ROLLERBLADE: Replicated Distributed Protocol Emulation on Top of Ledgers

Dionysis Zindros
Stanford University

Apostolos Tzinas
National Technical University of Athens

David Tse
Stanford University

Abstract—**We observe that most fixed-party distributed protocols can be rewritten by replacing a *party* with a *ledger* (such as a blockchain system) and the authenticated channel communication between parties with cross-chain relayers. This transform is useful because blockchain systems are always online and have battle-tested security assumptions. We provide a definitional framework that captures this analogy. We model the transform formally, and posit and prove a generic *metatheorem* that allows translating all theorems from the *party* setting into theorems in the *emulated* setting, while preserving analogies between party *honesty* and ledger *security*. In the heart of our proof lies a reduction-based simulation argument. As an example, our metatheorem can be used to construct a consensus protocol on top of other blockchains, creating a reliable rollup that assumes only the majority of the underlying layer-1s are secure.**

1. Introduction

A distributed ledger protocol is envisioned to play the role of a "world computer". This world computer, evolving its state through State Machine Replication, ensures execution is accurate as long as its ledger remains secure (is safe and live). This security is guaranteed as long as some majority of validators are honest.

As multiple ledger protocols become deployed, each of them functions as its own such "world computer". A natural question of recursive composability arises: Can we use these "world computers" as validators to run further protocols on top of them? For example, can we run an *overlay* ledger protocol on top of existing *underlying* ledger protocols, treating the underlying ledger protocols as *computers* which take the role of a *validator* in the overlay protocol?

In this paper, we observe that many distributed systems protocols, among others distributed ledger protocols e.g., Streamlet and HotStuff, can be run on top of other existing long-running and battle-tested ledger *underlying* protocols such as Bitcoin, Ethereum, Cardano, and Algorand. The underlying protocols play the role of always-online validators that participate in the overlay protocol's execution. If we run a consensus protocol on top of existing consensus protocols, we can realize a rollup, in the form of the overlay, which is securer than each of the constituent underlying Layer-1s it is based on. For example, we can construct a rollup that maintains security even if one of Bitcoin, Ethereum, Cardano, or Algorand faces a catastrophic failure such as a persistent 51% attack.

Our construction is quite generic. The class of overlay protocols our system can run is not limited to consensus protocols, but can be any distributed protocol, among others Reliable Broadcast, or a data availability protocol, as long as it satisfies a minimal set of axioms: It must not use any internally-generated randomness, and must be designed to work in the commonly used *authenticated channels* network model. The underlying ledger protocols must also satisfy a minimal set of axioms which are satisfied by all popular blockchain and related protocols today: It must realize a ledger functionality (with the ability to *write* transactions and *read* sequences of transactions in the form of a ledger) which promises to be secure; it must ascribe roughly accurate timestamps to each transaction on the ledger (a temporal ledger); it must allow the recording of arbitrary strings inside a transaction (a bulletin board, similar to Bitcoin's `OP_RETURN`); and it must support non-interactive clients (the ledgers must be *transcribable* into a string that can recover the original ledger). The minimal set of axioms are satisfied by Bitcoin (Nakamoto), Ethereum, Cardano (Ouroboros/Ouroboros Praos/Ouroboros Genesis), Algorand, Monero, Sui (HotStuff/Bullshark/Narwhal/Tusk), and all other distributed ledger protocols to our knowledge. Notably, we don't require that the underlying protocols have any smart contract support (although such support can greatly increase the efficiency of our protocol), or that existing bridging is implemented between them (as transcribability is enough to realize this functionality ourselves). Our construction works on top of both proof-of-work and proof-of-stake blockchains, among others.

Our contributions. The contributions of this paper are the following:

1) We formally define the *compositing problem*, of emulating one distributed protocol in a replicated fashion on top of other distributed protocols.
2) We put forth *rollerblade*, a generic method that allows transforming distributed systems protocols from the *party* to the *emulated* setting.
3) We show how our protocol works on top of any proof-of-work or proof-of-stake blockchain with minimal axiomatic assumptions. We give the exact properties

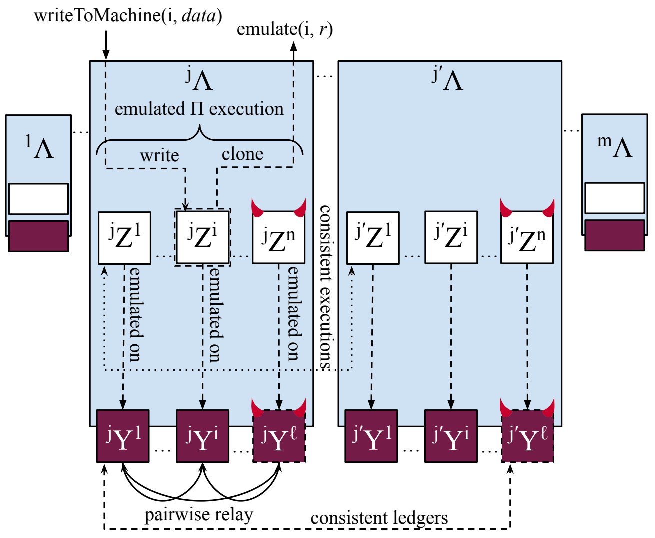

Figure 1. An overview of the ROLLERBLADE construction. There are m ROLLERBLADE clients $^1\Lambda, \ldots, {}^j\Lambda, \ldots, {}^m\Lambda$. Each $^j\Lambda$ such client of them is executing n instances $^j\mathsf{Z}^1, \ldots, {}^j\mathsf{Z}^i, \ldots, {}^j\mathsf{Z}^n$ of the overlay protocol Π within it. To emulate the i^{th} instance $^j\mathsf{Z}^i$, the respective underlying protocol $^j\mathsf{Y}^i$ is used (here, $n = \ell$). These underlying protocols are pairwise relayed to each other (shown only in $^j\Lambda$ for clarity). Some of the underlying protocols may be rendered insecure (red horns); their respective $^j\mathsf{Z}^i$ instances behave dishonestly (red horns). The construction will ensure that the executions of $^j\mathsf{Z}^i$ and $^{j'}\mathsf{Z}^i$ are "consistent" if i is a good underlying protocol.

required of the underlying ledger protocols and observe that smart contract capabilities and certificates are not necessary (but are helpful). Our protocols can run on top of Bitcoin, Ethereum, Cardano, etc.

4) We precisely define the formulation of distributed systems protocols required so that they can undergo the ROLLERBLADE transform.
5) We prove our generic transformation always yields good results by using a reduction-based simulation argument connecting the ledger setting and the party setting. The analogies elicited through our transform give insights about the functionality of ledger protocols more broadly and may be of independent interest.

Practical efficiency. Our construction is about what compositions of ledgers are theoretically *possible*, so we will not be concerned with the efficiency of our construction beyond the theoretical desire to remain polynomial. Our treatment is generic, and our aim is to formulate a minimal set of axioms required of underlying and overlay protocols to render them composable. Due to the generality of our construction, certain optimizations will not be possible, but efficiency can be greatly increased for concrete underlying protocols, for example leveraging smart contract capabilities if they are available. For a particular example on Tendermint, see TrustBoost [1].

Construction overview and paper structure. Our goal is to *emulate* the execution of a distributed protocol Π, the *overlay protocol*, executing across multiple parties. Each of these complete emulations will take place within the confines of a single machine: a *compositor client* $^j\Lambda$. Each compositor client $^j\Lambda$ will execute multiple Π parties $^j\mathsf{Z}^1, \ldots, {}^j\mathsf{Z}^n$, and these parties will be allowed to communicate with one another. For example, Π could be a distributed consensus protocol such as IT Streamlet [2], and each $^j\mathsf{Z}^i$ could take the role of a validator within that consensus protocol. We wish this execution to be *replicated* across different Λs: The emulation of Π in the view of $^j\Lambda$ should be consistent with the emulation of Π in the view of $^{j'}\Lambda$. We call the protocol Λ that allows this replicated emulation the *compositor protocol*. A compositor protocol is parametrized by Π and prescribes how to emulate Π in a replicated manner. We give the detailed definitions of compositors and the desiderata (of emulation and replication consistency) in Section 3. In order to perform this emulation in a replicated manner, the different compositor clients must use some shared infrastructure $\mathcal{Y}$ that enables the clients to communicate and maintain consistency in their replication. We term these pre-existing protocols that the compositor leverages the *underlying protocols*.

We give one construction of a compositor protocol, which we term ROLLERBLADE. ROLLERBLADE works by using existing *distributed ledger protocols* as its underlying infrastructure. For example, the underlying infrastructure could be Bitcoin, Ethereum, Cardano, etc. Each of the $^j\Lambda$ clients maintains a full node to each of the underlying ledgers. The protocol Π that can be emulated on top can be *any* distributed protocol, among others a consensus protocol, a data availability protocol, a distributed auction protocol, reliable broadcast etc. We give the detailed ROLLERBLADE construction in Section 4.

The construction briefly works as follows. Each of the ROLLERBLADE clients emulates *each* party of the overlay protocol locally, by using the respective underlying ledger protocol as a "guide" to the overlay party's execution. The respective underlying ledger indirectly records all the user input to the overlay protocol parties, as well as the network messages exchanged between the overlay participants. Any ROLLERBLADE client can attempt to *write* an input to any of the overlay party emulations. This is performed by recording this *write instruction* to the respective underlying ledger. In case the ledger written to is secure, the other ROLLERBLADE clients will also receive the instruction in question and replicate it within their own emulations.

The state of each ledger is relayed to each other ledger by helpful but untrusted *relayers*, at least one of which is assumed to be honest (any ROLLERBLADE client that does not trust the relayers can relay the data themselves). When a checkpoint of a source ledger appears within a target ledger, this corresponds to a network message exchange from the emulated source party to the emulated target party. Whereas no network messages are recorded on-ledger at any time, the ROLLERBLADE client can emulate, off-ledger, the network messages that *would have been sent* by the source by looking at the checkpointed source ledger data and performing a recursive emulation.

To prove our construction realizes the notions of emulation and replication consistency, we state and prove our main results in Theorem A.1 (replication consistency) and Theorem A.2 (emulation consistency). The Emulation Theorem compares the execution of the overlay protocol in the emulated setting within the confines of *one* ROLLERBLADE client, and some execution in the stand-alone *party* setting, with no emulation at all. Secondly, the Cross-Party Lemma concerns the execution of *multiple* compositor clients. It states that the execution of a emulated party within one client is the "same" as the execution of the same emulated party within a different client, as long as the respective underlying ledger is secure. Our main proof is a simulation-based reduction proof in an execution-driven model. The flavour of our proofs is reminiscent of the stand-alone version of Canetti's Universal Composability, albeit simpler. Our central result is that each party of Π is made to correspond to one underlying ledger, and the party's behavior is *honest* if the respective underlying ledger is *secure*. This ledger security requirement consists of the established notions of *safety* and *liveness*, as well as some additional axiomatization which we introduce in this work (most importantly the notion of *timeliness* and *temporal ledgers*). These lemmas are stated in Section B and proven in the Appendix.

2. Preliminaries

Notation. Given a sequence Y, we address it using $Y[i]$ to mean the i^{th} element (starting from 0). Negative indices address elements from the end, so $Y[-i]$ is the i^{th} element from the end, and $Y[-1]$ in particular is the last. We use $Y[i{:}j]$ to denote the subarray of Y consisting of the elements indexed from i (inclusive) to j (exclusive). The notation $Y[i{:}]$ means the subarray of Y from i onwards, while $Y[{:}j]$ means the subsequence of Y up to (but not including) j. We use $|Y|$ to denote the length of Y. We say that X is a, not necessarily strict, *prefix* of Y, denoted $X \preceq Y$ or $Y \succeq X$, when every element of X appears in Y at the same location (but Y may have more elements beyond $|X|$). We write $X[i] = \bot$ for $i \geq |X|$ to indicate that the sequence has been exhausted. We use $[n]$ to denote the set $\{1, \dots, n\}$. We write $X \,\|\, Y$ to mean the sequence obtained by concatenating the sequences X and Y. We use $X \overset{d}{=} Y$ to denote that the two random variables X and Y are equal in distribution. When A is a set whose natural ordering is implied (e.g., if it is a subset of natural numbers), we use A_i to denote the i^{th} element (1-based) of A under the natural order. We use $\text{supp}(X)$ to mean the support of distribution X, or, if X is a random variable, the support of its distribution.

Distributed Protocols. In this work, we are concerned with the composition of distributed protocols.

A distributed protocol Π is an interactive Turing machine (ITM). To simplify the exposition, we treat this ITM as an object-oriented class Π and the respective *party* (an Interactive Turing Machine Instance, or ITI) as an object (class instance).

Definition 2.1 (Distributed Protocol)**.** A *distributed protocol* is an ITM which exposes the follow methods:

- $\mathsf{construct}^{\text{NET}}(\Delta)$: This method called when the protocol is *instantiated* into a *party*. We denote this using the notation $\mathsf{new}\ \Pi^{\text{NET}}(\Delta)$, which returns a party that can be interacted with. It is also given oracle access to a network functionality NET and the network delay $\Delta \in \mathbb{N}$ to expect.
- $\mathsf{write}(\mathsf{data})$: Takes user input by accepting some *data* string.
- $\mathsf{read}()$: Produces user output in the form of some *data* string. We mandate that, upon its completion, the execution of a *read* instruction does not change the state of the interactive machine.
- $\mathsf{execute}()$: Executes a single round of the protocol. Within $\mathsf{execute}$, the machine can use the *netout* functionality of NET to send messages (as many times as it wishes) and the *netin* functionality of NET to receive messages (once per round).

The same protocol Π is *instantiated* into multiple instances, each called a *party* (conceptually running on a different computer each). We are interested in a population of parties, some of which are as *honest*, while others are designated as *adversarial*. All honest parties run the prescribed protocol Π, whereas adversarial parties may run any protocol of their choice (including Π). All adversarial parties are controlled by one colluding adversary $\mathcal{A}$.

Definition 2.2 (Permissioned Distributed Protocol)**.** A distributed protocol is *permissioned* if its *construct* method accepts, in addition to the network delay Δ, its own identity $j \in \mathbb{N}$ and the number $n \in \mathbb{N}$ of parties it will coordinate with.

Executions. Because Π can be randomized, we are interested in particular *executions* $\mathcal{E}$ of protocols. An execution captures everything that happened between the honest parties running the protocol and the adversary, including the local state of each party and the network messages that were exchanged. The execution concludes after a finite amount of time.

Definition 2.3 (*In vitro* invocation)**.** When multiple protocols are executed together in a collective execution $\mathcal{E}$, we can take a snapshot of an instance of a machine within the execution after some round r. After we have taken that snapshot, we can continue running the instance *outside* the execution by, for example, invoking some of its functions, without affecting the execution which has already concluded. We call this process an *in vitro* invocation (cf. Canetti's *in vitro* and *in vivo* [3]) after some round r.

Time. We model time as taking place in discrete *rounds* $1, 2, \dots$. The parties run in *lockstep*: During each round, each party is allowed to run some (finite) computation. Each party is initialized (by having its $\mathsf{construct}$ function called) before round 1 commences. Subsequently, round 1 starts by having the $\mathsf{execute}$ function of each party called, without any inputs provided by the environment (in the

form of "writes" or network inputs). By the end of each round, the machine can produce some network outputs to the environment. During every non-zero round, the environment makes some network inputs available to the party being executed, subject to the network constraints defined below.
Network. The parties are allowed to communicate with one another through an underlying *network*. We make use of two types of networks.

