDoubleSpend(tx):   *{ Prevention of double-spending*
                reply (Update, $\emptyset, \emptyset, \epsilon$)   *{ Processing aborted*
        for all $txID$ in msg **do:**   *{ Remove "processed" tx from requestQueue*
           for all $(ctr_{temp}, msg', round', pid')$ identitied by $txID$ is in $requestQueue$ **do:**
               $updRequestQueue$.add(ctr_{temp}, msg', $round'$, pid')
        for $txID \in \{txID_1, \ldots, txID_l\}$ **do:**   *{ In the order of the input msg*
           $ctr \leftarrow ctr + 1$
           Let $(ctr_{temp}, msg', round', initiator')$ be the first message in $requestQueue$ identified by $txID$ such that $initiator'$ is the initiator in msg'
           $msgListAppend$.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 are not in $msglist) \cup msgListAppend$
           reply (Update, $\emptyset, \emptyset, \epsilon$)   *{ Processing aborted*
         if there is a attachment in $requestQueue$ such that it is not part of $updRequestQueue$:
           reply (Update, $\emptyset, \emptyset, \epsilon$)   *{ Processing aborted*
         reply (Update, $msgListAppend$, $updRequestQueue$, ϵ)   *{ Return list extension and updated queue.*

Fig. 42. The update functionality $\mathcal{F}_{\text{update}}^{\mathbf{c}}$ of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ (Part 1)

Main:

