

A Security Framework for General Blockchain Layer 2 Protocols

Zeta Avarikioti
TU Wien & Common Prefix
Vienna, Austria
georgia.avarikioti@tuwien.ac.at

Matteo Maffei
TU Wien
Vienna, Austria
matteo.maffei@tuwien.ac.at

Yuheng Wang
TU Wien
Vienna, Austria
yuheng.wang@tuwien.ac.at

Abstract

Layer 2 (L2) solutions are the cornerstone of blockchain scalability, enabling high-throughput and low-cost interactions by shifting execution off-chain while maintaining security through interactions with the underlying ledger. Despite their common goals, the principal L2 paradigms—payment channels, rollups, and sidechains—differ substantially in architecture and assumptions, making it difficult to comparatively analyze their security and trade-offs.

To address this challenge, we present the first general security framework for L2 protocols. Our framework is based on the IITM-based Universal Composability (iUC) framework, in which L2 protocols are modeled as stateful machines interacting with higher-level protocol users and the underlying ledger. The methodology defines a generic execution environment that captures ledger events, message passing, and adversarial scheduling and characterizes protocol security through trace-based predicates parameterized by adversarial capabilities and timing assumptions. By abstracting away from protocol-specific details while preserving critical interface and execution behavior, the framework enables modular, protocol-agnostic reasoning and composable security proofs across a wide range of L2 constructions.

To demonstrate the framework’s applicability, we analyze an illustrative example from each of the three dominant L2 scaling paradigms: a payment channel (Brick), a sidechain (Liquid Network), and a rollup (Arbitrum). By instantiating each within our framework, we derive their respective security properties and expose key trade-offs. These include: the time required for dispute resolution, the distribution of off-chain storage and computation, and varying trust assumptions (e.g., reliance on honest parties or data availability). Our framework thus not only unifies the analysis of diverse L2 designs but also pinpoints their inherent strengths and limitations, providing a foundation for the secure and systematic development of future L2 systems.

1 Introduction

Despite their promise to transform the financial sector, blockchains face fundamental limitations in scalability, costs, and latency. To address these challenges, a broad class of protocols—collectively referred to as Layer 2 (L2) protocols—offload part of the workload from the base layer (L1) to an auxiliary layer that interacts with the L1 only when necessary [20]. Prominent examples include payment channels such as the Bitcoin Lightning Network [27], rollups like Optimism [26] and Arbitrum [8], and sidechains such as the Liquid Network [25] and Polygon [7].

While these systems are increasingly deployed and architecturally diverse, there is currently no unifying framework to formally reason about their security guarantees in a modular and composable way. Existing definitions are often tailored to specific constructions, making it difficult to compare protocols, identify

necessary and sufficient conditions for desired properties, or build reusable abstractions that enable principled design trade-offs.

In this work, we close this gap by introducing *the first general security framework for L2 protocols*. At the core of our framework lies the **ideal functionality** $\mathcal{F}_{\text{layer2}}^\Lambda$, which formalizes the key security guarantees that any L2 protocol Λ should satisfy. The functionality abstracts over protocol-specific mechanisms to capture the essential workflow common to all L2 constructions: joining the L2, submitting off-chain requests, updating the state, reading the state, and exiting the L2. Each phase of L2 interaction is captured by a separate subroutine ($\mathcal{F}_{\text{open}}$, $\mathcal{F}_{\text{submit}}$, $\mathcal{F}_{\text{update}}$, $\mathcal{F}_{\text{read}}$, $\mathcal{F}_{\text{settlement}}$), invoked through a central interface machine (see Figure 1). This decomposition enables formal reasoning about security, performance, and trust assumptions across a wide spectrum of designs, within a composable setting.

To demonstrate the generality and practical relevance of our framework, we instantiate it with three qualitatively distinct L2 protocols: the Brick *payment channel* [5], the Liquid Network *sidechain* [25], and the Arbitrum Nitro *rollup* protocol [8]. These case studies highlight how $\mathcal{F}_{\text{layer2}}$ unifies diverse constructions under a single formal umbrella, reveals structural commonalities, and sharpens the articulation of trade-offs.

Beyond capturing existing protocols, our framework reveals and formalizes fundamental trade-offs among safety, liveness, and data availability. We show that the way an L2 protocol instantiates core subroutines determines what guarantees it can provide. For example, payment channels offer instant finality but are vulnerable to crash faults; rollups inherit safety from L1 but incur latency and high on-chain storage cost; sidechains rely on internal consensus to balance trust and performance but naturally achieve lower security guarantees. These distinctions, long understood informally, are made precise in our framework and shown to reflect inherent limitations in protocol design. In particular, we prove a set of lower bounds (Theorems 5.4–5.6) that characterize the constraints each L2 paradigm faces under standard assumptions.

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

- We propose the first general security framework for L2 protocols, formalized as an ideal functionality $\mathcal{F}_{\text{layer2}}$ with modular subroutines that capture core L2 operations and support composable, protocol-agnostic reasoning (Sec. 3).
- We apply our framework to three qualitatively distinct L2 protocols—Brick (payment channel), Liquid (sidechain), and Arbitrum Nitro (rollup)—demonstrating its expressiveness and generality (Sec. 4).
- We formalize and prove key trade-offs between safety, liveness, and data availability across L2 designs, showing that these are intrinsic to subroutine instantiations (Sec. 5).

2 Model

We now formalize the system and threat model that underpins our framework. The goal is to abstract the behavior of a broad class of L2 protocols without committing to specific implementation details, allowing security analysis to be carried out in a modular and composable way.