Definition 2.4 (Gossip Network)**.** A *gossip network* allows any party to *diffuse* a message m to the rest of the parties. The network ensures that, if the sender is honest, then every other honest party will receive the diffused message m.

Definition 2.5 (Authenticated Channels)**.** An *authenticated channels network* allows any party P to *point-to-point* send a message to another party Q. The network faithfully reports the sending party to the receiving party, and the adversary cannot forge the origin of messages.

Definition 2.6 (Synchrony)**.** A network is *synchronous* with parameter $\Delta \in \mathbb{N}$ if any message m sent by an honest party P at the end of round r is delivered by the beginning of round $r + \Delta$.

In the gossip network model, the adversary may introduce an arbitrary number of messages and forge their origin, and may also reorder messages before they are delivered to the honest parties. In the authenticated channels model, the adversary is not allowed to forge the origin of messages, but may send an arbitrary number of messages from each of the corrupted parties she controls. In both models, the adversary may send different messages to each honest party at each round. The delay of each honestly sent message is also adversarially controlled, as long as the bound Δ is respected. However, the adversary is not allowed to censor honestly produced messages after the bound Δ has elapsed.
Ledgers.

Definition 2.7 (Ledger)**.** A distributed protocol Π, together with a *transaction validity language* $\mathcal{V}$ is a *temporal ledger* protocol if its *read* and *write* functionalities have the following semantics:

- $\mathsf{write}(\mathsf{tx})$: The write functionality accepts a *transaction*, which is a string that belongs to $\mathcal{V}$.
- $L \leftarrow \mathsf{read}()$: The read functionality returns a *ledger* $L \in (\mathbb{N} \times \mathcal{V})^*$, which is a finite sequence of transactions in $\mathcal{V}$.

It is desirable that our ledger protocols produce executions that satisfy the following virtues. Let ${}^{P}\boldsymbol{L}_r$ denote the ledger reported by the honest party P issuing a *read* instruction to its ledger protocol at the end of round r.

Definition 2.8 (Safe)**.** An execution $\mathcal{E}$ of a ledger protocol Π is *safe* if for all rounds $r_1 \leq r_2 \in \mathbb{N}$ and all honest parties P_1, P_2 running instances Π^1, Π^2, we have that ${}^{P_1}\boldsymbol{L}_{r_1} \preceq {}^{P_2}\boldsymbol{L}_{r_2}$ or ${}^{P_1}\boldsymbol{L}_{r_1} \succeq {}^{P_2}\boldsymbol{L}_{r_2}$. Additionally, the ledger is *sticky*: ${}^{P_1}\boldsymbol{L}_{r_1} \preceq {}^{P_1}\boldsymbol{L}_{r_2}$.

The last requirement (${}^{P_1}\boldsymbol{L}_{r_1} \preceq {}^{P_1}\boldsymbol{L}_{r_2}$) that an honest ledger is locally append-only (sticky) can be easily enforced in any safe protocol without stickiness by having the parties report the longest ledger they have seen so far [4].

Definition 2.9 (Live)**.** An execution $\mathcal{E}$ of a ledger protocol Π is *live* with parameter $u \in \mathbb{N}$ if, whenever *all* honest parties attempt to write the transaction tx during rounds $r, r+1, \ldots, r+u-1$, the transaction appears in all honest ledgers at all rounds $r' \geq r + u$.

The above definition requires that *all* honest parties attempt to write a transaction during rounds $r, \ldots, r+u-1$. However, we assume, without loss of generality, that parties *gossip* any transaction they wish to write onto the underlying ledger. All honest parties that receive a gossiped transaction will also attempt to include it, and, so, if *one* honest party attempts to introduce a transaction at round r, the transaction will make it to everyone's ledger by round $r + \Delta + u$.

Definition 2.10 (Ledger Security)**.** An execution $\mathcal{E}$ of a ledger protocol Π is *secure* (with parameter u) if the execution is *safe* and *live* (with parameter u).

3. Definitions

Definition 3.1 (Temporal Ledger)**.** A distributed protocol Π, together with a *transaction validity language* $\mathcal{V}$ is a *temporal ledger* protocol if its *read* and *write* functionalities have the following semantics:

- $\mathsf{write}(\mathsf{tx})$: The write functionality accepts a *transaction*, which is a string that belongs to $\mathcal{V}$.
- $L \leftarrow \mathsf{read}()$: The read functionality returns a *ledger* $L \in (\mathbb{N} \times \mathcal{V})^*$, which is a finite sequence of pairs (r, tx) where $\mathsf{tx} \in \mathcal{V}$ is a transaction, and r is a *round* indicating the time at which the transaction in question is recorded on the ledger.

The following definition is first introduced in this work, as it will prove immensely useful for composability, but is a natural property that all well-designed blockchain systems have:

Definition 3.2 (Timely)**.** An execution $\mathcal{E}$ of a temporal ledger protocol Π is *timely* with parameter $v \in \mathbb{N}$ if for all honest parties P and rounds r_1 it holds that:

1) The rounds recorded in ${}^{P}\boldsymbol{L}_{r_1}$ are non-decreasing.
2) The round recorded at ${}^{P}\boldsymbol{L}_{r_1}[-1]$ is at most r_1.
3) For all $r_2 \geq r_1$, the rounds recorded in ${}^{P}\boldsymbol{L}_{r_2}[|{}^{P}\boldsymbol{L}_{r_1}|:]$ are newer than $r_1 - v$.

All popular blockchain protocols report timestamps together with their transactions and ensure their timeliness. For example, Bitcoin and Ethereum produce blocks each of which contains a timestamp. These timestamps can be copied into the transactions therein when *reading*. Because the respective protocols do not accept chains with decreasing timestamps, or blocks with future timestamps the timeliness points 1 and 2 are ensured. Point 3 is more subtle and asks that very old transactions will not suddenly appear in the ledger.

Definition 3.3 (Temporal Ledger Security)**.** An execution $\mathcal{E}$ of a temporal ledger protocol Π is *secure* (with parameters (u, v)) if the execution is *safe*, *timely* (with parameter v), and *live* (with parameter u).

3.1. The Setting

We are given $n \in \mathbb{N}$ ledger protocols $\mathsf{Y}^1, \mathsf{Y}^2, \ldots, \mathsf{Y}^i, \ldots, \mathsf{Y}^n$, the so-called *underlying* ledger protocols. While mathematically, these Ys are interactive Turing machines of ledger protocols, in practice, these are preexisting, already operational ledger protocol executions such as Bitcoin, Ethereum, and Cardano, for which we have access to already running full nodes and we are asked to compose on top of.

We are also given a distributed protocol Π (not necessarily a ledger protocol), the so-called *overlay* protocol. We will emulate an n-party execution of Π, with each i of these n parties corresponding to the underlying ledger protocol Y^i.

The users of the protocol are $m \in \mathbb{N}$ ROLLERBLADE *clients* termed ${}^1\mathsf{RB}, {}^2\mathsf{RB}, \ldots, {}^j\mathsf{RB}, \ldots, {}^m\mathsf{RB}$ (with, potentially $m \neq n$). *Each* ${}^j\mathsf{RB}$ client runs a separate full node ${}^j\mathsf{Y}^1, \ldots, {}^j\mathsf{Y}^i, \ldots, {}^j\mathsf{Y}^n$ for each of the underlying Ys. The RB nodes do not have direct network communication, but only use the read/write functionalities of their respective Y instances to communicate. For example, when party ${}^1\mathsf{RB}$ *writes* a transaction tx to its ${}^1\mathsf{Y}^1$ instance, this transaction will eventually appear in ${}^2\mathsf{RB}$'s ${}^2\mathsf{Y}^1$ instance ledger output, as long as Y^1 is live. We will use ${}^j\boldsymbol{L}^i_r \leftarrow {}^j\mathsf{Y}^i_r.\mathsf{read}()$ to refer to the ledger reported at round r by the full node instance ${}^j\mathsf{Y}^i$ running the underlying ledger protocol Y^i operated by the overlay party ${}^j\mathsf{RB}$ (this ledger, like all ledgers, is a sequence of round/transaction pairs, and its k^{th} round/transaction pair is ${}^j\boldsymbol{L}^i_r[k]$).

We now describe the requirements of our underlying and overlay protocols.

3.2. Rollerblade Underlying Requirements

Firstly, in order to record data on our underlying ledgers, we require that any arbitrary string can be written to them. This is called a *bulletin*.

Bulletins. A bulletin ledger protocol offers two additional functions encode and decode: $\mathsf{tx} \leftarrow \mathsf{encode}(s)$ and $s \leftarrow \mathsf{decode}(\mathsf{tx})$. The encode function takes a string s and encodes it into a transaction tx that can be *written* into the ledger and is guaranteed to be accepted. The decode function takes a transaction tx and, if it is a bulletin transaction, decodes it back into s. Otherwise, decode can return $\perp$. All transactions produced by encode are bulletin transactions, but the adversary can also introduce arbitrary bulletin transactions of her choice indiscriminately. The ledger may also include non-bulletin transactions among the bulletin transactions.

Definition 3.4 (Bulletin Board)**.** A ledger protocol Π accompanied by a pair of computable functions $(\mathsf{encode}, \mathsf{decode})$, of which decode is deterministic, is called a *bulletin board* if it holds that, for any $s \in \{0,1\}^*$, the output of $\mathsf{tx} = \mathsf{encode}(s)$ is always a valid transaction and that $\mathsf{decode}(\mathsf{encode}(s)) = s$.

Bulletins provide ordering and data availability of arbitrary data without checking any semantic validity. As such, they constitute a *lazy* use of a ledger [5], [6]. All popular blockchains such as, for example, Ethereum and Bitcoin are bulletins. Bitcoin allows the recording of arbitrary data using OP_RETURN transactions, whereas Ethereum allows such recording by including the data in the CALLDATA of a smart contract call, or in the parameters of an event.

Definition 3.5 (Certifiability)**.** A ledger protocol Π accompanied by a computable functionality transcribe and a computable deterministic (non-interactive) function untranscribe is called *certifiable*. The functionality transcribe is called on a full node and returns a *transcription* of its current ledger. The function untranscribe is called with the transcription as the parameter, and is hoped to return the original ledger. The certifiable protocol is *live* if, in addition to the liveness requirements of the ledger protocol, whenever $\mathsf{transcribe}()$ is called on an honest party P with a ledger L and returns a transcription τ, then $\mathsf{untranscribe}(\tau) = L$.

The certifiable protocol is *safe* if, in addition to the safety requirements of the ledger protocol, whenever an honest party P_1 executes $\mathsf{untranscribe}(\tau)$ at round r_1 with some (honestly or adversarially produced) transcription τ, then for all honest parties P_2 at round r_2, it holds that ${}^{P_1}\boldsymbol{L}_{r_1} \preceq {}^{P_2}\boldsymbol{L}_{r_2} \vee {}^{P_2}\boldsymbol{L}_{r_2} \preceq {}^{P_1}\boldsymbol{L}_{r_1}$.

3.3. Compositors

To aid the analysis, we assume that the ITIs contain all the previous transcript of their execution (a history of machine configurations).

A *compositor* is a protocol that runs an *overlay* distributed protocol Π on top of a set of n *underlying* distributed protocols $\mathsf{Y}^1, \ldots, \mathsf{Y}^i, \ldots, \mathsf{Y}^n$. The protocol Π is generally designed to work in the *party setting* with a network (such as an authenticated channels network) connecting its instances directly. The compositor Λ executes Π in the *emulated setting* by utilizing the underlying Y^i protocols to help with the emulation and communication between the overlay protocol instances. Multiple instances ${}^1\Lambda, \ldots, {}^j\Lambda, \ldots, {}^m\Lambda$ of the compositor are executed. Each j^{th} compositor instance promises to emulate the execution of multiple instances of Π (${}^j\mathsf{Z}^1, \ldots, {}^j\mathsf{Z}^i, \ldots {}^j\mathsf{Z}^n$). These emulated Z instances should behave similarly to instances of Π running in a stand-alone setting.

Definition 3.6 (Compositor)**.** A *compositor* Λ with overlay Π and underlying $\mathcal{Y} = \mathsf{Y}^1, \ldots, \mathsf{Y}^n$, is a family of interactive machines $\Lambda_{\Pi,\mathcal{Y}}$ providing the following functionalities:

1) construct $(\mathsf{sid}, ({}^j\mathsf{Y}^1 \ldots {}^j\mathsf{Y}^n))$.
2) writeToMachine (i, data).
3) emuSnapshot (i, r).

The compositor is constructed by calling *construct* with parameter the session identifier sid. Note that compositors are permissionless, as they don't know their instance identity $j \in \mathbb{N}$. Each compositor instance provides a *writeToMachine* functionality that allows writing data to the i^{th} emulated machine. It also provides an *emuSnapshot* functionality that promises to emulate the execution of the i^{th} instance of Π up to round r, and return the instance of this emulation at round r (and note that this contains the full transcript of the emulation). Observe that *emuSnapshot* can be invoked at a later round $r' > r$. Note here the difference between $\mathcal{Y} = \mathsf{Y}^1, \dots, \mathsf{Y}^n$ (denoting *underlying protocols* in the form of ITMs) and $({}^j\mathsf{Y}^1, \dots, {}^j\mathsf{Y}^n)$ (denoting *underlying protocol instances* in the form of ITIs). The compositor is parametrized with the former, and the latter are passed as parameters to the *construct* functionality.

Even though we will not do the analysis in the full Universal Composability (UC) framework, and we treat executions as stand-alone, we do adopt some of the notation of the UC framework. Let $\textsc{Run}^{\mathsf{ES}}(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$ denote the transcript of an execution of the compositor protocol Λ parametrized with overlay protocol Π, and underlying protocols $\mathcal{Y} = (\mathsf{Y}^1, \dots, \mathsf{Y}^n)$, adversary $\mathcal{A}$, and environment $\mathcal{Z}$. Following the notation of the UC framework, we define the notion of an *execution* and a *view*.