> **recv** $(\texttt{Update}, [\texttt{txExchange}], \mathit{internalState})$ **from** $(\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{ledger}} : \texttt{client})$: $\qquad$ $\begin{cases} \textit{Record shared transactions in} \\ \textit{msglist} \end{cases}$
>> $\mathit{updRequestQueue} \leftarrow \emptyset, \mathit{msgListAppend} \leftarrow \emptyset$
>> $\mathit{ctr} \leftarrow \max\{i | (i, \_, \_, \_, \_, \_) \in \mathit{msglist}\}|$ $\qquad$ $\begin{cases} \textit{If tx exchange is done, we add it to the} \\ \textit{message list} \end{cases}$
>> **for all** $(\mathit{ctr}_{temp}, [\texttt{tx}, \mathit{txID}, \mathit{pid}_{recv}], r, \mathit{pid}) \in \mathit{requestQueue}$ **do:**
>>> $\mathit{ctr} \leftarrow \mathit{ctr} + 1$
>>> $\mathit{msgListAppend}.\mathrm{add}(\mathit{ctr}, \mathit{round}, \texttt{tx}, [\texttt{tx}, \mathit{txID}, \mathit{pid}_{recv}], r, \mathit{pid})$
>>> $\mathit{updRequestQueue}.\mathrm{add}(\mathit{ctr}_{temp}, [\texttt{tx}, \mathit{txID}, \mathit{pid}_{recv}], r, \mathit{pid})\}$
>> **reply** $(\texttt{Update}, \mathit{msgListAppend}, \mathit{updRequestQueue}, \epsilon)$ $\qquad$ $\{\textit{Send update to } \mathcal{F}_{\mathrm{ledger}}$
>
> **recv** $(\texttt{Update}, [\texttt{Validate}, \mathit{txID}, \mathit{pid}, \mathit{txID}_1, \ldots, \mathit{txID}_l, \mathit{id}_1^a, \ldots, \mathit{id}_o^a], \mathit{internalState})$ **from** $\texttt{I/O}$:
>
>> $\{\mathcal{A} \textit{ is allowed to access validation information of tx for every party involved in the tx and in appropriate context}$
>
>> **send** $\texttt{getCurrentKnowledge}$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{storage}}^{\mathbf{c}} : \texttt{storage})$ $\qquad$ $\{\textit{Get current knowledge of } \mathsf{pid}_{\mathsf{cur}}$
>> **wait for** $(\texttt{getCurrentKnowledge}, \mathit{txGraph}_{\mathsf{pid}_{\mathsf{cur}}}, \mathit{attachments}_{\mathsf{pid}_{\mathsf{cur}}})$
>> **send** $(\texttt{GetContent}, \texttt{tx}, \mathit{txID})$ **to** $(\mathsf{pid}_{\mathsf{cur}}, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{storage}}^{\mathbf{c}} : \texttt{storage})$ $\qquad$ $\{\textit{Get transaction details.}$
>> **wait for** $(\texttt{GetContent}, \texttt{tx}, \mathit{tx})$
>
>> **if** $\{\mathit{txID}_1, \ldots, \mathit{txID}_l\}$ not part of $\mathit{txGraph}_{\mathsf{pid}_{\mathsf{cur}}} \vee \{\mathit{id}_1^a, \ldots, \mathit{id}_o^a\}$ not part of $\mathit{attachments}_{\mathsf{pid}_{\mathsf{cur}}}$: $\qquad$ $\begin{cases} \textit{Check that } \mathcal{A} \textit{ provides a} \\ \textit{valid context} \end{cases}$
>>> **reply** $(\texttt{Update}, \epsilon, \epsilon, \epsilon)$ $\qquad$ $\{\textit{Validation declined}$