System Model and Assumptions. In this paper, we focus exclusively on the primitive operation of L2 protocols, considering only single-hop constructions where interactions occur directly between a client and the operator or supporting entities. Multi-hop extensions, such as payment channel hubs [28] and multi-hop payment protocols [2], are out of scope.

All parties are assumed to be probabilistic polynomial-time (PPT) interactive Turing machines. We assume access to a cryptographic ideal functionality $\mathcal{F}_{\text{cert}}$ that supports EUF-CMA secure signature schemes. If synchronous communication is required by the protocol, we assume the existence of a synchronous communication functionality \mathcal{F}_{com} . Finally, we model the L1 blockchain as an ideal ledger functionality $\mathcal{F}_{\text{ledger}}$, following the definition in [19]. We abstract network conditions and protocol logic through dedicated subroutines within our framework to preserve generality and modularity.

In practice, while terminology may vary across L2 designs, most protocols involve three core roles:

- **Operator:** Also referred to as channel owners, sequencers, or maintainers, operators are responsible for processing protocol requests, managing off-chain state transitions, and ensuring the correct execution of transactions.
- **Client:** Clients are users who join the L2 protocol to issue execution requests that update the off-chain state. They may initiate settlement procedures to commit the current state to the L1 ledger and can also query the protocol for its current status.
- **Third-party participants:** This role encompasses auxiliary actors that support the protocol, such as watchtowers that monitor L1 for disputes on behalf of clients, or validators responsible for verifying state transitions.

We assume that the set of operators and third-party participants in the L2 protocol is *static* and publicly known, treated as global parameters. While some protocols adopt *dynamic* operator sets to preserve security over long time periods, we fix the participant set for simplicity. In our analysis, we assume that any required trust assumptions, such as bounds on the adversarial corruption ratio, hold throughout the execution of the protocol.

Threat Model. We adopt the standard honest/Byzantine adversarial model. The adversary may corrupt a subset of participants up to a fixed bound defined by the protocol assumptions. Corrupted parties may arbitrarily deviate from the protocol, including submitting invalid inputs, crashing, or withholding messages. However, they cannot forge digital signatures or drop messages between honest parties, as this would violate the underlying cryptographic and network assumptions.

3 The Ideal Functionality for L2 Protocols

In this section, we formalize the core contribution of this work: an ideal functionality $\mathcal{F}_{\text{layer2}}$ that captures the essential behavior

and security properties of general L2 protocols. The functionality is defined in the iUC framework [10]¹ and is composed of modular subroutines corresponding to the phases of an L2 protocol’s lifecycle. We begin by describing the design rationale, followed by a detailed presentation of the subroutines and their interactions.

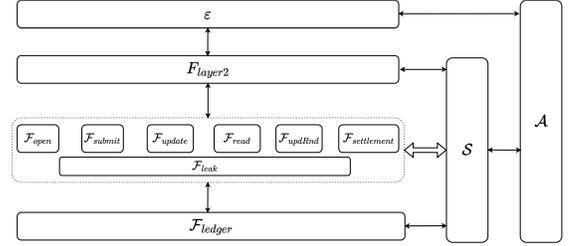


Figure 1: Structure of the ideal functionality $\mathcal{F}_{\text{layer2}}^\Lambda$ for a secure L2 protocol Λ . \mathcal{E} refer to the environment, \mathcal{S} refer to the simulator and \mathcal{A} refer to the adversary.

Design Rationale. The ideal functionality $\mathcal{F}_{\text{layer2}}$ is designed to capture the core phases of interaction in a broad class of L2 protocols, from payment channels to rollups and sidechains. Its structure reflects a minimal and modular decomposition of protocol behavior that abstracts away implementation-specific details while preserving the key security and performance considerations common to L2 designs.

Each subroutine models a fundamental aspect of L2 operation:

- $\mathcal{F}_{\text{open}}$ models the act of joining the L2 system, such as establishing a channel or registering with a rollup or sidechain.
- $\mathcal{F}_{\text{submit}}$ captures how clients issue off-chain requests or transactions.
- $\mathcal{F}_{\text{update}}$ formalizes how state transitions are agreed upon and executed—whether via unilateral action (channels), consensus (sidechains), or sequencer output (rollups).
- $\mathcal{F}_{\text{read}}$ reflects the visibility of the off-chain state, modeling different data availability assumptions.
- $\mathcal{F}_{\text{settlement}}$ abstracts the final anchoring of off-chain state back to the L1 ledger.
- $\mathcal{F}_{\text{updRnd}}$ simulates the internal protocol clock to capture time-related behaviors, such as synchronous communication.
- $\mathcal{F}_{\text{leak}}$ defines the information leakage during protocol execution resulting from corruption.

This structure is both expressive and minimal: each subroutine isolates a specific trust or performance dimension (e.g., execution validity, fault tolerance, or availability) that varies across L2 designs. By keeping the abstraction modular and interaction-driven, $\mathcal{F}_{\text{layer2}}$ enables rigorous reasoning without overfitting to any particular protocol type.

¹The iUC framework extends Canetti’s UC model [11] to support stateful, multi-session protocols. Prior work argues that iUC is more suitable for modeling complex systems like blockchain protocols than the standard UC [19]. We provide a brief overview in Appendix A.

3.1 Description of $\mathcal{F}_{\text{layer2}}$ and subroutines

At its core, the functionality $\mathcal{F}_{\text{layer2}}^\Lambda$ defines the interface machine that implements different roles and connects to the external environment \mathcal{E} , specifying how the protocol responds to the corresponding outputs to the higher-level protocols based on the parameter subroutines. In real-world implementations, this interface with different roles can be understood as the participants of a L2 protocol, and subroutines capture the underlying scheme used by the protocol. In this paper, we focus on defining the *client* role, but the description of other different roles can also be added if needed. The core logic of the client interface machine of ideal functionality $\mathcal{F}_{\text{layer2}}$ is shown in Figure 2.