The Emulated Setting. In particular, in the emulated setting execution, denoted by $\textsc{Run}^{\mathsf{ES}}_{m,n}(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$, the environment $\mathcal{Z}$ is constrained by the control program to initially spawn a number $n \times \ell$ (where n is a parameter of the execution, and $\ell = |\mathcal{Y}|$) underlying protocol instances

$$\begin{array}{c} {}^1\mathsf{Y}^1, \dots, {}^1\mathsf{Y}^\ell, \\ \vdots, \ddots, \vdots \\ {}^m\mathsf{Y}^1, \dots, {}^m\mathsf{Y}^\ell\,, \end{array}$$

where ${}^j\mathsf{Y}^i$ denotes the j^{th} instance of the protocol Y^i. The environment facilitates the communication between the underlying protocol instances ${}^j\mathsf{Y}^i$ and ${}^{j'}\mathsf{Y}^i$ for all j, j', whereas protocols ${}^j\mathsf{Y}^i$ and ${}^{j'}\mathsf{Y}^{i'}$ with $i \neq i'$ are not allowed to directly communicate. Next, the environment is constrained to spawn a number m of compositor clients ${}^1\Lambda, \dots, {}^m\Lambda$ (where $m \in \mathbb{N}$ is a parameter of the execution), allowing the environment to choose the sid parameter, and passing the tuple $({}^j\mathsf{Y}^1, \dots, {}^j\mathsf{Y}^\ell)$ of ITIs to ${}^j\Lambda$'s *construct* functionality (in UC language, each of these ${}^j\mathsf{Y}^i$ is constrained to be used as a *subroutine* solely by ${}^j\Lambda$). All of those Λ compositor clients are honest and will remain so throughout the execution. Outside of those ${}^j\mathsf{Y}^i$ instances controlled by the Λs, each of the protocols Y^i may have more instances running within the execution, spawned by the environment and potentially corrupted by the adversary. For example, if Y^1 is the Bitcoin protocol, then the ${}^j\mathsf{Y}^i$ instance running within ${}^j\Lambda$ is a Bitcoin client, whereas the execution comprises also others Bitcoin clients and full nodes, potentially corrupted by the adversary. The execution proceeds in rounds $r = 1, \dots, R$ for a polynomial number R of rounds (where the polynomial is taken with respect to the security parameter). For every round r, the environment is constrained to first call the *execute* function of each ${}^j\mathsf{Y}^i$ for every $j, i \in \mathbb{N}$, as well as the *execute* function of Ys living outside of the clients Λ. Next, the environment must call the *execute* function of each ${}^j\Lambda$ sequentially. Finally, the environment is constrained to call the adversary $\mathcal{Z}$ (a rushing adversary). The adversary is allowed to corrupt the Y instances living outside of Λs (we will later impose constraints, in the form of *beliefs*, which the environment ensures the adversary must respect). At any time, the environment may choose to provide inputs to any of the ${}^j\Lambda$ parties by invoking their *writeToMachine* functionality with inputs of its choice.

The Party Setting. In the party setting execution, denoted by $\textsc{Run}^{\mathsf{PS}}_n(\Pi, \mathcal{A}, \mathcal{Z}, n, H, \Delta)$, the environment $\mathcal{Z}$ is constrained by the control program to spawn n parties $\Pi^1, \dots, \Pi^n$ (where $n \in \mathbb{N}$ is a parameter of the execution). Note here that Π is a protocol (an ITM), whereas each Π^i is an instance (an ITI). The adversary $\mathcal{A}$ is allowed to corrupt parties indexed by $[n] \setminus H$ at the beginning of the execution (a static corruption model). This is done by the adversary sending a message to the environment requesting to corrupt the desired party. The environment grants this corruption wish as long as it respects the requirement that the corruption falls within $[n] \setminus H$. The execution proceeds in rounds $r = 1, \dots, R$, where R, again, is a polynomial of the security parameter. At every round, the environment is constrained to first call the *execute* function of each Π^i for every honest party, in order. Next, the environment must call the adversary (again, a rushing adversary). The environment is constrained to deliver messages between honest parties in an authenticated manner, and to deliver messages within Δ delay.

We would like to define a notion of *faithfulness* of a compositor, which captures the correspondence between the party setting execution and the emulated setting execution of Π. Roughly, a compositor is called *faithful* is these two settings are identical in the eyes of the honest parties. This faithfulness may be conditioned to work only on certain classes of overlay protocols Π and underlying protocols Y, and may require that a certain subset $\mathcal{H}$ of these Ys are well-behaved.

Definition 3.7 (Compositor Faithfulness (informal))**.** We say that a compositor Λ is *faithful* for an overlay protocol Π, a number of overlay parties $n \in \mathbb{N}$ running over a number of underlying protocols $\mathcal{Y} = (\mathsf{Y}^1, \dots, \mathsf{Y}^\ell)$ if: For all number of compositor parties $m \in \mathbb{N}$, for every compositor index $j \in [m]$, for every overlay index $i \in [n]$ that "corresponds' to "well-behaving" underlying $\mathcal{Y}$s it holds that: The party ${}^j\mathsf{Z}^i$'s emulated execution "is identical" to *some* party setting execution (it cannot tell if it is emulated or not). Within that emulated execution, the emulated instance ${}^j\mathsf{Z}^i$ is given the "same" inputs as *write*(*data*) by its environment as the compositor is given by its own environment as *writeToMachine*(i, *data*) for that same i.

The motivation for the above definition stems from the fact that, if it is known that Π is secure in the party setting, these security results can be translated to the emulated setting. The full definition of faithfulness will state that for all adversaries in the emulated setting, there is a simulator in the party setting that makes the views of honest parties identical. Since the protocol Π is secure in the party setting under that simulator, it must also be secure in the emulated setting under any adversary. This line of argument is not unlike Canetti's UC arguments.

While the fact that the view of the honest party is the same in both the emulated and party setting is sufficient to argue that the *write* instructions in both settings will be the same in the views of the honest parties, this will not be sufficient. The last part of the definition sketch above whereby the *writeToMachine* instructions of the emulated setting are replicated as *write* instructions in the party setting is a necessary ingredient to make the definition useful. Otherwise, trivial constructions in which *writeToMachine* instructions are ignored are possible, yet we want to avoid such pathologies.

To make the above definition precise, we must develop a number of tools. Namely, we must state what "well-behaving" underlying protocols are, and what the "correspondence" between overlay parties and underlying protocols means. Furthermore, we must specify what the "identical" emulated execution means, and what the "same" inputs are, in which definitions we will be required to allow for some slack.

We define the following views for the two execution settings.

Definition 3.8 (Honest View in the Party Setting)**.** Consider a party setting execution $\mathcal{E}'$ of duration R rounds sampled from $\textsc{Run}_n^{\mathsf{PS}}(\Pi, \mathcal{A}, \mathcal{Z}, n, H, \Delta)$. The *party setting view of honest parties* $\textsc{View}_H^{\mathsf{PS}}(\mathcal{E}')$ is the $|H| \times R$ matrix

$$\begin{bmatrix} \Pi_1^{H_1} & \dots & \Pi_R^{H_1} \\ \vdots & \ddots & \vdots \\ \Pi_1^{H_{|H|}} & \dots & \Pi_R^{H_{|H|}} \end{bmatrix},$$

of the transcripts of honest parties where $\Pi_r^{H_i}$ denotes the transcript of party Π^{H_i} (the party with index H_i, the i^{th} honest party) obtained at the end of round r.

Note that, in the above definition, the transcripts concerned pertain to the set H of guaranteed honest party indices only, even though the adversary may choose to leave some of the other parties uncorrupted, too. The transcript of those other parties are not included in $\textsc{View}_H^{\mathsf{PS}}(\mathcal{E}')$.

Note also that, in each row of the above definition, the transcripts are taken for a particular party H_i at increasing rounds, and therefore we will have that $\Pi_r^{H_i} \preceq \Pi_{r+1}^{H_i}$ and, so, the transcripts recorded in each row will be growing in an append-only fashion.

Definition 3.9 (Emulation Consistency)**.** An execution $\mathcal{E}$ with duration R rounds sampled from $\textsc{Run}_m^{\mathsf{ES}}(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$ is (j, H, Δ_v)-consistent, for a compositor index $j \in [m]$, a set of overlay machine indices $H \subseteq [n]$, and reality lag Δ_v, if for all $i \in H$, for all $r \geq 0$, for all $r + \Delta_v < r' < R$, it holds that ${}^j\Lambda.\mathsf{emuSnapshot}(i, r)$ executed *in vitro* at the end of round r' is equal to ${}^j\Lambda.\mathsf{emuSnapshot}(i, r)$ executed *in vitro* at the end of round $r' + 1$.

Note that in the above definition, we allow $r = 0$, even though rounds in the execution begin at 1.

Definition 3.10 (Honest View in the Emulated Setting)**.** Consider an emulated setting execution $\mathcal{E}$ with duration R rounds sampled from $\textsc{Run}_m^{\mathsf{ES}}(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$. If $\mathcal{E}$ is (j, H, Δ_v)-consistent, then the *emulated setting view of honest parties* $\textsc{View}_{j,H,\Delta_v}^{\mathsf{ES}}(\mathcal{E})$, parametrized by an index j, a set of indices H, and a *reality lag* $\Delta_v \in \mathbb{N}$, is the $|H| \times R$ matrix

$$\begin{bmatrix} E(H_1, 1) & \dots & E(H_1, R - \Delta_v - 1) \\ \vdots & \ddots & \vdots \\ E(H_{|H|}, 1) & \dots & E(H_{|H|}, R - \Delta_v - 1) \end{bmatrix},$$

where $E(H_{|H|}, 1)$ denotes the return value of invoking, *in vitro* at the end of round $r + \Delta_v$, the $\mathsf{emuSnapshot}$ functionality of the j^{th} compositor party ${}^j\Lambda$ with parameters the index H_i of the i^{th} overlay machine (among those included in H) and round r.

On the other hand, if the execution $\mathcal{E}$ is *not* (j, H, Δ_v)-consistent, then we let $\textsc{View}_{j,H,\Delta_v}^{\mathsf{ES}}(\mathcal{E}) = \bot$.

Definition 3.11 (Party Setting Externalities)**.** Consider a party setting view V of honest parties and size $|H| \times R$. The *party setting externalities* $\textsc{Extern}^{\mathsf{PS}}(V)$ is the $|H| \times R$ matrix

$$\begin{bmatrix} \mathsf{W}_1^{H_1} & \dots & \mathsf{W}_R^{H_1} \\ \vdots & \ddots & \vdots \\ \mathsf{W}_1^{H_{|H|}} & \dots & \mathsf{W}_R^{H_{|H|}} \end{bmatrix},$$

where $\mathsf{W}_r^{H_i}$ denotes the sequence of messages written into an honest party with index H_i during round r. This sequence of messages can be extracted from the transcript $\Pi_r^{H_i}$ found in V.

Definition 3.12 (Emulated Setting Externalities)**.** Consider an emulated setting execution $\mathcal{E}$ with duration R rounds sampled from $\textsc{Run}_m^{\mathsf{ES}}(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$. Consider the transcript of compositor client ${}^j\Lambda$ in $\mathcal{E}$. Within that transcript, observe the *writeToMachine* calls made by $\mathcal{Z}$ on ${}^j\Lambda$ during some fixed round $1 \leq r \leq R$. Among those, consider the calls to *writeToMachine* that were invoked with first argument some fixed machine index $1 \leq i \leq m$. These *writeToMachine* calls were invoked, in order, as $\mathsf{writeToMachine}(i, \mathsf{data}_1), \dots, \mathsf{writeToMachine}(i, \mathsf{data}_k)$ all during round r. Let $\mathsf{W}_r^i = (\mathsf{data}_1, \dots, \mathsf{data}_k)$ denote the sequence containing all the *data* parameter values of those calls. The *emulated setting externalities* $\textsc{Extern}_{j,H}^{\mathsf{ES}}(\mathcal{E})$, parametrized by an index j and a set of indices H, is the

$|H| \times R$ matrix

$$\begin{bmatrix} \mathsf{W}_1^{H_1} & \dots & \mathsf{W}_R^{H_1} \\ \vdots & \ddots & \vdots \\ \mathsf{W}_1^{H_{|H|}} & \dots & \mathsf{W}_R^{H_{|H|}} \end{bmatrix}.$$

Definition 3.13 (Externality Similarity)**.** Consider the externalities E_1, E_2 (in the party or emulated setting) with dimensions $n \times R_1$ and $n \times R_2$ respectively. We say that E_1 is *similar* to E_2 with *lateness* parameter $\Delta_u \in \mathbb{N}$ and *earliness* parameter $\Delta_v \in \mathbb{N}$, written $E_1 \lessapprox_{\Delta_u,\Delta_v} E_2$, if the following holds: For any message m located within the writebox at position (i, r) of E_1, with $r < R_1 - \Delta_u - \Delta_v$, there exists a round $r - \Delta_v \leq r' \leq r + \Delta_u$ such that the message m appears within the writebox at position (i, r') of E_2.

Definition 3.14 (Externality Similarity in Distribution)**.** Consider the externalities random variables E_1, E_2. We say that E_1 is *similar in distribution* to E_2 with *lateness* parameter $\Delta_u \in \mathbb{N}$ and *earliness* parameter $\Delta_v \in \mathbb{N}$, written $E_1 \overset{d}{\lessapprox}_{\Delta_u,\Delta_v} E_2$, if there exists a sample space Ω and two coupled random variables $\widetilde{E}_1(\omega), \widetilde{E}_2(\omega)$ such that $E_1' \overset{d}{=} E_1$ and $E_2' \overset{d}{=} E_2$ and $E_1 \lessapprox_{\Delta_u,\Delta_v} E_2$.

Definition 3.15 (Belief)**.** A *belief* B is any predicate over an execution $\mathcal{E}$.

For example, given an execution $\mathcal{E}$, with underlying distributed ledger protocols $\mathcal{Y} = (\mathsf{Y}^1, \dots, \mathsf{Y}^\ell)$, we can define a belief B asserting that the majority of underlying ledger protocols are secure.

Definition 3.16 (Belief System)**.** A *belief system* $\mathcal{B}$ is a set of beliefs.

Definition 3.17 (Honesty Correspondence)**.** Given a belief system $\mathcal{B}$ and a number $n \in \mathbb{N}$ of overlay parties, an *honesty correspondence* $\mathcal{H}(B)$ is any function from $\mathcal{B} \longrightarrow 2^{[n]}$.

Definition 3.18 (Belief-Respecting Environment)**.** An environment $\mathcal{Z}$ is *belief-respecting* for a belief B if for all executions $\mathcal{E}$ in the support of a given distribution of executions, it holds that $B(\mathcal{E})$.

Definition 3.19 (Emulation Faithfulness)**.** A compositor Λ is $(\Pi, n, \mathcal{Y}, \mathcal{B}, \mathcal{H}, \Delta_u, \Delta_v)$*-emulation-faithful* for an overlay protocol Π, a number of overlay parties $n \in \mathbb{N}$, a sequence of underlying protocols $\mathcal{Y} = (\mathsf{Y}^1, \dots, \mathsf{Y}^\ell)$, a belief system $\mathcal{B}$, honesty correspondence $\mathcal{H} : \mathcal{B} \longrightarrow 2^{[n]}$, lateness $\Delta_u \in \mathbb{N}$, and reality lag $\Delta_v \in \mathbb{N}$ if:

For all beliefs $B \in \mathcal{B}$, for all PPT adversaries $\mathcal{A}$ and all B-respecting PPT environments $\mathcal{Z}$, for all number of compositor parties $m \in \mathbb{N}$, for all compositor party indices $j \in [m]$, there is a PPT simulator $\mathcal{S}$ and there is a PPT environment $\mathcal{Z}'$ such that the following holds:

1) $\text{VIEW}^{\mathsf{ES}}_{j,\mathcal{H}(B),\Delta_v}(\mathcal{E}) \overset{d}{=} \text{VIEW}^{\mathsf{PS}}_{\mathcal{H}(B)}(\mathcal{E}')$
2) $\text{EXTERN}^{\mathsf{ES}}_{j,\mathcal{H}(B)}(\mathcal{E}) \overset{d}{\lessapprox}_{\Delta_u,\Delta_v} \text{EXTERN}^{\mathsf{PS}}(\text{VIEW}^{\mathsf{PS}}_{\mathcal{H}(B)}(\mathcal{E}'))$

where execution $\mathcal{E}$ is sampled from $\text{RUN}^{\mathsf{ES}}_m(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$ and execution $\mathcal{E}'$ is sampled from $\text{RUN}^{\mathsf{PS}}(\Pi, \mathcal{S}, \mathcal{Z}', n, \mathcal{H}(B), \Delta)$, and $\Delta = 2\Delta_v + \Delta_u$.