>> Remove all entries from $\mathit{txGraph}_{\mathsf{pid}_{\mathsf{cur}}}$ that are not in $\{\mathit{txID}_1, \ldots, \mathit{txID}_l\}$
>> Remove all entries from $\mathit{attachments}_{\mathsf{pid}_{\mathsf{cur}}}$ that are not in $\{\mathit{id}_1^a, \ldots, \mathit{id}_o^a\}$
>> $\mathit{leakage} \leftarrow \texttt{executeValidation}(\mathit{tx}, \mathit{txGraph}_{\mathsf{pid}_{\mathsf{cur}}}, \mathit{attachments}_{\mathsf{pid}_{\mathsf{cur}}}) \wedge \neg\texttt{isDoubleSpend}(\mathit{tx}))$
>>> $\{\textit{Validity check for tx in the provided context}$
>> **parse** $[(\mathit{inputTx}_1, \mathit{idx}_1), \ldots, (\mathit{inputTx}_l, \mathit{idx}_l)]$ **from** tx
>> **reply** $(\texttt{Update}, \epsilon, \epsilon, \mathit{leakage})$ $\qquad$ $\{\textit{Return the validity of the transaction to } \mathcal{A}$
>
> **recv** $(\texttt{Update}, (\texttt{GetID}, \mathit{type}, \mathit{msg}), \mathit{internalState})$ **from** $\texttt{I/O}$: $\qquad$ $\begin{cases} \mathcal{A} \textit{ may query } \mathcal{F}_{\mathrm{ledger}}^{\mathbf{c}} \textit{ for ids} \end{cases}$
>> **send** $(\texttt{GetID}, \mathit{type}, \mathit{msg})$ **to** $\_, \mathsf{sid}_{\mathsf{cur}}, \mathcal{F}_{\mathrm{storage}}^{\mathbf{c}} : \texttt{storage})$ $\qquad$ $\{\textit{Request Id at } \mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$
>> **wait for** $(\texttt{GetID}, \mathit{type}, \mathit{id})$
>> $\mathit{leakage} \leftarrow \mathit{id}$
>> **reply** $(\texttt{Update}, \epsilon, \epsilon, \mathit{leakage})$ $\qquad$ $\{\textit{Return the id of the requested object to } \mathcal{A}$

Fig. 43. The update functionality $\mathcal{F}_{\mathrm{update}}^{\mathbf{c}}$ of $\mathcal{F}_{\mathrm{ledger}}^{\mathbf{c}}$ (Part 2)

Procedures and Functions:

 function executeBasicChecks :
  if $\exists$ a tx push msg in $requestQueue$: $\big\{$ *Valid transactions pushed between entities need to be in the state*
                              before further updates
    reply (Update, $\emptyset, \emptyset, \epsilon$) $\{$ *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$_{\mathrm{cur}}$, sid$_{\mathrm{cur}}$, $\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$: storage) $\big\{$ *Get IDs from $\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$*
    wait for (GetID, id)
    $reqQueue$.add([id, msg'], $sRound, pid'$)
  if $msg \neq [(txID_1, \ldots, txID_l), [(id_1^a, pid_i^a), \ldots, (id_1^a, pid_i^a)],$ $\{$ *Check message format*
   $(txID_{i_1}, tx_{i_1}, pid_{i_1}, r_{i_1}), \ldots, (txID_{i_h}, tx_{i_h}, pid_{i_h}, r_{i_h}),$
   $(id_1^a, attachment_1, pid_1), \ldots, (id_m^a, attachment_m, pid_m)]$:
    reply (Update, $\emptyset, \emptyset, \epsilon$) $\{$ *Processing aborted*