Description of $\mathcal{M}_{\text{client}}$ of functionality $\mathcal{F}_{\text{layer2}}$
<p>Implement roles: Layer 2 protocol client</p> <p>Internal state :</p> <ul style="list-style-type: none"> • Round • RequestQueue • ExecutedRequest • StateList • LastReadPointer • OnchainState • Identities <p>Corrupt behavior:</p> <p>Receive corruption request for $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{role})$:</p> <ol style="list-style-type: none"> (1) Send $(\text{Corrupt}, \text{pid}_{\text{cur}}, \text{side}, \text{Internal state})$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{leak}} : \text{leak})$; (2) Wait for $(\text{Corrupt}, \text{leak})$ from $\mathcal{F}_{\text{leak}}$; (3) Reply leak through NET; <p>Main:</p> <p>Receive $(\text{Submit}, \text{request})$ from I/O:</p> <ol style="list-style-type: none"> (1) Send $\{\text{Submit}, \text{request}, \text{Internal state}\}$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{submit}} : \text{submit})$; (2) Wait for $\{\text{Submit}, \text{response}, \text{leak}\}$ from $\mathcal{F}_{\text{submit}}$ s.t. $\text{response} \in \{\text{True}, \text{False}\}$; (3) If $\text{response} = \text{True}$, add request to RequestQueue. Send $\{\text{Submit}, \text{leak}\}$ to \mathcal{S} through NET;
<p>Receive $\{\text{Open}, \text{Initialstate}, \text{Attachment}\}$ from NET:</p> <ol style="list-style-type: none"> (1) Send $\{\text{Open}, \text{Initialstate}, \text{Attachment}, \text{Internal state}\}$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{open}} : \text{open})$; (2) Wait for the output $\{\text{Open}, \text{response}, \text{leak}\}$ from $\mathcal{F}_{\text{open}}$ s.t. $\text{response} \in \{\text{True}, \text{False}\}$; (3) If $\text{response} = \text{True}$, then update Internal state according to $\{\text{Initialstate}, \text{Attachment}\}$, and reply $\{\text{Open}, \text{Initialstate}, \text{Attachment}\}$ to corresponding parties in Identities via I/O; (4) Send $\{\text{Open}, \text{leak}\}$ through NET;
<p>Receive $\{\text{Update}, \text{NewState}, \text{Attachment}\}$ from NET:</p> <ol style="list-style-type: none"> (1) Send $\{\text{Update}, \text{NewState}, \text{Internal state}\}$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{update}} : \text{update})$; (2) Wait for $\{\text{Update}, \text{response}, \text{NewState}, \text{leak}\}$ from $\mathcal{F}_{\text{update}}$; (3) If $\text{response} = \text{True}$, update the Internal state accordingly with NewState, and send leak through NET;
<p>Receive $\{\text{Read}\}$ from I/O:</p>

<ol style="list-style-type: none"> (1) Send $\{\text{Read Internal state}\}$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{read}} : \text{read})$; (2) Wait for ReadResult from $\mathcal{F}_{\text{read}}$, if ReadResult is not empty, reply $\{\text{Read}, \text{ReadResult}\}$ to I/O; (3) Update the LastReadPointer accordingly.
<p>Receive $\{\text{Settlement}, \text{Attachment}\}$ from NET:</p> <ol style="list-style-type: none"> (1) Send $\{\text{Settlement}, \text{Attachment}, \text{Internal state}\}$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{settlement}} : \text{settlement})$; (2) Wait for $\{\text{Settlement}, \text{response}, \text{leak}\}$ from $\mathcal{F}_{\text{settlement}}$ s.t. $\text{response} \in \{\text{True}, \text{False}\}$; (3) If $\text{response} = \text{True}$, reply $\{\text{Settlement}, \text{Success}, \text{LatestState}\}$ to I/O;
<p>Receive $\{\text{UpdateRound}\}$ from NET or I/O:</p> <ol style="list-style-type: none"> (1) Send $\{\text{UpdateRound}, \text{internal state}\}$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{updRnd}} : \text{updRnd})$; (2) Wait for $\{\text{response}, \text{leak}\}$ from $\mathcal{F}_{\text{updRnd}}$ s.t. $\text{response} \in \{\text{True}, \text{False}\}$; (3) If $\text{response} = \text{True}$, $\text{Round} = \text{Round} + 1$, reply to I/O; (4) Send $\{\text{UpdateRound}, \text{leak}\}$ to NET;

Figure 2: The ideal functionality $\mathcal{F}_{\text{layer2}}$'s main logic for handling the requests. pid_{cur} is the current party and sid_{cur} the L2 protocol's current session.

During the execution of $\mathcal{F}_{\text{layer2}}$, there may exist multiple instances of the ideal functionality, each representing a distinct L2 protocol running concurrently and uniquely identified by a session ID (*sid*). In this paper, we focus on a single instance. Within a session, the functionality supports an unbounded number of parties, each identified by a unique party ID (*pid*), who may issue requests. A party is either honest or corrupted and once corrupted, certain information may be leaked. Security guarantees apply only to honest parties. The entities in subroutine machines are assumed to be incorruptible. In what follows, we describe the entire lifecycle of a L2 protocol from the perspective of an honest client party to show how subroutines work.