Definition 3.20 (Replication Faithfulness)**.** A compositor Λ is $(\Pi, n, \mathcal{Y}, \mathcal{B}, \mathcal{H}, \Delta_u, \Delta_v)$*-replication-faithful* for overlay protocol Π, number of overlay parties $n \in \mathbb{N}$, a sequence of underlying protocols $\mathcal{Y} = (\mathsf{Y}^1, \dots, \mathsf{Y}^\ell)$, belief system $\mathcal{B}$, honesty correspondence $\mathcal{H} : \mathcal{B} \longrightarrow 2^{[n]}$, lateness $\Delta_u \in \mathbb{N}$, and reality lag $\Delta_v \in \mathbb{N}$ if:

For all beliefs $B \in \mathcal{B}$, all all PPT adversaries $\mathcal{A}$, for all B-respecting PPT environments $\mathcal{Z}$, for all number of compositor parties $m \in \mathbb{N}$, for all compositor party indices $j, j' \in [m]$ and for all overlay party indices $i \in \mathcal{H}(B)$, for all rounds $1 \leq r < R - \max(\Delta_v, \Delta_u)$, and all compositor party indexes $j, j' \in [m]$ it holds that ${}^j\text{SIM}_r = {}^{j'}\text{SIM}_r$, where ${}^j\text{SIM}_r$ (resp. ${}^{j'}\text{SIM}_r$) indicates the result of calling $\mathsf{emuSnapshot}$ of $\Lambda[j]$ (resp. $\Lambda[j']$) with inputs (i, r) *in vitro* at the end of round $r + \Delta_v$ of $\mathcal{E}$, where $\mathcal{E}$ is an execution sampled from $\text{RUN}^{\mathsf{ES}}_m(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$.

4. Construction

Before diving into the details of the pseudocode, we give an intuitive overview of the ROLLERBLADE construction.

There are m ROLLERBLADE clients numbered $1, 2, \dots, j, \dots, m$ and n underlying protocols $\mathsf{Y}^1, \mathsf{Y}^2, \dots, \mathsf{Y}^i, \dots, \mathsf{Y}^n$. Each ${}^j\mathsf{RB}$ of the clients runs a full node for each of the n underlying ledger protocols, ${}^j\mathsf{Y}^1, {}^j\mathsf{Y}^2, \dots, {}^j\mathsf{Y}^i, \dots, {}^j\mathsf{Y}^n$. Each Y^i of these underlying protocols promises (but may not deliver on that promise) to be safe, live with liveness u_i, and timely with timeliness v_i. Note the technical difference between Y^i, denoting the i^{th} underlying protocol (an ITM), and ${}^j\mathsf{Y}^i$, denoting an instance of the i^{th} underlying protocol running on the j^{th} client (an ITI).

Each ${}^j\mathsf{RB}$ *emulates* within its implementation n different instances of the overlay protocol Π, the instances ${}^j\mathsf{Z}^1, {}^j\mathsf{Z}^2, \dots, {}^j\mathsf{Z}^i, \dots, {}^j\mathsf{Z}^n$, one for every underlying ledger protocol, ${}^j\mathsf{Y}^1, {}^j\mathsf{Y}^2, \dots, {}^j\mathsf{Y}^i, \dots, {}^j\mathsf{Y}^n$.

Our rollerblade construction works assuming that the underlying and overlay protocols satisfy certain requirements which we now specify.

Definition 4.1 (ROLLERBLADE-Suitable Underlying Protocol)**.** A *distributed protocol* is called a ROLLERBLADE-*Suitable Underlying Protocol* if it is a (potentially permissionless) distributed temporal ledger protocol which is a bulletin board and a certifiable protocol.

In summary, the underlying ledgers provide the following functionalities: (1) **construct**: Initializes the protocol; (2) **execute**: Executes one round of the protocol; (3) **write**: Writes a transaction to the ledger. The transaction appears within u rounds if the ledger protocol is *live*; (4) **read**: Reads the temporal ledger. The ledger read is consistent with whatever other honest parties are reading if the ledger protocol is *safe*, and the transactions appear with correct recorded rounds if the protocol is *timely*; (5) **encode**: Given an arbitrary string, produces a valid bulletin transaction; (6) **decode**: Given a bulletin transaction, produces the original string used to encode it; (7) **transcribe**: Produces a transcription τ of the current temporal ledger; (8) **untranscribe**: Given a transcription τ, produces a ledger promised to be *safe* and *live* as compared to the rest of the honest parties.

Our construction allows running *any* deterministic overlay permissioned distributed protocol working over authenticated channels.

Definition 4.2 (ROLLERBLADE-Suitable Overlay Protocol)**.** A distributed protocol is called ROLLERBLADE-*Suitable Overlay Protocol* if it is deterministic and permissioned.

The ROLLERBLADE client ${}^{j}\mathsf{RB}$ allows the external user to issue a *write* instruction with some data to any i^{th} emulated machine ${}^{j}\mathsf{Z}^{i}$. This is done as follows. When the user of ${}^{j}\mathsf{RB}$ issues a *write* instruction to ${}^{j}\mathsf{Z}^{i}$, this instruction is not given to ${}^{j}\mathsf{Z}^{i}$ directly. Instead, it is serialized into a string and *encoded* into a transaction to be recorded on the respective underlying ledger ${}^{j}\mathsf{Y}^{i}$. This is captured by the $\mathsf{writeToMachine}$ function illustrated in Algorithm 1. The encoding functionality is made available because the underlying ledgers are assumed to be bulletin boards. When the 'write' instruction becomes recorded in the ledger ${}^{j}\boldsymbol{L}^{i}$ reported by ${}^{j}\mathsf{Y}^{i}$, then it will be passed to the emulated machine ${}^{j}\mathsf{Z}^{i}$ as soon as the respective round emulation takes place. Through this recording of machine inputs on-ledger, we intent for other rollerblade clients ${}^{j'}\mathsf{RB}$ to replicate the exact emulation of ${}^{j'}\mathsf{Z}^{i}$ for that same i (in the Analysis section, this is made precise in Theorem A.1). Additionally, the external user can, at any time, issue a *read* instruction to the emulated machine ${}^{j}\mathsf{Z}^{i}$, without affecting its current state (recall that we require Π's *read* method to not alter the machine's internal state upon completion).

Algorithm 1 The *writeToMachine* function made available to the external user by a ROLLERBLADE party.

```
1: function WRITETOMACHINE($i$, data)
2:     instr ← {sid:this.sid, type:'write', data:data}
3:     $s$ ← serialize(instr)
4:     tx ← this.${}^{j}\mathsf{Y}^{i}$.encode($s$)
5:     this.${}^{j}\mathsf{Y}^{i}$.write(tx)
6: end function
```

Each of ${}^{j}\mathsf{Z}^{i}$ is emulated in a per-round basis by having its *execute* function invoked once per round r. When it is emulated, it expects its *write* function to have been called some (0 or more) number of times prior to *execute* being invoked, indicating user input for round r. When *execute* is invoked, it has access to read and write into the network through the authenticated channels interface NET. When it *reads* from the network, it consumes network messages that other machines have dispersed into the network during previous rounds (potentially with some delay Δ).

In order to emulate this communication between machines, the system works as follows. Every rollerblade client is tasked with *checkpointing* every ledger to every other ledger during every round. Checkpointing is performed as follows. The rollerblade client runs a full node in each of the underlying protocols Y^{i} and invokes the *transcribe* function ${}^{j}\mathsf{Y}^{i}.\mathsf{transcribe}()$ to obtain a transcription τ; this functionality is made available because underlying ledgers are assumed to be transcribable. This transcription is then checkpointed into every other ledger protocol ${}^{j}\mathsf{Y}^{i'}$ by *encoding* it into a bulletin transaction $\mathsf{tx} = {}^{j}\mathsf{Y}^{i'}.\mathsf{encode}(\tau)$ and *writing* it into ${}^{j}\mathsf{Y}^{i'}$. The relayer functionality of the rollerblade is illustrated in Algorithm 2.

Algorithm 2 The ROLLERBLADE *relay* function, executed on every round over the ${}^{j}\mathsf{Y}^{1}, \ldots, {}^{j}\mathsf{Y}^{n}$ underlying protocol instances.

```
1: function RELAY( )                          ▷ Executed on every round
2:     for $i$ ← 1 to this.$n$ do                        ▷ Source
3:         $\tau$ ← this.${}^{j}\mathsf{Y}^{i}$.transcribe()              ▷ Checkpoint
4:         for $i'$ ← 1 to this.$n$ do                     ▷ Target
5:             if $i = i'$ then
6:                 continue
7:             end if
8:             instr ← {
9:                 type:'chkpt',
10:                data:{from:$i'$, cert:$\tau$}
11:            }
12:            enc ← this.${}^{j}\mathsf{Y}^{i}$.encode(serialize(instr))
13:            this.${}^{j}\mathsf{Y}^{i}$.write(enc)
14:        end for
15:    end for
16: end function
```

It is not necessary for each rollerblade to run this relayer functionality. Instead, separate untrusted but helpful *relayers* can be tasked with checkpointing one chain to another. There is no trust placed upon such a relayer, but at least one relayer must be honest for the protocol to be secure. Therefore, a ROLLERBLADE client can run its own relayer if it wishes, so this does not introduce any additional trust assumptions (this is what we do in our construction above).

These "write" and "chkpt" instructions are the only thing ever written to the ledger from the honest rollerblade clients. In particular, no network outputs produced by any of the ${}^{j}\mathsf{Z}^{i}$ are ever written on the ledger. Instead, we observe that reading the checkpoints is sufficient for each rollerblade client to reproduce the network outputs of all emulated machines.

Let us now explore, in more detail, how this emulation works. Each rollerblade execution of a particular overlay protocol Π on top of a certain number of underlying ledger protocols Y is marked with a *session id* sid in order to achieve context separation with other rollerblades running, potentially different, protocols Π' on top of a, potentially different but not disjoint, set of underlying ledger protocols Y'. This sid is placed into every 'write' instruction recorded on-ledger. Note that 'chkpt' instructions do not need a session id, as checkpoints can be shared across different rollerblade sessions. This means that relayers can be the same across *all* rollerblade sessions.

When a series of 'write' and 'checkpoint' instructions have been recorded as bulletin transactions onto an underlying ledger, a time will come for them to be re-read in order to fuel the emulation. The responsibility of re-reading the instructions from the ledger is bestowed upon the function decodeUnderlying illustrated in Algorithm 3. The function is given an underlying ledger L, which may contain relevant and irrelevant transactions. Its task is to filter out the irrelevant transactions and decode the relevant ones. It returns a sequence of (r, instr) pairs, where r is the round number with which the instruction instr is recorded. To do this, it fills the sequence ret to be returned with the relevant instructions. It iterates over all transactions tx in the ledger L (Line 3). For each such transaction, it attempts to *decode* it (Line 6). If the transaction is a bulletin transaction, the decoding will succeed and return the string that was encoded in it (otherwise, the transaction is ignored in Line 10). Once decoded into a string, the string is deserialized into a dictionary containing the instruction information. This information is checked for relevance in Line 12. In particular, if the instruction is a 'write' instruction, we ensure that its sid matches the sid of the rollerblade in question. As 'chkpt' instructions are not tied to any particular rollerblade, they are always considered relevant.

Algorithm 3 The decodeUnderlying function ran by party ${}^j\mathsf{RB}$ at round r on an underlying ledger $L = {}^j\mathsf{Y}^i.\mathsf{read}()$ sanitizes the underlying ledger by decoding and deserializing each of its transactions into a sequence of messages. It is used by the prepEmuInputs function.

```
1: function DECODEUNDERLYING($i, L$)
2:     ret ← []
3:     for $(r, \mathsf{tx}) \in L$ do
4:         try
5:             ▷ Bulletin board decoding
6:             $s \leftarrow \mathsf{this}.{}^j\mathsf{Y}^i.\mathsf{decode}(\mathsf{tx})$
7:             instr ← deserialize($s$)
8:         catch
9:             ▷ Not bulletin tx, or invalid serialization
10:            continue
11:        end try
12:        if instr.type = 'write' ∧ instr.sid ≠ sid then
13:            continue
14:        end if
15:        ret.append(($r$, instr))
16:    end for
17:    return ret
18: end function
```

The emulation only runs on-demand when the user of ${}^j\mathsf{RB}$ issues a *read* instruction to the machine ${}^j\mathsf{Z}^i$. The user can indicate that the read instruction pertains to a particular round emuRound, and the *read* instruction is conveyed to the snapshot of ${}^j\mathsf{Z}^i$ immediately after round emuRound has been executed.

In order for the *read* functionality to be invoked, the user must first invoke the emuSnapshot functionality of ${}^j\mathsf{RB}$ with parameter i. This function will return a snapshot of the machine ${}^j\mathsf{Z}^i$ at the requested round, and, subsequently, the user can invoke the read functionality on the returned snapshot *in vitro*. The emuSnapshot functionality is illustrated in Algorithm 4. The emuSnapshot function takes the identity i of the machine to emulate and the round for which the emulation is requested emuRound. The emuSnapshot function emulates the execution of the machine ${}^j\mathsf{Z}^i$ using the data obtained by *reading* the ledger ${}^j\boldsymbol{L}^i$ reported by ${}^j\mathsf{Y}^i$ at the current round. Upon reading the ledger, the emuSnapshot function invokes the emulate function to obtain an instance of the ${}^j\mathsf{Z}^i$ machine executed up to round emuRound (inclusive). The emuSnapshot function subsequently returns this instance.

Algorithm 4 The emuSnapshot function made available to the external user by a ROLLERBLADE party.

```
1: function EMUSNAPSHOT($i$, emuRound)
2:     ${}^j\boldsymbol{L}^i \leftarrow \mathsf{this}.{}^j\mathsf{Y}^i.\mathsf{read}()$
3:     $Z \leftarrow \mathsf{this}.\mathsf{emulate}(i, {}^j\boldsymbol{L}^i, \mathsf{emuRound}).Z$
4:     return $Z$
5: end function
```

The core emulation is implemented in the function emulate illustrated in Algorithm 5, which takes parameters with the same semantics as the emuSnapshot function parameters. The job of the emulate function is to emulate the execution of machine ${}^j\mathsf{Z}^i$ up to, and including, round emuRound. This is all done within the mind of machine ${}^j\mathsf{RB}$, with no external communication at all. Upon completing the emulation, the emulate function returns the instance of the machine ${}^j\mathsf{Z}^i$, which can be used to apply read on it by the user.