  if $\exists txID_i, txID_j, i, j \in [l], i \neq j$ in $(txID_1, \ldots, txID_l)$, **s.t.** $txID_i = txID_j$: $\big\{$ *Check that no txID*
                             is inserted twice
    reply (Update, $\emptyset, \emptyset, \epsilon$) $\{$ *Processing aborted*
  for all $(txID, tx, pid', r)$ from $[d_1, \ldots, d_h]$ in msg **do:** $\big\{$ *Register txIDs for $\mathcal{A}$'s inputs, d_j of form*
                          $(txID_{i_j}, tx_{i_j}, pid_{i_j}, r_{i_j})$
    send (SetID, tx, $txID$, tx) **to** (pid$_{\mathrm{cur}}$, sid$_{\mathrm{cur}}$, $\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$: storage)
    wait for (SetID, $accepted$)
    if $accepted = \mathtt{false}$: $\{$ *Abort processing*
     reply (Update, $\emptyset, \emptyset, \epsilon$)
  for all $(txID, tx, pid', r)$ from $[d_1, \ldots, dh]$ in msg **do:** $\big\{$ *Check correct format/identies/roles, d_j of form* $\big\{$ *Check*
                      $(txID_{i_j}, tx_{i_j}, pid_{i_j}, r_{i_j})$ *correct*
    if $tx \neq [\mathtt{tx}, (initiator, [signee_1, \ldots, signee_m], notary, formerNotary, proposal)]$:   *format*
     reply (Update, $\emptyset, \emptyset, \epsilon$)
    if $\neg[initiator, signee_1, \ldots, signee_m$ are clients ($role = \mathtt{client}$, pid prefixed by $\mathtt{client}$)$\wedge$ $\big\{$ *Check correct identi-*
     $notary, formerNotary$ are notaries ($role = \mathtt{notary}$, pid prefixed by $\mathtt{notary}$) $\wedge$ *ties and roles*
     $\forall pid \in \{initiator, signee_1, \ldots, signee_m, notary, formerNotary\} : pid \in identities$]:
     reply (Update, $\emptyset, \emptyset, \epsilon$)
    if $accepted = \mathtt{false}$: $\{$ *Abort processing*
     reply (Update, $\emptyset, \emptyset, \epsilon$)
  for all $(id^a, attachment, pid')$ in $(d_1, \ldots, d_m)$ in msg **do:** $\big\{$ *Register attachment IDs for $\mathcal{A}$'s inputs, d_j are of form*
                         $(id_j^a, attachment_j, pid_j)$
    send (SetID, $attachment$, id^a, $attachment$) **to** (pid$_{\mathrm{cur}}$, sid$_{\mathrm{cur}}$, $\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$: storage)
    wait for (SetID, $accepted$)
    if $accepted = \mathtt{false}$:
     reply (Update, $\emptyset, \emptyset, \epsilon$) $\{$ *Processing aborted*
  for all $(id', pid') \in \{(id_1^a, pid_i^a), \ldots, (id_1^a, pid_i^a)\}$ **do:** $\big\{$ *Get attachment content*
    send (GetContent, $attachment$, id') **to** ($_{-}$, sid$_{\mathrm{cur}}$, $\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$: storage)   *from $\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$*
    wait for (GetContent, $attachment$, $attachment'$)
    if ($_{-}$, ($attachment'$, $attachment'$), $_{-}$, pid') $\notin requestQueue$:
     reply (Update, $\emptyset, \emptyset, \epsilon$) $\{$ *Processing aborted*

Fig. 44. The update functionality $\mathcal{F}_{\mathrm{update}}^{\mathbf{c}}$ of $\mathcal{F}_{\mathrm{ledger}}^{\mathbf{c}}$ (Part 3)