Input request communication. The higher-level protocol may instruct an honest party to propose multiple different types of execution requests during the time this party participates in the protocol, and three types of requests are accepted: (1) Protocol opening (joining) request, (2) State transition request, and (3) State settlement request.

Upon receiving this request, $\mathcal{F}_{\text{layer2}}$ forwards it to the subroutine $\mathcal{F}_{\text{submit}}$. The $\mathcal{F}_{\text{submit}}$ subroutine verifies whether the submitted request is *semantically valid*, ensuring it is correctly formatted as one of the three types of requests. Based on these checks, $\mathcal{F}_{\text{submit}}$ outputs a boolean value indicating whether the submission is successful, along with the corresponding information leak to the adversary, as determined by $\mathcal{F}_{\text{leak}}$. Finally, $\mathcal{F}_{\text{layer2}}$ updates the RequestQueue based on the boolean response and sends the request and leak through a network connection.

Protocol opening (joining). To open or join the protocol, there should be enough agreements on the initial state from participants

and commit the according initial state on the L1 ledger. In our framework, we use the parameter subroutine $\mathcal{F}_{\text{open}}$ to check whether these requirements for opening a protocol are done. If all the checks pass, the `StateList`, `OnchainState`, `ExecutedRequest` as well as the `Identities` in the `Internal` state of the $\mathcal{F}_{\text{layer2}}$ machine, representing the global state of the protocol, will be updated to the initial state and notify all the participants that the protocol is opened. And the corresponding request for opening recorded in `RequestQueue` will be deleted.

Since the adversary can significantly influence the open procedure by either ignoring requests or tampering with message delivery, we capture this behavior by allowing the simulator to trigger the checking procedure. Specifically, instead of letting the subroutine perform the check immediately upon receiving a request from $\mathcal{F}_{\text{layer2}}$, the simulator explicitly sends a request to $\mathcal{F}_{\text{layer2}}$ through the network connection `NET` to initiate the check. The request includes the `InitialState` and `Attachments`, which contain additional information required for subroutine checks and for generating the final output—such as signatures indicating agreement or the transaction needs to be committed on blockchain. Note that the subroutine like $\mathcal{F}_{\text{open}}$ does not verify the cryptographic validity of the signatures; it only checks structural correctness, such as whether the required number of signatures is present. Only after the protocol is considered as opened will the following steps proceed, such as state update and state settlement, proceed.

Protocol state update. $\mathcal{F}_{\text{layer2}}$ will also receive the update request from the network connection to update the participants' states. This especially captures protocols like sidechain, where the state update could be carried out by the participants corrupted by the adversary. Additionally, even for the protocols in which the state update is started by the honest party, the adversary can still be influenced by corrupted participants' non-responding or delaying the message delivery among honest participants. Upon receiving the request for state update, $\mathcal{F}_{\text{layer2}}$ forwards it to the $\mathcal{F}_{\text{update}}$ subroutine, along with the current `Internal` state it maintains. $\mathcal{F}_{\text{layer2}}$ then waits for a response from $\mathcal{F}_{\text{update}}$, which includes the updated state after execution and the corresponding information leak to the adversary. At the end of the procedure, $\mathcal{F}_{\text{layer2}}$ will update the `StateList` according to the update result. And `RequestQueue` is also updated to represent the requests submitted before it is executed, and the evidence for execution is recorded in `ExecutedRequest`. By customizing the checks in $\mathcal{F}_{\text{update}}$, our framework can accommodate a wide range of mechanisms used by different L2 protocols, such as consensus-based agreement in sidechain protocols or layer 1 interactions in rollup protocols.

L2 state settlement. As the mark of ending the L2 protocol, a higher-level protocol may instruct to close or exit the L2 protocol, which requires the final settlement of the L2 state on the L1 ledger. After the settlement request is sent through $\mathcal{F}_{\text{submit}}$, the functionality waits for the simulator to trigger the settlement subroutine $\mathcal{F}_{\text{settlement}}$ to decide whether the settlement is successful. The $\mathcal{F}_{\text{settlement}}$ access the L1 ledger's functionality, $\mathcal{F}_{\text{ledger}}$, to verify the on-chain state published by participants whether is the correct one according to `Internal` state.

Read L2 information. A L2 protocol should support read access for clients to necessary L2 information, such as participants'

states and the executed requests that are related to the state transition. Upon receiving a read request, $\mathcal{F}_{\text{layer2}}$ forwards it to the $\mathcal{F}_{\text{read}}$ subroutine, along with the identity of the requester as well as the `Internal` state. $\mathcal{F}_{\text{read}}$ then determines which information the requester is permitted to access based on the underlying schemed use by the protocol and the possible influence of the adversary.

L2 clock simulation. Time-related assumptions are common in L2 protocols. Our framework introduces $\mathcal{F}_{\text{updRnd}}$ to simulate an internal clock for L2 protocols to describe such assumptions like message delivery among protocol participants.

Information leakage. For the complementary requirements of the framework, we introduce the subroutine $\mathcal{F}_{\text{leak}}$ to capture the information leaked to the adversary through corruption. This subroutine can be formally instantiated when analyzing protocols that aim to achieve certain privacy properties. However, such privacy guarantees typically depend on specific cryptographic primitives employed by the protocol and our focus is to capture differences in functional logic across L2 protocol designs. Therefore, in the remainder of this paper, we assume that protocols do not involve any secret inputs and that all requests are received from the environment. Under this assumption, $\mathcal{F}_{\text{leak}}$ leaks all information like request plaintext to the simulator and adversary.