Additionally, the emulate function returns all the network outputs that ${}^j\mathsf{Z}^i$ produced during the emulation, which we call its *outboxes* (Line 29). The outboxes (initialized in Line 4) are structured as an array containing one *outbox* per index, each index corresponding to a round of execution. In particular $\mathsf{outboxes}[r]$ contains the outbox produced by emulated party ${}^j\mathsf{Z}^i$ during round r. This outbox $\mathsf{outboxes}[r]$, produced during round r, is a list of authenticated messages netouts. Each such authenticated message netout in netouts is a dictionary containing two entries: The *to* entry is a number between 1 and n indicating the recipient of the message; the *msg* entry is the string of the message to be delivered. Since rounds begin at 1, the entry $\mathsf{outboxes}[0]$ is the empty outbox (ensured in its initialization in Line 4).

To conduct the emulation, the emulate function initially calls prepEmuInputs (Line 2), which prepares two arrays *writeboxes* and *inboxes* to be used by the emulation of ${}^j\mathsf{Z}^i$. These sequences are produced by reading the ledger ${}^j\boldsymbol{L}^i$. *The whole emulation of ${}^j\mathsf{Z}^i$ is based only on the data recorded on its respective ledger.* The array *writeboxes* is structured as an array containing one *writebox* per index, each index corresponding to a round of execution. In particular, $\mathsf{writeboxes}[r]$ contains the writebox to be used prior to the execution of round r. The writebox $\mathsf{writeboxes}[r]$ is a list of strings each of which must be

Algorithm 5 The *emulate* function ran by a ROLLERBLADE party executing the overlay protocol for party i at a particular *emulation round* using ledger ${}^j\boldsymbol{L}^i$. The emulation returns the outbox (messages "sent" to other parties) and the instance ${}^j\mathsf{Z}^i$ of the emulated machine as reported by overlay party i.

```
1:  function EMULATE($i, {}^j\boldsymbol{L}^i$, emuRound)
2:      in ← this.prepEmuInputs($i, {}^j\boldsymbol{L}^i$, emuRound)
3:      (writeboxes, inboxes) ← in
4:      outboxes ← [[]]
5:      inboxThisRound ← []
6:      outboxThisRound ← []
7:      function SEND(recp, msg)
8:          outboxThisRound.append({to:recp, msg:msg})
9:      end function
10:     function RECV()
11:         return inboxThisRound
12:     end function
13:     ▷ Oracles passed to the overlay protocol
14:     NET ← (send, receive)
15:     ${}^j\mathsf{Z}^i$ ← new $\Pi^{\text{NET}}$(this.$\Delta, j, n$)
16:     for $r$ ← 1 to emuRound do
17:         roundReceives ← []
18:         outboxThisRound ← []
19:         for data ∈ writeboxes[$r - 1$] do
20:             ▷ Submit writes recorded at $r - 1$
21:             ${}^j\mathsf{Z}^i$.write(data)
22:         end for
23:         ▷ Prepare network msgs to be delivered at $r$
24:         inboxThisRound ← inboxes[$r - 1$]
25:         ▷ Run a single round $r$ of overlay machine
26:         ${}^j\mathsf{Z}^i$.execute()
27:         outboxes.append(outboxThisRound)
28:     end for
29:     return {outboxes:outboxes, $Z$:${}^j\mathsf{Z}^i$}
30: end function
```

written using the *write* functionality of ${}^j\mathsf{Z}^i$ before *execute* is invoked for round r. These writes correspond to (honestly or adversarially) recorded 'write' instructions on the ledger ${}^j\boldsymbol{L}^i$. The data structure inboxes is structured similarly. The inbox inboxes$[r]$ contains the inbox to be used *during* the execution of ${}^j\mathsf{Z}^i$ at round r. The inbox inboxes$[r]$ is a list of authenticated messages netins. Each such authenticated message netin in netins is a dictionary containing two entries: The *from* entry is a number between 1 and n indicating the sender of the message; the *msg* entry is the string of the message to be delivered.

The emulation begins by initializing a new machine ${}^j\mathsf{Z}^i$ from scratch (Line 15). The emulation proceeds in rounds managed by the main loop in Line 16. For each iteration of this *for* loop, the *execute* function of the emulated machine is invoked, once per every round r. In preparation of the execution for round r, we have to invoke the *write* function of ${}^j\mathsf{Z}^i$ for every write in the writebox pertaining to round r. This is performed in the *for* loop of Line 19, which invokes ${}^j\mathsf{Z}^i$'s *write* method for each *write* in the current round's writebox.

During initialization (Line 15), the emulated machine is initialized by giving it access to the network oracle NET. Whereas, normally, the protocol Π would have expected this oracle to realize an authenticated channels functionality, we *trap* the calls to the network *send* and network *receive* functions that the instance ${}^j\mathsf{Z}^i$ of Π may call and we replace them with our own implementation based on the underlying ledgers. The trapped functions are implemented in Line 7 and Line 10 respectively. The machine ${}^j\mathsf{Z}^i$ may invoke the network *send* method, during its *execute* invocation, zero or more times during the round. When the network *send* method is invoked by ${}^j\mathsf{Z}^i$, the message is *not* conveyed to other machines. Instead, it is appended to the array outboxThisRound. The variable contains the append-only outbox produced by the emulation during round r. It is initialized as the empty array (Line 18) at the beginning of every round. Upon the completion of a round's execution, the particular outbox is appended to the list of all outboxes (Line 27). Lastly, in preparation for the execution of round r, the variable inboxThisRound is initialized to inboxes$[r-1]$ (Line 24) and is an inbox containing the network messages to be delivered at round r (and may have been disbursed during rounds $r - \Delta, \ldots, r - 1$). This inbox is returned whenever the emulated machine calls the network *receive* function of the oracle NET. The variable inboxThisRound is reassigned at the beginning of every round.

Once the last iteration of the *for* loop of Line 16 completes, the emulated machine ${}^j\mathsf{Z}^i$ has concluded its execution of rounds $1, \ldots,$ emuRound. At the end of the emulation, the emulate function returns the outboxes generated by ${}^j\mathsf{Z}^i$ throughout its execution as well as the machine instance ${}^j\mathsf{Z}^i$ itself. The machine instance can be used for example to call *read* on it.

The last piece of the puzzle is the prepEmuInputs function which is called by emulate and illustrated in Algorithm 6. This function receives as input the ledger ${}^j\boldsymbol{L}^i$ and returns the writeboxes and inboxes to be used by emulate to emulate the machine ${}^j\mathsf{Z}^i$. It will be important for our security results that the output of prepEmuInputs is deterministic and depends only on the ledger ${}^j\boldsymbol{L}^i$ and not on any of the other ledgers ${}^j\boldsymbol{L}^{i'}, i' \neq i$.

In particular, the function prepEmuInputs receives as parameters the underlying index i, the ledger ${}^j\boldsymbol{L}^i$, and the emulation round emuRound. The method begins by initializing inboxes and writeboxes. For every round $r = 0 \ldots$ emuRound we set inboxes$[r]$ and writeboxes$[r]$ to be empty lists initially (Line 6). The entries inboxes$[0]$ and writeboxes$[0]$ will remain empty since the emulation begins at round 1 with an empty inbox and writebox.

The rest of the indices in inboxes and writeboxes we fill in by reading the relevant rollerblade transactions within ${}^j\boldsymbol{L}^i$. Initially, the relevant transactions are extracted from the ledger ${}^j\boldsymbol{L}^i$ into a sequence of instructions L by invoking decodeUnderlying (Line 15). The sequence L contains pairs $(r,$ instr$)$ of instructions, each accompanied by the round with which the relevant instruction was recorded on

Algorithm 6 The *prepare emulation inputs* function ran by ROLLERBLADE party j to prepare the necessary inputs for the emulation of overlay party i at round emuRound based on the underlying ledger ${}^j\boldsymbol{L}^i$. The function returns the inputs, which consist of *writes* (user inputs) and *inboxes* (network inputs produced as network output when recursively simulating all other parties ${}^j\mathsf{Y}^{i'}$) arranged by round.

```
function PREPEMUINPUTS($i, {}^j\boldsymbol{L}^i$, emuRound)
    if emuRound $>$ this.now $- v_i$ then
        return $\perp$        ▷ Emu round is too far in future
    end if
    inboxes $\leftarrow$ [[ ]]; writeboxes $\leftarrow$ [[ ]]
    for $r \leftarrow 1$ to emuRound do
        inboxes.append([ ])
        writeboxes.append([ ])
    end for
    seenLen $\leftarrow$ [ ]
    for $i' \leftarrow 1$ to $n$ do
        seenLen.append(0)
        $F^{i'} \leftarrow$ [ ]
    end for
    $L \leftarrow$ this.decodeUnderlying($i, {}^j\boldsymbol{L}^i$)
    for ($r$, instr) $\in L$ do
        if $r \geq$ emuRound then
            break
        end if
        if instr.type = 'write' then
            writeboxes[$r$].append(instr.data)
        end if
        if instr.type = 'chkpt' then
            $i' \leftarrow$ instr.from
            $\tau \leftarrow$ instr.data.cert
            ${}^j\mathfrak{L}^{i'} \leftarrow$ this.${}^j\mathsf{Y}^{i'}$.untranscribe($\tau$)
            $F^{i'} \leftarrow F^{i'} \parallel ({}^j\mathfrak{L}^{i'}[|F^{i'}|:])$
            res $\leftarrow$ this.emulate($i', F^{i'}, r - u_i - v_{i'} - 1$)
            outboxes $\leftarrow$ res.outboxes
            netins $\leftarrow$ this.outToIn(outboxes, $i', i$)
            for netin $\in$ netins[seenLen[$i'$]:] do
                inboxes[$r$].append(netin)
            end for
            seenLen[$i'$] $\leftarrow$ seenLen[$i'$] + |newNetins|
        end if
    end for
    return (writeboxes, inboxes)
end function
```

the ledger ${}^j\boldsymbol{L}^i$. This is where we will use the temporal nature of the underlying ledger ${}^j\boldsymbol{L}^i$: The round r recorded on-ledger for the instruction instr is the round of the emulation of ${}^j\mathsf{Z}^i$ during which we will utilize this instruction.

The instructions in L are processed sequentially by the *for* loop in Line 16. As we are only interested in the data necessary to emulate up to round emuRound, we can conclude this loop early if we see a transaction recorded with round $r \geq$ emuRound (Line 17). The easy case is when the instruction type is a 'write'. In that case, we extend the writebox writeboxes[r] for round r by appending the data instr.data to be written to it (Line 21).

The more complicated case pertains to instruction type 'chkpt'. This is a checkpoint from underlying ledger i' to underlying ledger i, i.e., concerns a certificate of ledger i' that was recorded onto ledger i. This certificate's data instr.data.cert is fully recorded within ${}^j\boldsymbol{L}^i$ and does not require reading from ${}^j\mathsf{Y}^{i'}$. The function untranscribe of ${}^j\mathsf{Y}^{i'}$ is used to untranscribe this certificate into a ledger ${}^j\mathfrak{L}^{i'}$ (Line 26), but the return value of this untranscribe function does not at all depend on the local state of ${}^j\mathsf{Y}^{i'}$, but only on the parameter instr.data.cert passed to it. This "imaginary ledger", ${}^j\mathfrak{L}^{i'}$ extracted from untranscribing, is the ledger with index i' extracted based on the data included within ${}^j\boldsymbol{L}^i$. It is possible that ${}^j\mathfrak{L}^{i'} \neq {}^j\mathsf{Y}^{i'}$.read() (especially if ledger i' is unsafe).

This ledger ${}^j\mathfrak{L}^{i'}$, obtained by untranscribing, is then passed into a *new* emulate invocation for machine ${}^j\mathsf{Z}^{i'}$ (Line 28). Here, note that emulate and prepEmuInputs are mutually recursive functions. This recursive emulation is performed based on ledger ${}^j\mathfrak{L}^{i'}$ and not on ${}^j\mathsf{Y}^{i'}$.read(). The emulate function is executed up to emulation round $r - u_i - v_{i'} - 1$. The reason for this arithmetic will become apparent in the security proof, but, for now, suffice to say that, during round r, the emulation ${}^j\mathsf{Z}^i$ will see received messages originating from i' emitted during rounds up to $r - u_i - v_{i'} - 1$, and messages emitted during later rounds will be received later. The *outboxes* returned from this emulation of ledger i', within the confines of emulate executed upon ledger i, are then collected into the outboxes variable, which contains one outbox per round. These outboxes correspond to messages sent by i' to i on the emulated network. Every message received by i and originating from i' is collected into the array netins in Line 30 by invoking the function outToIn. This netins contains *all* the messages sent from i' to i throughout the execution of i' up to round $r - u_i - v_{i'} - 1$. Note that, if i' is a safe ledger, then in every two iterations of the loop of Line 16, the values of the variable netins will be prefixes of one another. As netins grows, some of the messages within it have already been received by ${}^j\mathsf{Z}^i$ during previous rounds, whereas some messages are new and have not been processed yet. We wish to capture those messages that have appeared anew. Towards this purpose, we maintain a count seenLen[i'] of incoming messages originating from i' that have already been processed by ${}^j\mathsf{Z}^i$ during previous rounds. The newly arriving messages that still need to be processed are netins[seenLen[i']:]. All of these newly arriving messages are placed in inboxes[r] and, as such, are scheduled to be received by the emulation of ${}^j\mathsf{Z}^i$ for round $r + 1$. The variable seenLen[i'] is then updated (Line 34) to record the count of messages that have already been processed by i' in previous rounds, so that the messages are not processed again in the future.

Lastly, let us describe the outToIn function, which translates the outboxes of party i' to an inbox for party i. The function is called by prepEmuInputs and is illustrated in

Algorithm 7 The outboxes-to-inbox translation algorithm ran by party ${}^{j}\mathsf{RB}$. The algorithm rewrites messages placed in an outbox of one overlay party i' and translates them into messages in an inbox which is to be consumed by a different overlay party i. Only the relevant messages are reported by filtering out the messages that have a to field that does not match i.

```
function OUTTOIN(outboxes, $i'$, $i$)
    inbox $\leftarrow$ [ ]
    for outbox $\in$ outboxes do
        for netout $\in$ outbox do
            if netout.to $= i$ then
                inbox.append({from: $i'$,
                              msg: netout.msg})
            end if
        end for
    end for
    return inbox
end function
```

Algorithm 7. The function takes an $\mathsf{outboxes}$ argument containing one outbox per round, as well as the sending party index i' and the receiving party index i. The function returns an inbox for party i. To do this, the function loops through each outbox in $\mathsf{outboxes}$, and iterates over each netin within the outbox. For each such netout, the function checks that i is the recipient of the message (Line 5) and ignores the message if not. Otherwise, the message is translated from a netout to a netin by setting its *from* field to be the sender i' and is appended to the inbox (Line 7). This completes the construction. The full construction is illustrated in Algorithm 8.