Participating roles: {storage}
Corruption model: *incorruptible*
Protocol parameters:
  – $\eta \in \mathbb{N}$ {*The security parameter, defining the length of a "hash-value"*

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

Implemented role(s): {storage}
Internal state:
  – labels $\subset \{0,1\}^* \times \{\text{tx}, \text{attachment}\} \times \{0,1\}^*$.   {*Storage for IDs of transactions and attachments*
  – hasAccess{tx, attachment} $\times \{0,1\}^*$ {*Bookkeeping for data directly known by $\mathcal{A}$*
CheckID(pid, sid, $role$):
    Accept all messages with the same sid.
Main:

    recv (GetID, $type$, msg) **from** I/O **s.t.** $type \in \{\text{tx}, \text{attachment}\}$:   {*Request for IDs*

        if $\exists (id, \_, msg) \in$ labels for some $id \in \{0,1\}^*$:
            reply (GetID, $type$, id)   {*If ID exists, return it*
        else:
            send responsively (SetID, $type$) **to** NET
            wait for (SetID, $type$, id)
            while $[|id| \neq \eta] \vee [\exists(id, \_, \_) \in$ labels] **do**   {*Ensure correct length of the id and its uniqueness*
               send responsively (SetID, $type$) **to** NET
               wait for (SetID, $type$, id)
            reply (GetID, $type$, id)   {*Return newly generated id*

    recv (SetCorrID, $type$, $idProposal$, msg) **from** NET **s.t.** $type \in \{\text{tx}, \text{attachment}\}$:   {*Store IDs*

        if $[\exists(idProposal, type', msg') \in$ labels, **s.t.** $msg' \neq msg] \vee |idProposal| \neq \eta$:
                                 {*If id is already used for something else or length of the id is not η, reject it*
            reply (SetCorrID, false)
        else: {*Otherwise: store Id and return approved Id*
            labels.add($type$, $idProposal$, msg)   {*Store id*
            reply (SetCorrID, true)

    recv (GetContent, $type$, id) **from** I/O **s.t.** $type \in \{\text{tx}, \text{attachment}\}$:   {*Request for content*

        if $\exists (id, type, content) \in$ labels: {*Check whether requested object exists*
            reply (GetContent, $type$, $content$)
        else: {*Otherwise: return ϵ*
            reply (GetContent, $type$, ϵ)

    recv (getTxGraph, $internalState$, $mode$) **from** I/O:
        $msglistID \leftarrow \emptyset$
        if $mode = $ incBuffer: {*In incBuffer mode, include buffer to graph generation*
            $transactions \leftarrow \{tx | (\_, \_, \text{tx}, msg, \_, \_) \in msglist \vee (\_, tx, \_, \_) \in requestQueue \wedge tx$ is of form $[\text{tx}, (initiator, [signee_1, \ldots,$
            $signee_m], notary, formerNotary, proposal)]$
        else: {*Only "finalized" messages are in the graph*
            $transactions \leftarrow \{tx | (\_, \_, \text{tx}, msg, \_, \_) \in msglist \wedge tx$ is of form $[\text{tx}, (initiator, [signee_1, \ldots, signee_m], notary, formerNotary,$
            $proposal)]$

        for all $tx \in transactions$ **do:**
            Let $txID$ **s.t.** $(txID, \text{tx}, msg) \in$ labels   {*Extract tx ids*
            $txAttachments \leftarrow \emptyset$
            parse $(id_1^a, \ldots, id_l^a)$ **from** msg {*Extract used attachments*
            for all $id^a \in \{id_1^a, \ldots, id_l^a\}$ **do:**   {*Connect attachments to transactions*
               Let $attachment$ **s.t.** $(id^a, \text{attachment}, attachment) \in$ labels   {*Extract attachment content*
               $txAttachments.$add(id^a, $attachment$)
            $msglistID.$add($txID$, msg, $txAttachments$)

                                  {*Generate a (most likely disconnected) directed graph over all tx in the state.*
                                  {*Clients are transactions msg (including the full attachment information they*
        $txGraph \leftarrow$ buildTxGraph($msglistID$)   {*need), identified by txID and edges are from a transaction that consume a tx*
                                  {*to its input, i.e., $(txID', txID)$, **s.t.** $(txID', outputID)$ is a consumed input*
                                  {*in $(txID)$*

        reply (getTxGraph, $txGraph$)   {*Return full (notarised) tx graph*

Fig. 45. Shared storage $\mathcal{F}_{\text{storage}}^{\mathbf{c}}$ for subroutines in $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$ (Part 1)