3.2 Security Properties

After introducing the components of our framework $\mathcal{F}_{\text{layer2}}$, we then formally define the security properties that a L2 protocol should realize with the parameter subroutines of our framework.

Correct initialization. A fundamental step for the subsequent execution of the L2 protocol is the correct initialization of the state after a client requests to join the protocol. In particular, the check within the subroutine $\mathcal{F}_{\text{open}}$ should guarantee two main perspectives: (1) all the involved honest clients should agree on the initial state; (2) there should be a corresponding state value committed on the blockchain (L1). Assuming that the environment issues a request for an honest participant p_h open or join the protocol with an initial state s_{int} , we define the following security property to capture the guarantees required during the open procedure formally:

Definition 3.1 (Correct Initialization). A L2 protocol Λ built on a secure L1 ledger $\mathcal{F}_{\text{ledger}}$ realizes *correct initialization* if its corresponding ideal functionality $\mathcal{F}_{\text{layer2}}^\Lambda$'s interface $\mathcal{F}_{\text{layer2}}$ outputs $\{\text{Open}, \text{True}, s_{\text{int}}\}$ to an honest participant p_h only when:

- (1) s_{int} was previously requested by an honest participant p_h .
- (2) s_{int} is committed as a valid state in the output of $\mathcal{F}_{\text{ledger}}$ through a `Read` request.

f -safety. Due to the involvement of multiple participants, a critical security property during the update procedure is *safety*, which ensures the consistency of the executed request order among all honest participants and guarantees execution correctness. We further introduce the parameter f to represent the maximum number of deviating participants that the protocol can tolerate while still maintaining *safety*. In our framework, f -*safety* is captured by the subroutines $\mathcal{F}_{\text{update}}$ and $\mathcal{F}_{\text{read}}$. Although $\mathcal{F}_{\text{update}}$ does not directly produce outputs to the environment, its modifications to the `Internal` state influence the subsequent outputs of $\mathcal{F}_{\text{read}}$. To formally define this property, we use the notation \mathcal{R}_f^P to denote the

read output about the executed requests from $\mathcal{F}_{\text{layer2}}^\Lambda$, as instructed by the subroutine $\mathcal{F}_{\text{read}}^\Lambda$ at round r for participant P , and define f -safety as follows:

Definition 3.2 (f -safety). A L2 protocol Λ built on a secure L1 ledger $\mathcal{F}_{\text{ledger}}$ satisfies f -safety if the executed requests in read output of $\mathcal{F}_{\text{read}}^\Lambda$ at round r satisfies:

- **(Self-consistency)** For any honest participant P and any rounds $r_1 \leq r_2$, it holds that $\mathcal{R}_{r_1}^P \leq \mathcal{R}_{r_2}^P$.
- **(View-consistency)** For any honest participants P_1, P_2 , and any round r , it holds that $\mathcal{R}_r^{P_1} \leq \mathcal{R}_r^{P_2}$ or $\mathcal{R}_r^{P_2} \leq \mathcal{R}_r^{P_1}$.

$\{f, T\}$ -liveness. Note that f -safety guarantees consistency among all participants regardless of the content, meaning that even a protocol incapable of executing any request would still achieve *safety*. However, such a protocol would be functionally useless. To capture the requirement for system *liveness*, we formally define another critical security property, $\{f, T\}$ -liveness, which characterizes both the corruption tolerance f and the maximum time latency T for an executed request Q proposed by an honest client to be executed and accessible by all honest participants eventually. In L2 protocols, multiple types of input requests exist. In this paper, we focus on the liveness of three types of requests: (1) Protocol opening (joining) requests, (2) State update requests, and (3) Settlement requests. Since different requests can exhibit different liveness characteristics in a L2 protocol, we define a protocol as satisfying f, T -liveness based on the read result as follows:

Definition 3.3 ($\{f, T\}$ -liveness for request). An L2 protocol Λ built on a secure L1 ledger $\mathcal{F}_{\text{ledger}}$ satisfies $\{f, T\}$ -liveness if the ideal functionality $\mathcal{F}_{\text{layer2}}^\Lambda$ that it realizes ensures that for any valid request Q sent to its interface $\mathcal{F}_{\text{layer2}}$, the request receives a reply from the interface or is included in the output of $\mathcal{F}_{\text{read}}^\Lambda$ at some time $t' \geq t + T$ for all honest participants, provided that no more than f participants are corrupted by the adversary.

Correct settlement. As the final step of an L2 protocol, *correct settlement* must be ensured to guarantee that either an agreed-upon result or the final execution result of the L2 protocol is eventually committed to the L1 ledger. In our framework, the completion of the settlement procedure is marked by the output message {Settlement, True} from the interface $\mathcal{F}_{\text{layer2}}$, which is instructed by the subroutine $\mathcal{F}_{\text{settlement}}$. In most cases, this subroutine verifies that the latest state in `StateList`, as provided by the caller interface, has been committed to the L1 ledger. Some protocols also permit settlement for a state based on an agreement between participants, a topic that will be discussed further in the next section. The definition of this property is as follows:

Definition 3.4 (correct settlement). An L2 protocol Λ , built on a secure L1 ledger $\mathcal{F}_{\text{ledger}}$, satisfies *correct settlement* if $\mathcal{F}_{\text{settlement}}^\Lambda$ outputs True to an honest participant p_h only when:

- (1) $S_n^{p_h}$ is the latest state for the participant included in `StateList` or is explicitly included in the settlement request submitted p_h .
- (2) $S_n^{p_h}$ is committed as a valid state in the output of $\mathcal{F}_{\text{ledger}}$ through a Read request.