Algorithm 8 The ROLLERBLADE compositor.

```
protocol ROLLERBLADE
    public function emuSnapshot ($i$, emuRound)
    public function writeToMachine ($i$, data)
    function prepEmuInputs ($i$, ${}^{j}\boldsymbol{L}^{i}$, emuRound)
    function emulate ($i$, ${}^{j}\boldsymbol{L}^{i}$, emuRound)
    function outToIn (outboxes, $i'$, $i$)
    function relay ()
    function decodeUnderlying ($i$, $L$)
    function construct(${}^{j}\mathsf{Y}$, $\Pi$)
        this.${}^{j}\mathsf{Y} \leftarrow {}^{j}\mathsf{Y}$
        this.$\Pi \leftarrow \Pi$
        this.$n \leftarrow |{}^{j}\mathsf{Y}|$          ▷ Number of overlay parties
        this.$\Delta_u \leftarrow \max\{$this.${}^{j}\mathsf{Y}^{i}.u\}_{i\in[n]}$
        this.$\Delta_v \leftarrow \max\{$this.${}^{j}\mathsf{Y}^{i}.v\}_{i\in[n]}$
        this.$\Delta \leftarrow 2\Delta_v + \Delta_u$
        this.now $\leftarrow 0$             ▷ Current round number
    end function
    function execute( )
        this.now $\leftarrow$ this.now $+ 1$
        this.relay()
    end function
end protocol
```

5. Conclusion

Summary. We have developed a generic construction that allows running a replicated emulation of (virtually) any distributed protocol using underlying ledgers as the communication mechanism. During the process of proving our construction secure, we developed a technical framework to discuss the analogy between the party setting and the emulated setting, and proved that the two settings are equivalent. Our main security result took the form of a simulation-based argument in two main theorems: The Emulation Theorem (showing that, within the view of a single compositor, the party setting and the emulated setting are equivalent), and the Replication Theorem (showing that different compositors share the same view within the same execution). Our construction can be used for various applications, for example to run a consensus protocol on top of existing distributed ledgers, giving rise to a layer 2 rollup that provides better security guarantees than any of the underlying ledger protocols alone.

Related work. Building reliable systems out of unreliable components is a classical problem [7], [8]. In consensus, Lamport's Byzantine Fault Tolerance problem [9] aims to solve a reliability problem, where different processors disagree about their outcomes. The composition of multiple *blockchain* protocols was explored by *Fitzi et al.* [10], but for the purpose of performance in terms of *latency*, not reliability. In their paper, they also introduce the notion of *relative persistence*, in which they talk of *dynamic ledgers* (*cf.*, our *temporal ledgers*) and transaction *ranks* (*cf.*, our *recorded rounds*) which is related, but not equivalent, to our notion of *timeliness*. They also define the notion of a *blockchain combiner* (*cf.*, our *compositors*). Their protocol is *passive* (*cf.* TrustBoost [1]), meaning the "combiner" does not achieve full consensus ([10, Section 5]). Their "combiner" is an instance of our *compositors*, which further allow for generic distributed protocols to run in an emulated and replicated fashion, and achieve full consensus.

The idea of borrowing security has been explored in *merged mining* [11], *merged staking* [12], and *checkpointing* [13], The idea of composing ledgers to achieve a more reliable overlay ledger was first proposed in a short Cosmos GitHub issue called *recursive Tendermint* [14]. This concept was expanded upon by TrustBoost [1] where they build a composition using Cosmos as the underlying construction, IBC for cross-chain communication, and Information Theoretic HotStuff as the overlay protocol. They conjecture that their construction is secure. However, they stop short of proving the security of their construction. Their security theorem in the so-called "active mode" ([1, Theorem 2]) states the variable m in the theorem statement. That variable is interpreted in the *party setting* in the *proof*, but in the *emulated setting* in the rest of the paper. Therefore, the theorem's positive or negative statement for m interpreted as ledgers (and not parties) is not proven. The correspondence between the party setting and the emulated setting is only conjectured in the short paragraph "Security guarantee" [1, Section 4.1].

Proving this correspondence requires significant technical work and the framework which we develop in this paper. In the present work, we answer the question TrustBoost left open affirmative and calculate the correct parametrization to instantiate their system securely (e.g., the calculation of Δ in Theorem A.2). We note that any secure deployment of TrustBoost must include a correct choice of the parameter Δ, whose calculation is missing from their paper.

References

[1] X. Wang, P. Sheng, S. Kannan, K. Nayak, and P. Viswanath, "TrustBoost: Boosting Trust among Interoperable Blockchains," *arXiv preprint arXiv:2210.11571*, 2022.

[2] J. Neu and D. Zindros, "Information Theoretic Streamlet," Jul 2023, Unpublished manuscript.

[3] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001, pp. 136–145.

[4] B. Y. Chan and E. Shi, "Streamlet: Textbook streamlined blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 1–11.

[5] M. Al-Bassam, "LazyLedger: A distributed data availability ledger with client-side smart contracts," *arXiv preprint arXiv:1905.09274*, 2019.

[6] E. N. Tas, D. Zindros, L. Yang, and D. Tse, "Light Clients for Lazy Blockchains," *arXiv preprint arXiv:2203.15968*, 2022.

[7] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[8] E. F. Moore and C. E. Shannon, "Reliable circuits using less reliable relays," *Journal of the Franklin Institute*, vol. 262, no. 3, pp. 191–208, 1956.

[9] R. Shostak, M. Pease, and L. Lamport, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[10] M. Fitzi, P. Gaži, A. Kiayias, and A. Russell, "Ledger combiners for fast settlement," in *Theory of Cryptography Conference*. Springer, 2020, pp. 322–352.

[11] V. Durham. (2011, Apr) Namecoin. Available at: https://namecoin.org/. [Online]. Available: https://namecoin.org/

[12] A. Kiayias, P. Gaži, and D. Zindros, "Proof-of-stake sidechains," in *IEEE Symposium on Security and Privacy*, IEEE. IEEE, 2019.

[13] D. Karakostas and A. Kiayias, "Securing proof-of-work ledgers via checkpointing," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2021, pp. 1–5.

[14] mossid. (2019, Sep) Recursive Tendermint. Available at: https://github.com/cosmos/ibc/issues/547. [Online]. Available: https://github.com/cosmos/ibc/issues/547

[15] A. Lewis-Pye and T. Roughgarden, "Permissionless Consensus," *arXiv preprint arXiv:2304.14701*, 2023.

[16] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[17] J. Yun, C. Goes, and A. Sripal. (2019, Feb) ICS-002: Client Semantics. Available at: https://github.com/cosmos/ibc/tree/main/spec/core/ics-002-client-semantics. [Online]. Available: https://github.com/cosmos/ibc/tree/main/spec/core/ics-002-client-semantics

[18] E. N. Tas, D. Tse, and Y. Wang, "A Circuit Approach to Constructing Blockchains on Blockchains," *arXiv preprint arXiv:2402.00220*, 2024.

Appendix

1. Discussion

Certificates VS transcriptions. Our ROLLERBLADE construction was given assuming the underlying protocols provide a $\mathsf{transcribe}$ and $\mathsf{untranscribe}$ interface which guarantees that dishonest certificates are always untranscribed into ledgers that are safe (i.e., the ledger resulting from the untranscription will always be consistent, albeit not up-to-date, with every other honest party's ledger across the execution). This requirement is similar to Roughgarden *et al.*'s notion of *certifiable protocols* [15]. However, we can relax this requirement by using mere *transcriptions* instead of *certificates*. In this case, the $\mathsf{transcribe}$ interface remains the same and returns a transcription τ. However, $\mathsf{untranscribe}$ now takes as input multiple transcriptions $\tau_1, \ldots, \tau_n$ and returns *one* ledger. The $\mathsf{untranscribe}$ function is guaranteed to return a safe and live ledger as long as *at least one* of the transcriptions passed into it was recently generated by an honest party. If the $\mathsf{untranscribe}$ function is invoked with only dishonest transcriptions, no guarantees are provided. This relaxation allows us to work on top of overlay protocols which are not certifiable, but only transcribable, such as Nakamoto consensus (certificates are impossible as chains never adopted by honest parties can be convincing to a receiver who does not see the current longest chain).

To change the ROLLERBLADE construction to make it work on top of transcribable underlying protocols is as follows. The data written on the ledger is identical to the certifiable ROLLERBLADE construction. When the time comes for Λ to $\mathsf{untranscribe}$, it collects all transcriptions of the past recorded on the ledger. If a source ledger i "sends" a message to destination ledger i', with both of them being good, the security proof follows through: Liveness of the receiving ledger guarantees that, every $u_{i'}$ rounds, an honest transcription of i will be recorded, and thus the untranscribed source ledger will be safe.

Practical optimizations. In our ROLLERBLADE construction, we allowed relayers to freely write all checkpoints between all pairs of ledgers. This theoretically demonstrates the breadth of theoretical applicability of our scheme, and highlights the minimal set of axioms required to build compositors. However, if such schemes are to be deployed in practice, the relaying must be optimized. One easy optimization is to have the relayers only send the *delta* between the previous checkpoint and the current one. Additionally, if the underlying ledgers provide smart contract capabilities, one way to do so is to deploy an *on-chain* light client on each destination chain that consumes data from the source chain. Such on-chain light clients are already deployed, for example, in the Cosmos ecosystem [16] and take the form of IBC connections [17]. For more details on how to construct this practically, see TrustBoost [1].

2. Analysis

The following lemma will allow us to argue that all parties share a common view of sufficiently old transactions.

Lemma A.0.1 (Past Perfect)**.** *Consider a temporal ledger protocol Y's execution $\mathcal{E}$ with duration R rounds in which Y is safe, live with liveness u, and timely with timeliness v. If for some honest party P_1 and some round r_1 it holds that $(r^*, \mathsf{tx}) \in {}^{P_1}\boldsymbol{L}_{r_1}$, then for all honest parties P_2 and for all rounds $r_2 > r^* + v$ it holds that $(r^*, \mathsf{tx}) \in {}^{P_2}\boldsymbol{L}_{r_2}$, as long as at least one new honest transaction tx' appears at any round $r_1 < r_3 \leq R - u$.*

Proof. Consider an execution as in the statement and suppose, towards a contradiction, that $(r^*, \mathsf{tx}) = {}^{P_1}\boldsymbol{L}_{r_1}[k]$ for some $k \in \mathbb{N}$, but $(r^*, \mathsf{tx}) \notin {}^{P_2}\boldsymbol{L}_{r_2}$ with $r^* + v < r_2$. From safety, ${}^{P_2}\boldsymbol{L}_{r_2} \prec {}^{P_1}\boldsymbol{L}_{r_1}$ and $|{}^{P_2}\boldsymbol{L}_{r_2}| \leq k < |{}^{P_1}\boldsymbol{L}_{r_1}|$. Due to liveness, $(r', \mathsf{tx}') = {}^{P_2}\boldsymbol{L}_{r_3+u}[k']$, for some $r', k' \in \mathbb{N}$. As tx' is new, it is not in ${}^{P_1}\boldsymbol{L}_{r_1}$. Due to safety, $k' \geq |{}^{P_1}\boldsymbol{L}_{r_1}| > k$. Due to safety, ${}^{P_2}\boldsymbol{L}_{r_3+u}[k] = (r^*, \mathsf{tx})$. Therefore $(r^*, \mathsf{tx}) \in {}^{P_2}\boldsymbol{L}_{r_3+u}[|{}^{P_2}\boldsymbol{L}_{r_2}|:]$. Since $r^* < r_2 - v$, this contradicts the timeliness with parameter v. □

Definition A.1 (Rollerblade Belief System and Honesty Correspondence)**.** Given a sequence of ROLLERBLADE-underlying-respecting protocols $\mathcal{Y} = (\mathsf{Y}^1, \ldots, \mathsf{Y}^\ell)$, define for each $i \in [\ell]$ the predicate, on a collective execution $\mathcal{E}$ of $\mathcal{Y}$, $\text{good}_i(\mathcal{E}) =$ "Y^i is safe, live(u_i), and timely(v_i) in $\mathcal{E}$".

Next, for any $H \subseteq [\ell]$, we define $\text{good}_H(\mathcal{E}) =$ "$\forall i \in H : \text{good}_i(\mathcal{E})$".

Define $\mathcal{H}^{-1}$ on the domain $2^{[n]}$ as $\mathcal{H}^{-1}(H) = \text{good}_H$. Then the ROLLERBLADE *honesty correspondence* $\mathcal{H}$ is the inverse of $\mathcal{H}^{-1}$ and the ROLLERBLADE *belief system* $\mathcal{B}$ is the domain of $\mathcal{H}$.

For example, if $\mathcal{Y} = (\mathsf{Y}^1, \mathsf{Y}^2, \mathsf{Y}^3)$, then for $B = \text{good}_1 \wedge \text{good}_3$, it holds that $B \in \mathcal{B}$ and $\mathcal{H}(B) = \{1, 3\}$.

As an example of a Y^1, consider the Bitcoin Backbone protocol. Then a good_1-respecting environment $\mathcal{Z}$ is an environment that ensures the Bitcoin Backbone protocol execution is safe, live, and timely. This can be done, for example, by demanding that the corruption of Y^1 parties outside of Λs remains in the minority. In case the environment detects an upcoming security violation (due to a negligible event such as a Random Oracle collision), it can conclude the execution.

Theorem A.1 (Replication)**.** *For all* ROLLERBLADE-*suitable-overlay protocols* Π *and any sequence of* ROLLERBLADE-*suitable-underlying protocols* $\mathcal{Y} = (\mathsf{Y}^1, \ldots, \mathsf{Y}^\ell)$ *for the* ROLLERBLADE-*belief-system* $\mathcal{B}$ *of* $\mathcal{Y}$ *and* ROLLERBLADE-*honesty-correspondence* $\mathcal{H}$*, the compositor* ROLLERBLADE *of Section 4 is* $(\Pi, n, \mathcal{Y}, \mathcal{B}, \mathcal{H}, \Delta_u, \Delta_v)$-replication-faithful, *where* $n = \ell$, $\Delta_u = \max\{u_i\}_{i\in[n]}$, $\Delta_v = \max\{v_i\}_{v\in[n]}$ *and* u_i, v_i *are the promised liveness and timeliness of* Y^i.

Proof. The function $\mathsf{emuSnapshot}$ of party j (resp. party j') calls the deterministic function $\mathsf{emulate}$ with overlay

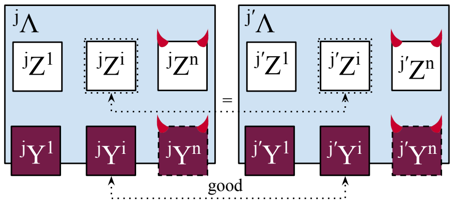

Figure 2. The Replication Theorem (Theorem A.1) states that the execution of the emulated machines ${}^{j}\mathsf{Z}^i$ and ${}^{j'}\mathsf{Z}^i$ across two different compositor parties ${}^{j}\Lambda$ and ${}^{j'}\Lambda$ is consistent, as long as the underlying ledger Y^i is *good*.

party index i, ledger ${}^{j}\boldsymbol{L}^i_{r+v_i}$ (resp. ledger ${}^{j'}\boldsymbol{L}^i_{r+v_i}$), emulation round r, and current round $r + v_i$. The function $\mathsf{emulate}$ runs its main *for* loop (Algorithm 5 Line 16) up to r (inclusive), which consumes data from $\mathsf{writeboxes}[r-1]$ and $\mathsf{inboxes}[r-1]$ and earlier. These are produced by the function $\mathsf{prepEmulnputs}$ by looking at transactions recorded in ${}^{j}\boldsymbol{L}^i_{r+v_i}$ (resp. ${}^{j'}\boldsymbol{L}^i_{r+v_i}$) with recorded round $< r$. Because $\boldsymbol{L}^i$ is *good* in $\mathcal{E}$, it is safe, live(u_i), and timely(v_i). It suffices to show that all transactions with recorded round $< r$ are the same in ${}^{j}\boldsymbol{L}^i_{r+v_i}$ and ${}^{j'}\boldsymbol{L}^i_{r+v_i}$. This holds because of Lemma A.0.1 invoked with parties ${}^{j}\mathsf{Y}^i$ and ${}^{j'}\mathsf{Y}^i$, rounds $r_1 = r_2 = r + v_i$, $r_3 = r + v_i$ and $r^* < r$. During round r_3, the new honest transaction is due to any honest relayer. □

Lemma A.1.1 (Consistency)**.** *Consider a* ROLLERBLADE Λ *execution* $\mathcal{E}$ *with duration* R *rounds sampled from* $\text{RUN}^{\mathit{ES}}_m(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$ *with overlay protocols* $\mathcal{Y}$*, adversary* $\mathcal{A}$ *and environment* $\mathcal{Z}$*, and let* $\Delta_v = \max\{v_i\}_{i\in[n]}$*, where* v_i *denotes the timeliness promised by ledger protocol* ${}^{i}\mathsf{Y}$*. Let* $H \subseteq [n]$ *be the subset of good underlying ledger protocol indices among* $\mathcal{Y}$ *in* $\mathcal{E}$*. Then the execution is* (j, H, Δ_v)*-consistent for all* $j \in [m]$.