Description of $M_{\mathrm{storage}}^{\mathbf{c}}$ (continued):

Main:

> **recv** getKnowledge **from** I/O: $\{\mathcal{F}_{\mathrm{ledger}}^{\mathbf{c}}$ *subroutine may ask for the possible knowledge of* $\mathrm{pid}_{\mathrm{cur}}$
> > $transactions \leftarrow \emptyset, attachments \leftarrow \emptyset$
> > Execute getTxGraph call above in normal mode and store output in $txGraph$
> > **for all** $(\_, \_, \mathrm{tx}, msg, \_, \_) \in msglist$ **do:** $\{$*Collect knowledge from msglist*
> > > **parse** $(initiator, signee_1, \ldots, signee_m, notary, formerNotary)$ **from** msg
> > > Let id **s.t.** $(id, \mathrm{tx}, msg) \in$ labels $\{$*Get txID*
> > > **if** $\mathrm{pid}_{\mathrm{cur}} \in \{initiator, signee_1, \ldots, signee_m, notary, formerNotary\}$:
> > > > $txSubGraph \leftarrow$ getConnectedSubGraph$(msg, txGraph)$
> > > >
> > > > $\left\{\begin{array}{l} \text{getConnectedSubGraph}(msg, txGraph) \text{ outputs the (maximal) connected subgraph of } txGraph \text{ such that } msg\text{'s} \\ \text{outputs are all edges of this subgraph ("maximal" means } \nexists \text{ a connected subgraph } txGraph \text{ containing } msg\text{'s} \\ \text{output and that is a superset of } \text{getConnectedSubGraph}(msg, txGraph)) \end{array}\right.$
> > > >
> > > > **for all** $(txID, tx, attachments_{tx})$ in $txSubGraph$ **s.t.** $txId$ is an input to msg **do:**
> > > > > $transactions.\mathrm{add}(txId, tx); attachments.\mathrm{add}(attachments_{tx})$ $\left\{\begin{array}{l} Leak \;\; txId, \;\; content, \;\; and \\ connected \; attachments \end{array}\right.$
> >
> > Execute getTxGraph call above in incBuffer mode and store output in $txGraph$
> > **for all** $(\_, msg, \_, \mathrm{pid}_{\mathrm{cur}}) \in requestQueue$ **do:** $\{$*Collect knowledge from requestQueue*
> > > **parse** $(initiator, signee_1, \ldots, signee_m, notary, formerNotary)$ **from** msg
> > > Let id **s.t.** $(id, \mathrm{tx}, msg) \in$ labels $\{$*Get txID*
> > > **if** $\mathrm{pid}_{\mathrm{cur}} = initiator$: $\{$*tx initiator has access to the full subgraph "below" a transaction*
> > > > $txSubGraph \leftarrow$ getConnectedSubGraph$(msg, txGraph)$
> > > >
> > > > $\left\{\begin{array}{l} \text{getConnectedSubGraph}(msg, txGraph) \text{ outputs the (maximal) connected subgraph of } txGraph \text{ such that } msg\text{'s} \\ \text{outputs are all edges of this subgraph ("maximal" means } \nexists \text{ a connected subgraph } txGraph \text{ containing } msg\text{'s} \\ \text{output and that is a superset of } \text{getConnectedSubGraph}(msg, txGraph)) \end{array}\right.$
> > > >
> > > > **for all** $(txID, tx, attachments_{tx})$ in $txSubGraph$ **s.t.** $txId$ is an input to msg **do:**
> > > > > $transactions.\mathrm{add}(txId, tx); attachments.\mathrm{add}(attachments_{tx})$
> > > **else if** $\mathrm{pid}_{\mathrm{cur}} \in \{signee_1, \ldots, signee_m, notary, formerNotary\}$: $\left\{\begin{array}{l} Signees \; and \; notaries \; only \; have \\ access \; to \; the \; full \; subgraph \; "be\text{-} \\ low" \; a \; transaction \; if \; the \; initia\text{-} \\ tor \; has \; the \; right \; to \; dispatch \; the \\ data \end{array}\right.$
> > > > $transaction.\mathrm{add}(id', msg)$ $\{$*Add tx* $\mathrm{pid}_{\mathrm{cur}}$ *is involved in to knowledge*
> > > > **if** $\exists(\_, msg, \_, initiator) \in requestQueue \lor \exists(\_, \_, \mathrm{tx}, msg, \_, initiator) \in msglist$:
> > > > > $txSubGraph \leftarrow$ getConnectedSubGraph$(msg, txGraph)$
> > > > >
> > > > > $\left\{\begin{array}{l} \text{getConnectedSubGraph}(msg, txGraph) \text{ outputs the (maximal) connected subgraph of } txGraph \text{ such that } msg\text{'s} \\ \text{outputs are all edges of this subgraph ("maximal" means } \nexists \text{ a connected subgraph } txGraph \text{ containing } msg\text{'s} \\ \text{output and that is a superset of } \text{getConnectedSubGraph}(msg, txGraph)) \end{array}\right.$
> > > > >
> > > > > **for all** $(txID, tx, attachments_{tx})$ in $txSubGraph$ **s.t.** $txId$ is an input to msg **do:**
> > > > > > $transactions.\mathrm{add}(txId, tx); attachments.\mathrm{add}(attachments_{tx})$
> > **for all** $(ctr_{temp}, attachment, r, pid) \in msglist$ **do:**
> > > **send** $(\mathrm{GetID}, attachment, attachment)$ **to** $(\mathrm{pid}_{\mathrm{cur}}, \mathrm{sid}_{\mathrm{cur}}, \mathcal{F}_{\mathrm{storage}}^{\mathbf{c}})$ $\left\{\begin{array}{l} Request \; id \; of \; the \; attach\text{-} \\ ment \end{array}\right.$
> > > **wait for** $(\mathrm{GetID}, attachment \, id)$ $\{\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$ *returns an id or* $\mathtt{false}$
> > > $attachments.\mathrm{add}(id, attachment)$
> > Add all transaction/attachments from hasAccess and all transactions leaked (according to *transcript*) to *transactions*, resp. *attachments*
> > **reply** $(\text{getKnowledge}, transactions, attachments)$ $\{$*Return knowledge*
>
> **recv** $(\text{hasAccess}, type, id, mode)$ **from** I/O: $\{$*Check whether* $\mathcal{A}$ *knows this object*
> > **if** $mode \neq$ subgraph:
> > > **if** $type, id \in$ hasAccess $\lor type, id$ was leaked according to *transcript*:
> > > > **reply** $(\text{hasAccess}, \mathtt{true})$
> > > **else:**
> > > > **reply** $(\text{hasAccess}, \mathtt{false})$
> > **else if** $mode =$ subgraph $\land type = \mathrm{tx}$:
> > > **if** $type, id \in$ hasAccess $\lor type, id$ was leaked according to *transcript* and all dependent tx/attachments in the id's subgraph are in hasAccess or leaked:
> > > > **reply** $(\text{hasAccess}, \mathtt{true})$
> > > **else:**
> > > > **reply** $(\text{hasAccess}, \mathtt{false})$
> > **else:**
> > > **reply** $(\text{hasAccess}, \mathtt{false})$