After formally defining the security properties, we can propose the definition for a secure L2 protocol:

Definition 3.5. An L2 protocol Λ is secure if and only if its real-world protocol \mathcal{P}^Λ realizes the ideal functionality $\mathcal{F}_{\text{layer2}}^\Lambda = (\mathcal{F}_{\text{layer2}} | \mathcal{F}_{\text{open}}^\Lambda, \mathcal{F}_{\text{submit}}^\Lambda, \mathcal{F}_{\text{update}}^\Lambda, \mathcal{F}_{\text{read}}^\Lambda, \mathcal{F}_{\text{updRnd}}^\Lambda, \mathcal{F}_{\text{settlement}}^\Lambda, \mathcal{F}_{\text{leak}}^\Lambda)$ satisfying *correct initialization, safety, liveness and correct settlement*.

3.3 Efficiency Property

Although secure L2 protocols achieve the required security properties, the employed different underlying schemes will lead to variations in efficiency performance. We focus on (G_{L_2}, G_{L_1}) -*data availability*, a widely discussed efficiency metric that reflects the lower bound of L2 and L1 storage cost to ensure security.

Let $f_{st}(S, E) = S'$ denote the deterministic state transition function used by an L2 protocol, where S represents the previous valid state, and E contains the executed requests for the state transition. The property *data availability* captures the *storage lower bound* of an L2 protocol, which also affects the outputs of $\mathcal{F}_{\text{read}}^\Lambda$. Specifically, under *data availability* constraints, there are three types of data that should be output by $\mathcal{F}_{\text{read}}^\Lambda$: (1) The initial state S_{L_1} ; (2) The current state S' at round r ; (3) The executed requests and evidence E required to reconstruct the state transition $f_{st}(S_{L_1}, E) = S'$. We define the property formally as follows:

Definition 3.6 ((G_{L_2}, G_{L_1}) -Data Availability). A Layer2 protocol Λ , built on a secure Layer1 blockchain $\mathcal{F}_{\text{ledger}}$, realizes (G_{L_2}, G_{L_1}) -*data availability* if, for any honest participant p_h at any round r , the data set $\{E, S_{L_1}, S'\}$ is accessible via $\mathcal{F}_{\text{read}}^\Lambda$ and satisfies $f_{st}(S_{L_1}, E) = S'$. Furthermore, the storage lower bounds on Layer2 and Layer1 are G_{L_2} and G_{L_1} , respectively.

4 Case Studies in L2 Protocols

After defining secure Layer 2 (L2) protocols and the corresponding security properties, we now demonstrate how our framework captures the three primary classes of L2 scaling approaches: payment channels (and channel factories), sidechains, and rollups. We instantiate our framework with one representative protocol from each category—Brick, Liquid, and Arbitrum Nitro—omitting privacy considerations for clarity. Full formalizations and proofs are provided in Appendices B to D due to space constraints.

4.1 Case Study: The Brick Channel

4.1.1 The Real Brick Protocol $\mathcal{P}^{\text{Brick}}$. The Brick channel [5] is the first two-party payment channel designed to operate under an asynchronous communication assumption. The protocol involves two types of participants:

- **Clients (Operators):** Two parties conduct transactions with each other through the payment channel. They serve as both the clients and operators of the protocol. In the analysis of Brick, we use these two designations interchangeably. And at least one of them is assumed to be honest.
- **Third-party wardens:** A group of third-party participants who verify and store state updates from the clients and assist in settling the channel state on the L1 ledger unilaterally if needed. Brick assumes more than $\frac{2}{3}$ wardens are honest.

The general procedure of the Brick channel operates as follows:

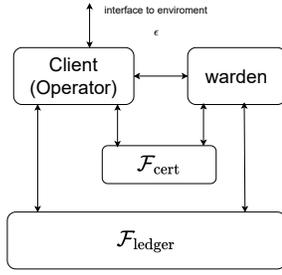


Figure 3: The Brick payment channel protocol

Channel Opening. The two clients communicate to agree on the initial state and select the wardens. Once the wardens are notified, all participants deposit collateral on the L1 ledger. After verifying that the collateral has been committed on the L1 ledger, the channel is considered open.

Channel Update. After the channel is opened, the two clients can update the channel state in two steps: (1) The two clients exchange the new state value and sign the state. (2) Once both signatures are collected, the signed new state is sent to all wardens. A client considers the newly updated state valid once at least $\frac{2}{3}$ of the wardens' signatures have been received.

Channel State Settlement. In the Brick channel, an client can settle the channel state on the L1 ledger through two methods:

- **Collaborative Settlement:** A client first sends a settlement request to the counterparty along with the proposed settlement state. If the counterparty agrees, they sign the collaborative settlement request. The settlement transaction, including the settlement state and both participants' signatures, is then published on the L1 ledger. Once confirmed, the channel state is considered settled, and the channel is closed.
- **Unilateral Settlement:** Due to the asynchronous communication network assumption, the Brick channel allows a client to settle the state unilaterally with the help of wardens. To do so, the client instructs the wardens to publish the stored latest state on the L1 ledger. After at least $\frac{2}{3}$ of the wardens have published their stored values, the channel will be closed with the state with the highest state sequence number.

Then we can define the real-world Brick payment channel protocol as $\mathcal{P}^{\text{Brick}} = (\text{Client}|\text{Warden}, \mathcal{F}_{\text{cert}})$, where Client represents the machine with the code for both the client's behavior in the protocol, where could exist multiple instances representing different clients in the real protocol. The Client machine is also connected to the external environment to receive instruction requests; Warden represents the machine with the code for the warden's behavior, but it does not receive requests from the environment, only the requests from the Client. The connection among different machines, along with the underlying ideal L1 blockchain functionality, can be shown in Figure 3.