Proof. Let $i \in H$, $r \geq 0$, $r + \Delta_v < r' < R$ be arbitrary. When $\mathsf{emuSnapshot}$ is invoked at the end of round r' (resp. $r' + 1$), the value $\mathsf{this.now}$ is r' (resp. $r' + 1$), so the check $\mathsf{emuRound} > \mathsf{this.now} - v_i$ of Algorithm 6 Line 2 is *false*, and the emulation succeeds. This is the only stateful check in this function. The result of the function $\mathsf{prepEmulnputs}$ only depends on the transactions of its ${}^{j}\boldsymbol{L}^i$ argument with recorded rounds up to $\mathsf{emuRound}$. Therefore it suffices to show that the transactions in ${}^{j}\boldsymbol{L}^i_{r'}$ with recorded rounds up to $\mathsf{emuRound}$ are the same as the transactions in ${}^{j}\boldsymbol{L}^i_{r'+1}$ with recorded rounds up to $\mathsf{emuRound}$. All ${}^{j}\boldsymbol{L}^i_{r'}$ transactions with recorded rounds up to $\mathsf{emuRound}$ are included in ${}^{j}\boldsymbol{L}^i_{r'+1}$ by *stickiness*. Conversely, all ${}^{j}\boldsymbol{L}^i_{r'+1}$ transactions with recorded rounds up to $\mathsf{emuRound}$ are included in ${}^{j}\boldsymbol{L}^i_{r'}$ by *timeliness*. □

Lemma A.1.2. *Consider a rollerblade execution* $\mathcal{E}$ *with duration* $R + \Delta_v$ *in the emulated setting with arbitrary good ledgers* Y^i, $\mathsf{Y}^{i'}$*, rollerblade party* ${}^{j}\Lambda$ *and round* $1 \leq r^* \leq R$*. Consider the call* ${}^{j}\Lambda.\mathsf{prepEmulnputs}(i', {}^{j}\boldsymbol{L}^{i'}_{\hat{r}}, \hat{r} - \Delta_v - 1)$*. for some* $\hat{r} < R - u_{i'}$*. Within that* in vitro *execution,* *consider any iteration (r^*, tx^*) of the* for *loop in Line 16 of Algorithm 6. Let f^* denote the value attained by the variable F^i at the* end *of that iteration. If $r^* < \hat{r} - v_{i'}$, then the transactions with recorded round up to $r = r^* - v_i - u_{i'} - 1$ in f^* and ${}^j\boldsymbol{L}^i_R$ are the same.*

Sketch. The proof is analogous to the Lemma A.0.1 proof. □

Theorem A.2 (Emulation). *For all* ROLLERBLADE-*suitable-overlay protocols* Π *and any sequence of* ROLLERBLADE-*suitable-underlying protocols* $\mathcal{Y} = (\mathsf{Y}^1, \ldots, \mathsf{Y}^\ell)$ *for the* ROLLERBLADE-*belief-system* $\mathcal{B}$ *of* $\mathcal{Y}$ *and* ROLLERBLADE-*honesty-correspondence* $\mathcal{H}$, *the compositor* ROLLERBLADE *of Section 4 is* $(\Pi, n, \mathcal{Y}, \mathcal{B}, \mathcal{H}, \Delta_u, \Delta_v)$-emulation-faithful, *where* $n = \ell$, $\Delta_u = \max\{u_i\}_{i\in[n]}$, $\Delta_v = \max\{v_i\}_{v\in[n]}$ *and* u_i, v_i *are the promised liveness and timeliness of* Y^i.

Proof. Let Π, $\mathcal{Y}$, $\mathcal{B}$ and $\mathcal{H}$ be as in the statement, and $\Delta = 2\Delta_v + \Delta_u$. Let the adversary $\mathcal{A}$, the belief $B \in \mathcal{B}$, and the environment $\mathcal{Z}$, the number of compositor parties m, and the index of the compositor j of interest be arbitrary as in the statement, and set $H = \mathcal{H}(B)$. Let $\mathcal{E}$ and $\mathcal{E}'$ be the emulated and party setting executions, respectively, sampled as in Definition 3.19. As Π is a ROLLERBLADE-suitable-overlay protocol it is deterministic.

We will prove faithfulness by construction of the simulator $\mathcal{S}$ and environment $\mathcal{Z}'$.

The simulator $\mathcal{S}$ and environment $\mathcal{Z}'$ work in tandem as follows. Initially, $\mathcal{S}$ samples an execution $\mathcal{E}^*$ in the emulation setting from $\text{RUN}^{\mathsf{ES}}_m(\Lambda, \mathcal{Y}, \mathcal{A}, \mathcal{Z})$ (see Figure 3). Let R be the duration of $\mathcal{E}^*$ in rounds. The simulator looks at compositor party j of $\mathcal{E}^*$ and its ${}^j\mathsf{Y}^1, \ldots, {}^j\mathsf{Y}^n$. The environment $\mathcal{Z}'$ chooses the duration, in rounds, of $\mathcal{E}'$ to be $R - \Delta_v - 1$. It initializes n parties $\Pi^1, \ldots, \Pi^i, \ldots, \Pi^n$ by invoking the *constructor* method with parameters Δ, i, n.

The simulator initially obtains, for every $i \in [n]$, a copy of the configuration M_i of the ITI ${}^j\mathsf{RB}$ from $\mathcal{E}^*$ at the end of $\mathcal{E}^*$. The simulator calls $\mathsf{prepEmuInputs}(i, {}^j\boldsymbol{L}^i_R, R-\Delta_v-1)$ on M_i *in vitro* at the end of round R, to obtain the pair $(\mathsf{writeboxes}^i, \mathsf{inboxes}^i)$, where $|\mathsf{writeboxes}^i| = R-\Delta_v-1$ and $|\mathsf{inboxes}^i| = R - \Delta_v - 1$. At the beginning of round r of $\mathcal{E}'$, the simulator calls $\mathsf{write}(\mathsf{data})$ on party i for every $\mathsf{data} \in \mathsf{writeboxes}^i[r-1]$. At every round, $\mathcal{Z}'$ activates each party, in order, by invoking $\Pi^i.\mathsf{execute}()$. If party Π^i invokes the network receive method $\mathsf{recv}()$, while active, then $\mathcal{Z}'$ provides $\mathsf{inboxes}^i[r-1]$. If Π^i invokes the network send method $\mathsf{send}(i', \mathsf{msg})$, while active, then it is ignored by $\mathcal{Z}'$.

We note that $\mathcal{S}$ uses the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$ in this simulation, so $\mathcal{E}$ and $\mathcal{E}^*$ are identically distributed. For (1) it suffices to show that for all j, $\text{VIEW}^{\mathsf{ES}}_{j,H,\Delta_v}(\mathcal{E}^*) = \text{VIEW}^{\mathsf{PS}}_H(\mathcal{E}')$ (i.e., we will show the these two random variables are equal, not just equal by distribution), and similarly for (2) it suffices to show that for all j, $\text{EXTERN}^{\mathsf{ES}}_j(\mathcal{E}^*) \lessapprox_{\Delta_u,\Delta_v} \text{EXTERN}^{\mathsf{PS}}(\mathcal{E}')$ (i.e., we

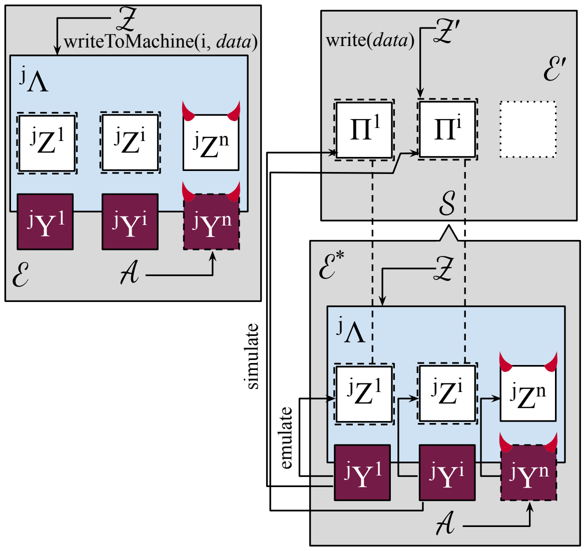

Figure 3. A visualization of the argument that ROLLERBLADE is faithful. In the proof of Theorem A.2, the simulator $\mathcal{S}$ and simulated environment $\mathcal{Z}'$ give rise to execution $\mathcal{E}'$ in the party setting (top right) by sampling an execution $\mathcal{E}^*$ in the emulated setting with adversary $\mathcal{A}$ and environment $\mathcal{Z}$ (bottom right). The simulator uses each good ledger ${}^i\mathsf{Y}^j$ in $\mathcal{E}^*$ to feed data into the respective honest party Π^i in $\mathcal{E}'$.

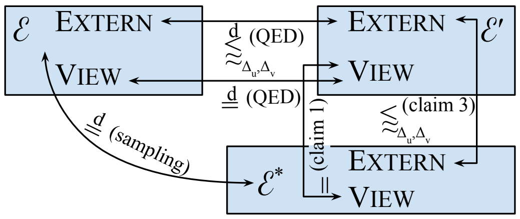
$\mathcal{E}$
EXTERN
VIEW
$\overset{d}{\lessapprox}_{\Delta_u,\Delta_v}$ (QED)
$\overset{d}{=}$ (QED)
EXTERN
VIEW
$\mathcal{E}'$
$\overset{d}{=}$ (sampling)
$\lessapprox_{\Delta_u,\Delta_v}$ (claim 3)
= (claim 1)
$\mathcal{E}^*$
EXTERN
VIEW

Figure 4. The environments $\mathcal{Z}$ in $\mathcal{E}$ and $\mathcal{Z}'$ in $\mathcal{E}'$ respectively give rise to externalities similar in distribution ($\overset{d}{\lessapprox}_{\Delta_u,\Delta_v}$, middle top), whereas the views of the honest parties in $\mathcal{E}$ and $\mathcal{E}'$ are distributionally equivalent ($\overset{d}{=}$, middle top). The executions $\mathcal{E}$ and $\mathcal{E}^*$ are sampled from the same distribution ($\overset{d}{=}$, bottom left). The views of the honest parties in $\mathcal{E}^*$ and $\mathcal{E}'$ are equal (not just equal in distribution) and the externalities in $\mathcal{E}^*$ and $\mathcal{E}'$ similar (not just similar in distribution).

will show that the externalities of $\mathcal{E}'$ are similar to $\mathcal{E}^*$, not just *distributionally* similar). See Figure 4.

Since $\mathcal{Z}$ is B-respecting, therefore $B(\mathcal{E}^*)$ holds. Hence, for all $i \in H$, Y^i is safe, live (u_i), and timely (v_i) in $\mathcal{E}^*$.

Claim 1: $\text{VIEW}^{\mathsf{ES}}_{j,H,\Delta_v}(\mathcal{E}^*) = \text{VIEW}^{\mathsf{PS}}_H(\mathcal{E}')$. Let $V^* = \text{VIEW}^{\mathsf{ES}}_{j,H,\Delta_v}(\mathcal{E}^*)$ and $V' = \text{VIEW}^{\mathsf{PS}}_H(\mathcal{E}')$. We know that $\mathcal{E}^*$ is (j, H, Δ_v)-consistent from Lemma A.1.1, and so V^* is well-defined. The two views have the same size. We need to show that the views of all *honest* parties in the two executions are identical. Let i be an arbitrary party in H.

Let $c^*(r_1, r_2, r_3)$ denote the configuration of the machine ${}^j\mathsf{Z}^i$ after the invocation of Algorithm 5 Line 26 during the r_3-rd iteration of the *for loop* in Algorithm 5 Line 16 (or, if $r_3 = 0$, we set $c^*(r_1, r_2, r_3)$ to be the state of ${}^j\mathsf{Z}^i$ before the *for loop*) when $\mathsf{emuSnapshot}$ is invoked *in vitro* after round r_1 of execution $\mathcal{E}^*$, where $r_2 + \Delta_v \le r_1 \le R + \Delta_v$ with parameter $\mathsf{emuRound} = r_2$, and $r_3 \le r_2$. This causes an invocation of $\mathsf{emulate}$ with parameters $\mathsf{emuRound} = r_2$ and $\mathsf{realRound} = r_1$. Let $c'(r_3)$ denote the configuration of the machine Π^i in $\mathcal{E}'$ at the end of round r_3. We set $c'(0)$ to be the initial configuration of Π^i, after it's *constructor* is invoked.

We will prove that $V^*_{i,r} = V'_{i,r}$ by induction on r. Let r be an arbitrary round such that $1 \le r \le R - \Delta_v - 1$.

If $r = 1$, then

$$c^*(r + \Delta_v, r-1, r-1) = c'(r-1) \tag{1}$$

because the state of machine ${}^j\mathsf{Z}^i$ and Π^i are identical after the initial invocation of the *constructor* as Π is deterministic.

If $r > 1$, then by inductive hypothesis we have $V^*_{i,r-1} = V'_{i,r-1}$ by which Eq 1 also follows.

In either case, Eq 1 holds, and we wish to show that $c^*(r + \Delta_v, r, r) = c'(r)$, from which it will follow that $V^*_{i,r} = V'_{i,r}$.