Fig. 46. Shared storage $\mathcal{F}_{\mathrm{storage}}^{\mathbf{c}}$ for subroutines in $\mathcal{F}_{\mathrm{ledger}}^{\mathbf{c}}$

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

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

Implemented role(s): {updRnd}
Subroutines: $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$: 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 variables it includes*}

      send (getTxGraph, $internalState$, ϵ) **to** (pid_{cur}, sid_{cur}, $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$: storage)   {*Generate transaction graph, located in $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$*}
      wait for (getTxGraph, $txGraph$)
      for all $(ctr_{temp}, tx', submissionRound, pid') \in requestQueue$ **do:**
         send (GetID, tx, tx') **to** (pid_{cur}, sid_{cur}, $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$: storage)   {*Get tx id*}
         wait for (GetID, $txID'$)
         $attachments' \leftarrow \epsilon$
         parse $(id^a_1, \ldots, id^a_l)$ **from** tx' {*Extract used attachments*}
         for all $id^a \in \{id^a_1, \ldots, id^a_l\}$ **do:**   {*Connect attachments to transactions*}

            send (GetContent, attachment, id^a) **to** (pid_{cur}, sid_{cur}, $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$: storage)   {*Get attachment from $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$*}
            wait for (GetContent, attachment, $attachment$)
            $attachments'$.add(id^a, $attachment$)
         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 $txGraph'$ according to the explanation above*}

            $txSubGraph \leftarrow$ getConnectedSubGraph(tx, $txGraph'$)   {*getConnectedSubGraph(msg, $txGraph$) outputs the (maximal) connected subgraph of txGraph such that msg's outputs are all edges of this subgraph ("maximal" means $\nexists$ a connected subgraph txGraph containing msg's output and that is a superset of getConnectedSubGraph(msg, $txGraphs$))*}

            $lastAgreement \leftarrow$ findLastAgreement(tx, $requestQueue$)   {*findLastAgreement(tx, $requestQueue$) outputs the round of the last agreement to tx in requestQueue.*}

            if participantsAgreed(tx) $\land lastAgreement + (3 + 4 \cdot |txSubGraph|) \cdot \delta > round$:

                {*If initiator and all signees agree on a transaction, it should be part of the state after $(1 + |txSubGraph|) \cdot \delta$ rounds. participantsAgreed(tx) outputs* true *if all signees and the initiator are not corrupted and for all signees and the initiator theres exist a submit request for the tx in requestQueue.*}

               reply (UpdateRound, false, ϵ)
   reply (UpdateRound, true, ϵ)

Fig. 47. The round update/time update functionality $\mathcal{F}^{\mathbf{c}}_{\mathrm{updRnd}}$ of $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$

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

Description of $M^{\mathbf{c}}_{\mathrm{leak}}$:

Implemented role(s): {leak}
CheckID(pid, sid, $role$):
   Accept all messages with the same sid.
Subroutines: $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$: 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}, $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$: storage)   {*Query pid_{cur}'s "state" at $\mathcal{F}^{\mathbf{c}}_{\mathrm{storage}}$*}
      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 send/is involved in*}

Fig. 48. The leakage subroutine $\mathcal{F}^{\mathbf{c}}_{\mathrm{leak}}$ of $\mathcal{F}^{\mathbf{c}}_{\mathrm{ledger}}$

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

Participating roles: {init}
Corruption model: uncorruptable
Protocol parameters:
    – networkmap $\subset \{0,1\}^* \times \{\texttt{client}, \texttt{notary}\}$

$\left\{\begin{array}{l} \textit{Map of identities exisiting in the network containing tuples of} \\ (\textit{identfier}, \textit{type}). \ \textit{We expect that the role prefixes the pid in} \\ \textsf{networkmap} \end{array}\right.$

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

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

        recv Init: $\left\{\begin{array}{l}\textit{We allow } \mathcal{A} \textit{ to query } \mathcal{F}_{\text{init}}^{\mathbf{c}} \textit{ as well}\end{array}\right.$
            $\textit{identities} \leftarrow \emptyset, \textit{msglist} \leftarrow \emptyset, \textit{ctr} \leftarrow 0$
            for all $(\textit{pid}, \textit{protocolRole}) \in$ networkmap **do:**
                $\textit{identities}.\texttt{add}(\textit{pid}, 0)$ $\left\{\begin{array}{l}\mathcal{F}_{\text{init}} \textit{ provides the network map to requestors}\end{array}\right.$
                $\textit{msglist}.\texttt{add}(\textit{ctr}, 0, \texttt{meta}, (\textit{pid}, \textit{protocolRole}), \perp, \perp)$
                $\textit{ctr} \leftarrow \textit{ctr} + 1$
            reply (Init, $\textit{identities}, \epsilon, \epsilon, \epsilon$)

Fig. 49. The initialization functionality $\mathcal{F}_{\text{init}}^{\mathbf{c}}$ of $\mathcal{F}_{\text{ledger}}^{\mathbf{c}}$