4.1.2 The Ideal Functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ After showing the real-world protocol of the Brick channel, we then propose the formal definition for the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}} = (\mathcal{F}_{\text{layer2}}|\mathcal{F}_{\text{submit}}^{\text{Brick}}, \mathcal{F}_{\text{open}}^{\text{Brick}}, \mathcal{F}_{\text{update}}^{\text{Brick}}$,

$\mathcal{F}_{\text{read}}^{\text{Brick}}, \mathcal{F}_{\text{settlement}}^{\text{Brick}}, \mathcal{F}_{\text{updRnd}}^{\text{Brick}}, \mathcal{F}_{\text{leak}}^{\text{Brick}}$). While the interface $\mathcal{F}_{\text{layer2}}$ remains the same as discussed before, in the following, we will show how the subroutines are specified in order to capture the security properties. Noted the entities subroutine machine instance can not be corrupted.

$\mathcal{F}_{\text{submit}}^{\text{Brick}}$ checks whether the received request belongs to one of the following three types of requests: (i) The open request to initiate the payment channel. (ii) The transaction requests to update the channel state. (iii) The state settlement request. $\mathcal{F}_{\text{submit}}^{\text{Brick}}$ verifies that incoming requests are semantically valid. Besides, no state update request will be accepted before the open request is executed and after the settlement request is executed.

$\mathcal{F}_{\text{open}}^{\text{Brick}}$ checks that the channel open request proposed by the simulator \mathcal{S} that it is agreed by honest clients by checking the RequestQueue modified by $\mathcal{F}_{\text{submit}}^{\text{Brick}}$. Besides, $\mathcal{F}_{\text{open}}^{\text{Brick}}$ also checks the proposed transaction, along with all clients' and wardens' collateral, are committed on the L1 ledger by interacting with $\mathcal{F}_{\text{ledger}}$.

$\mathcal{F}_{\text{update}}^{\text{Brick}}$ verifies that any newly proposed state update from the simulator \mathcal{S} includes agreements from all honest clients and at least $\frac{2}{3}$ of the wardens. Notably, $\mathcal{F}_{\text{update}}^{\text{Brick}}$ does not validate the correctness of signatures; rather, it checks whether the received state update proposal is formally well-structured and contains the required components.

When receiving a read request, $\mathcal{F}_{\text{read}}^{\text{Brick}}$ first checks with the adversary \mathcal{A} through the NET connection. Due to the asynchronous communication network assumption, the adversary \mathcal{A} can only influence the read result by delaying message delivery. As a result, the read output will either contain the latest state or the previous state. Additionally, all intermediate states from the initial state onward, along with their corresponding certificates, must also be returned to the read requester.

$\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ verifies two types of settlement proposals triggered by the simulator \mathcal{S} . For **collaborative settlement**, $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ checks that all honest clients agree on the settlement state and that the settlement transaction has been successfully committed on the L1 ledger. For **unilateral settlement**, $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ verifies that at least $\frac{2}{3}$ of the wardens have published their stored state on the L1 ledger and ensures that only the latest state is committed. Once these checks pass, all participants are notified via the output of the interface. By enforcing these conditions, $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ guarantees *correct settlement* for both types of settlement procedures.

Since the Brick channel operates under an asynchronous communication assumption, the internal clock subroutine $\mathcal{F}_{\text{updRnd}}^{\text{Brick}}$ does not impose additional checks on requests to update the round.

With the subroutines above, we can have the following conclusion for the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$:

THEOREM 4.1. *The ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ guarantees all the security properties of a secure L2 protocol.*

The security of the real Brick payment channel protocol can then be demonstrated by proving that the real protocol iUC realizes the ideal functionality. This can be formally defined as follows:

THEOREM 4.2. *Let $\mathcal{F}_{\text{ledger}}$ be the idealized L1 ledger functionality and $\mathcal{F}_{\text{cert}}$ be the idealized functionality for EUF-CMA secure signature*

scheme. Then, the real Brick payment channel protocol \mathcal{P}^{Brick} realizes the ideal Brick payment channel functionality $\mathcal{F}_{layer2}^{Brick}$.

4.2 Case Study: The Liquid Network Sidechain

4.2.1 *The Real Liquid Network Protocol \mathcal{P}^{Liquid}* . The Liquid Network [25] is a sidechain built on top of Bitcoin. The protocol involves the following participants:

- **Operators:** In the original protocol, there are two distinct roles: block signers and watchmen. Block signers are responsible for generating blocks for the sidechain, while watchmen handle the creation of peg-out transactions on the L1 ledger. For simplicity, we assume both roles are combined under the term operator. It is assumed more than $\frac{2}{3}$ operators are honest.
- **Clients:** Clients are the users of the Liquid Network sidechain. They receive instructions from the environment and submit requests to the protocol.

The original Liquid Network employs a more complex design to support additional functional properties, such as privacy preservation. However, to capture the core secure functional logic of the Liquid Network, we abstract and describe the general procedure of the sidechain from the perspective of a client as follows:

Sidechain Joining. In order to join the sidechain, the client must first send collateral coins to a publicly known deposit address controlled by the operator federation on the L1 ledger. Once the transaction has been confirmed for a sufficient period (100 blocks), the client generates and submits a peg-in transaction to the operators. This peg-in transaction includes proof of the existence of the corresponding deposit transaction on the L1 ledger. Once the peg-in transaction is successfully recorded on the sidechain, the client is considered to have joined the sidechain.