Because $\mathcal{E}^*$ is consistent, we have that

$$c^*(r + \Delta_v, r-1, r-1) = c^*(r + \Delta_v, r-1, r-1)\,. \tag{2}$$

Each iteration of the *for* loop of Algorithm 5 Line 16 proceeds until the round reaches $\mathsf{emuRound}$, therefore $c^*(r + \Delta_v, r, r-1) = c^*(r + \Delta_v, r-1, r-1)$. From this and Eq 2 we conclude that $c^*(r + \Delta_v, r, r-1) = c^*(r + \Delta_v, r-1, r-1)$. From this and Eq 1 we conclude that

$$c^*(r + \Delta_v, r, r-1) = c'(r-1)\,. \tag{3}$$

Let $(\mathsf{writeboxes}^*, \mathsf{inboxes}^*)$ be the pair returned by the invocation of $\mathsf{prepEmuInputs}(i, {}^j\boldsymbol{L}^i_{r+\Delta_v}, r)$ *in vitro* after round $r + \Delta_v$ for party ${}^j\Lambda$ of $\mathcal{E}^*$, and, likewise define $(\mathsf{writeboxes}', \mathsf{inboxes}')$ to be the pair returned by the invocation of $\mathsf{prepEmuInputs}(i, {}^j\boldsymbol{L}^i_R, R - \Delta_v - 1)$ *in vitro* after round R of $\mathcal{E}^*$ on party ${}^j\Lambda$. Because ${}^j\mathsf{Y}^i$ in $\mathcal{E}^*$ is *sticky* and *timely*(v_i) the transactions in ${}^j\boldsymbol{L}^i_{r+\Delta_v}$ with recorded round $r-1$ are identical to the transactions in ${}^j\boldsymbol{L}^i_R$ with recorded round $r-1$ (all transactions with recorded round $r-1$ of ${}^j\boldsymbol{L}^i_{r+\Delta_v}$ will appear in ${}^j\boldsymbol{L}^i_R$ by stickiness; and all transactions with recorded round $r-1$ of ${}^j\boldsymbol{L}^i_R$ will have appeared in ${}^j\boldsymbol{L}^i_{r+\Delta_v}$ by timeliness; in the case of $r = 1$, no transactions with recorded round $r-1$ will appear in either ledger). Hence,

$$\mathsf{writeboxes}^*[r-1] = \mathsf{writeboxes}'[r-1] \tag{4}$$

$$\mathsf{inboxes}^*[r-1] = \mathsf{inboxes}'[r-1]\,, \tag{5}$$

since the loop of Algorithm 6 Line 16 will run identically in the two invocations of $\mathsf{prepEmuInputs}$ up to and including round $r-1$ (and for $r=1$, $\mathsf{writeboxes}^*[0] = \mathsf{inboxes}^*[0] = \mathsf{writeboxes}'[0] = \mathsf{inboxes}'[0]$ by construction).

By the ROLLERBLADE construction, $c^*(r + \Delta_v, r, r)$ is the result of taking the machine configuration $c^*(r + \Delta_v, r, r-1)$ and running the *for* loop of Algorithm 5 Line 16 for one more iteration, providing inputs $\mathsf{writeboxes}^*[r-1], \mathsf{inboxes}^*[r-1]$ during the execution of ${}^j\mathsf{Z}^i$. To see why $\mathsf{writeboxes}^*, \mathsf{inboxes}^*$ in particular are used in this iteration, note that these correspond to the invocation of $\mathsf{prepEmuInputs}$ in Algorithm 5 Line 2.

Similarly, by the simulator construction, the simulator evolves the machine Π^i in $\mathcal{E}'$ from round $r-1$, in which it has configuration $c'(r-1)$, to round r, in which it has configuration $c'(r)$, by feeding it the inputs $\mathsf{writeboxes}'[r-1], \mathsf{inboxes}'[r-1]$.

As $c^*(r + \Delta_v, r, r-1) = c'(r-1)$ (by Eq 3) and the two configurations evolve using the same inputs (by Eqs 4 and 5), and, since Π is deterministic, therefore $c^*(r + \Delta_v, r, r) = c'(r)$ (see Figure 5). We conclude that $V^*_{i,r} = V'_{i,r}$, completing the induction and the proof of Claim 1.

Claim 2: $\mathcal{Z}'$ respects the network model. Namely, the claim is that for all $i, i' \in H$, for all rounds r of $\mathcal{E}'$:

(a) If Π^i sends a message to $\Pi^{i'}$ at round r, then this message is delivered to $\Pi^{i'}$ (with source i) between round $r+1$ and $r+\Delta$, and
(b) If $\Pi^{i'}$ received a message from Π^i at round r^*, then this message was sent by Π^i.

Let r be the round during which Π^i sends message msg to $\Pi^{i'}$. That means that Π^i invoked network function $\mathsf{send}(i', \mathsf{msg})$ at round r. In Claim 1, we showed that the transcript of machine Π^i in $\mathcal{E}'$ is identical to the transcript of machine ${}^j\mathsf{Z}^i = {}^j\Lambda.\mathsf{emuSnapshot}(i, r)$ in $\mathcal{E}^*$ called in vitro at the end of round $r + \Delta_v$, which, by consistency, is the same as ${}^j\Lambda.\mathsf{emulate}(i, {}^j\boldsymbol{L}^i_{R-\Delta_v-1}, r).Z$ called in vitro at the end of round R. Hence, ${}^j\mathsf{Z}^i$ also invoked network function $\mathsf{send}(i', \mathsf{msg})$ at round r. This means that $\{\mathsf{to} : i', \mathsf{msg} : \mathsf{msg}\} \in {}^j\Lambda.\mathsf{emulate}(i, {}^j\boldsymbol{L}^i_{R-\Delta_v-1}, r).\mathsf{outboxes}[r]$.

Let (r^*, tx^*) be the first bulletin 'chkpt' transaction from Y^i in ${}^j\boldsymbol{L}^{i'}_R$, such that $r^* > r + v_i + u_{i'}$. Such a transaction (r^*, tx^*) will exist because party ${}^j\Lambda$ will relay Y^i at round $r + v_i + v_{i'}$, and because of the liveness of $\mathsf{Y}^{i'}$, it will appear on ${}^j\boldsymbol{L}^{i'}$ for the first time no later than round $r + v_i + u_{i'} + v_{i'}$ with recorded round after $r + v_i + u_{i'}$ but no later than $r + v_i + u_{i'} + v_{i'}$. Therefore $r^* \le r + v_i + u_{i'} + v_{i'}$.

Let f^* be the value of F^i in Line 28 during iteration with parameter (r^*, tx^*). By Lemma A.1.2, the transactions with recorded round up to $r \le r^* - v_i - u_{i'} - 1$ in ${}^j\boldsymbol{L}^i_{R+\Delta_v}$ and f^* are the same.

In order to collect the incoming messages for $\Pi^{i'}$, the simulator calls *in vitro* (at the end of round R) ${}^j\Lambda.\mathsf{prepEmuInputs}(i', {}^j\boldsymbol{L}^{i'}_{R-\Delta_v-1}, R - \Delta_v - 1)$. In Line 28, during iteration with parameter (r^*, tx^*), function ${}^j\Lambda.\mathsf{emulate}(i, F^i, r^* - v_i - u_{i'} - 1)$ is invoked. It holds that $\{\mathsf{to} : i', \mathsf{msg} : \mathsf{msg}\} \in {}^j\Lambda.\mathsf{emulate}(i, F^i, r^* - v_i - u_{i'} - 1).\mathsf{outboxes}[r]$. By the minimality of (r^*, tx^*), the msg will be included in $\mathsf{inboxes}[r^*]$. The environment $\mathcal{Z}'$ will provide msg to $\Pi^{i'}$ if network method $\mathsf{recv}()$ is invoked at round r^*.

We conclude that since msg sent at r was delivered at r^*, and $r^* \le r + v_i + u_{i'} + v_{i'} \le r + 2\Delta_v + \Delta_u \le r + \Delta$,

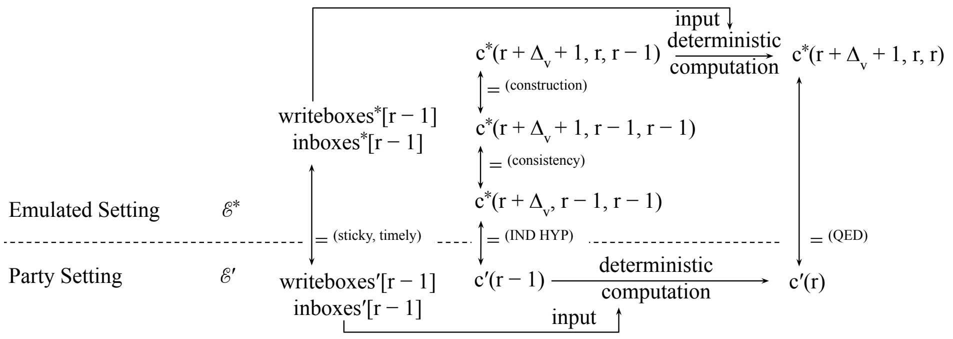

Figure 5. The inductive step in the proof of the Emulation Theorem.

the delay is respected. We observe that the delay for every message is always greater than $u_i + v_{i'}$.

This completes the proof of (a).

Let msg be the message that $\Pi^{i'}$ received from Π^i at round r^*. To prove (b), it suffices to show that Π^i invoked network function $\mathsf{send}(i', \mathsf{msg})$ at some round r.

The message msg is received at round r^* if $\mathcal{Z}'$ provides it to $\Pi^{i'}$ after network method $\mathsf{recv}()$ is invoked during round r^*. Hence, the msg must be included in $\mathsf{inboxes}[r^*]$ after ${}^j\Lambda.\mathsf{prepEmuInputs}(i', {}^j\boldsymbol{L}^{i'}_R, R - \Delta_v - 1)$ is called *in vitro* at the end of round R by the simulator. Therefore, there is a bulletin 'chkpt' transaction (r^*, tx^*), such that when the *for* loop in Algorithm 6 Line 16 is entered with it as parameter, it holds that $\{\mathsf{to} : i', \mathsf{msg} : \mathsf{msg}\} \in {}^j\Lambda.\mathsf{emulate}(i, F^i, r^* - v_i - u_{i'} - 1).\mathsf{outboxes}[r]$, where $r \le r^* - v_i - u_{i'} - 1$.

Let f^* be the value of F^i in Line 28 during iteration with parameter (r^*, tx^*). By Lemma A.1.2, the transactions with recorded round up to $r \le r^* - v_i - u_{i'} - 1$ in ${}^j\boldsymbol{L}^i_{R+\Delta_v}$ and f^* are the same. Therefore, it holds that ${}^j\Lambda.\mathsf{emulate}(i, f^*, r^* - v_i - u_{i'} - 1).\mathsf{outboxes}[r] = {}^j\Lambda.\mathsf{emulate}(i, {}^j\boldsymbol{L}^i_R, r).\mathsf{outboxes}[r]$. That means that ${}^j\mathsf{Z}^i$ invoked network function $\mathsf{send}(i', \mathsf{msg})$ at round r. In Claim 1, we showed that the transcript of machine Π^i in $\mathcal{E}'$ is identical to the transcript of machine ${}^j\mathsf{Z}^i = {}^j\Lambda.\mathsf{emuSnapshot}(i, r)$ in $\mathcal{E}^*$ called in vitro at the end of round $r + \Delta_v$, which, by consistency, is the same as ${}^j\Lambda.\mathsf{emulate}(i, {}^j\boldsymbol{L}^i_R, r).Z$ called in vitro at the end of round R. Hence, Π^i also invoked network function $\mathsf{send}(i', \mathsf{msg})$ at round r.

This completes the proof of claim (b).

Claim 3: $\textsc{Extern}^{\mathsf{ES}}_{j,H}(\mathcal{E}^*) \lessapprox_{\Delta_u,\Delta_v} \textsc{Extern}^{\mathsf{PS}}(\textsc{View}^{\mathsf{PS}}_{\mathcal{H}(B)}(\mathcal{E}'))$. Let $i \in H$ and $r \le R - \Delta_u - \Delta_v - 1$ be arbitrary, and W^i_r be the writebox at position (i, r) of $\textsc{Extern}^{\mathsf{ES}}(\mathcal{E}^*)$. Let m be an arbitrary message in W^i_r. Message m was included in a *writeToMachine* call to ${}^j\mathsf{RB}$ with parameter i at round r. By the rollerblade construction, the *write* function of ${}^j\mathsf{Y}^i$ will be called by ${}^j\mathsf{RB}$ at round r with parameter a bulletin transaction tx with payload ('write', m). Because bulletin transactions are high entropy, this transaction is

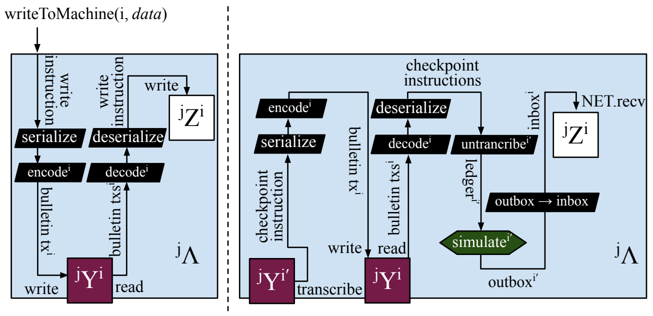

Figure 6. The two types of instructions recorded on-ledger to enable the emulation. On the left, the flow of a *write* instruction, indicating user (environment) input is illustrated. Whereas the environment invokes *writeToMachine*, the instruction, after becoming recorded on the ledger and appropriately translated, eventually makes it as a *write* call to the emulated machine ${}^j\mathsf{Z}^i$. On the right, the flow of a *checkpoint* instruction, initiated by the relaying process of a rollerblade client, is illustrated. The source ledger (i') is transcribed onto the destination ledger (i). This transcription is later untranscribed and used as the input to a recursive emulation call to produce the network outputs of a machine i', which are, upon appropriate translation, given to the machine ${}^j\mathsf{Z}^i$ as network input.

fresh, namely it does not appear in ${}^j\boldsymbol{L}^i_r$. Because i is *good*, therefore it is live in $\mathcal{E}^*$ with liveness u_i, therefore tx will appear in ${}^j\boldsymbol{L}^i_{r+u_i}$ with some recorded round r^*. Note that round $r + u_i \le R$ falls within the duration of $\mathcal{E}^*$, and so the ledger ${}^j\boldsymbol{L}^i_{r+u_i}$ can be obtained by the simulator. Because i is *timely* in $\mathcal{E}^*$ with timeliness v_i, therefore $r{+}1{-}\Delta_v \le r{+}1{-}v_i \le r^* \le r{+}u_i \le r{+}\Delta_u \le R{-}\Delta_v{-}1$. When the function $\mathsf{prepEmuInputs}$ is invoked at $\mathsf{realRound} = R$ and with $\mathsf{emuRound} = R - \Delta_v - 1$, the transaction will contribute to the *writebox* of party i, since $r^* \le R - \Delta_v - 1$. Hence, m will be inputted to the *write* function call of Π^i in $\mathcal{E}'$ at round r^* (and note that $1 \le r^* \le R - \Delta_v - 1$ falls within the duration of $\mathcal{E}'$), completing the proof of Claim 3. □

3. Acknowledgements

The authors wish to thank Joachim Neu for helping discover the necessity of transcribability and certifiability of ledgers to achieve different assurances in the virtualized world, as well as the relationship between safety failures of ledgers and rollback corruptions of simulated parties. We also thank Ertem Nusret Tas, Kostis Karantias, and Srivatsan Sridhar for the insightful discussions during the development of this project.

This paper and the concurrent related work in which blockchains were analyzed as “circuits” [18] both came out of many fruitful discussions about blockchain composability among Ertem Nusret Tas, David Tse, and Dionysis Zindros, when they were all at Stanford University.

This work was partially funded by Input Output Global.