Sidechain Update. A client's state is updated based on transactions occurring within the sidechain. To initiate a state update, the client generates and submits a transaction to the operators. The operators execute a two-phase Byzantine Fault Tolerant (BFT) consensus protocol among themselves. Here, we assume that a block is considered valid if it receives a quorum certificate, meaning that more than $\frac{2}{3}$ of all operators agree on its validity with signature. Once the transaction is included in a block proposed by the operators and the quorum certificate is obtained, the transaction is considered executed, and the client's state is updated accordingly.

Sidechain Leaving. To leave the sidechain protocol, the client prepares a peg-out transaction and submits it to the operators. Once the transaction is recorded on the sidechain, the operators publish a corresponding transaction on the L1 ledger to transfer the appropriate amount of coins to the client, completing the exit process.

Based on the protocol description above, we can define the real-world protocol as $\mathcal{P}^{Liquid} = (\text{Client}|\text{Operator}, \mathcal{F}_{com}, \mathcal{F}_{cert})$, where Client represents the interaction interface with the environment and processes requests to the L2 protocol. Operator includes the machine code that implements the core logic for reaching consensus on new blocks and issuing peg-out transactions on the L1 ledger. The structure of the real protocol is shown in Figure 4.

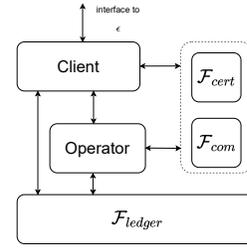


Figure 4: The Liquid Network sidechain protocol

4.2.2 *The Ideal Functionality $\mathcal{F}_{layer2}^{Liquid}$* . After showing the real-world protocol of the Liquid Network sidechain, we then propose the formal definition for the ideal functionality $\mathcal{F}_{layer2}^{Liquid} = (\mathcal{F}_{layer2}^{Liquid} | \mathcal{F}_{submit}^{Liquid}, \mathcal{F}_{open}^{Liquid}, \mathcal{F}_{update}^{Liquid}, \mathcal{F}_{read}^{Liquid}, \mathcal{F}_{settlement}^{Liquid}, \mathcal{F}_{upRnd}^{Liquid}, \mathcal{F}_{leak}^{Liquid})$. While the interface \mathcal{F}_{layer2} remains the same, in the following, we will show how the subroutines are specified in order to capture the security properties. Noted the entities subroutine machine instance can not be corrupted.

$\mathcal{F}_{submit}^{Liquid}$ checks the received requests belonging to one of the following: (i) the request from the client to join the Liquid Network sidechain, (ii) the transaction request to update the sidechain state, and (iii) the request from the client to leave the sidechain protocol. $\mathcal{F}_{submit}^{Liquid}$ verifies that incoming requests are semantically valid. In Case (i), the request must include proof for the peg-in transaction on the L1 ledger. In Case (ii), the transaction must be signed by the initiator. In Case (iii), the request must include the necessary proofs for its identity. While the content validity of the request is not checked at this stage, the subroutine ensures that the requests are semantically well-formed.

$\mathcal{F}_{open}^{Liquid}$ verifies that a sidechain joining request, triggered by the simulator \mathcal{S} , satisfies two conditions. First, a corresponding peg-in transaction must be recorded in the StateList , which represents the current transaction list of the sidechain. Second, a collateral deposit transaction must exist on the L1 ledger. Since a client can only join the sidechain by initiating the deposit itself, and the majority of the Operators are assumed to be honest, the joining process is guaranteed to be known by all participants.

$\mathcal{F}_{update}^{Liquid}$ verifies that any newly proposed state update from the or the simulator \mathcal{S} , which is a newly formed block. Notably, $\mathcal{F}_{update}^{Liquid}$ checks there are agreements from more than $\frac{2}{3}$ of all the operators.

In the Liquid Network sidechain, the communication among participants is assumed to be synchronous, and a client is guaranteed to be connected to at least one honest operator under the majority honest assumption. Thus, after receiving the read request, the $\mathcal{F}_{read}^{Liquid}$ simply replies to all the transaction lists, as well as the state value recorded in StateList . Thus $\mathcal{F}_{read}^{Liquid}$ guarantees the *data availability* property.

$\mathcal{F}_{settlement}^{Liquid}$ verifies the settlement proposals triggered by the adversary \mathcal{A} based on two checks: (1) there exists an according settlement request recorded in the ExecutedRequest ; (2) there is also and transferring transaction committed on the L1 ledger through the checking with underlying \mathcal{F}_{ledger} . Under the assumption of

honest majority operator and the secure L1 ledger, $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$ guarantees *correct settlement*.

Since the Liquid Network sidechain assumes the communication within L2 participants is synchronous, the subroutine $\mathcal{F}_{\text{updRnd}}^{\text{Liquid}}$ will not agree the round proceed if there exist request in RequestQueue last for more than δ time.

With the subroutines above, we can have the following result:

THEOREM 4.3. *The ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ guarantees all the security properties of a secure L2 protocol.*

The security of the Liquid Network sidechain protocol can be demonstrated by proving that the real protocol iUC realizes the ideal functionality. This can be formally defined as follows:

THEOREM 4.4. *Let $\mathcal{F}_{\text{ledger}}$ be the idealized L1 ledger functionality, \mathcal{F}_{com} be the synchronous communication channel, $\mathcal{F}_{\text{cert}}$ be the idealized functionality for EUF-CMA secure signature scheme. Then, the real Liquid Network sidechain protocol $\mathcal{P}^{\text{Liquid iUC}}$ realizes the ideal Liquid Network sidechain functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$.*

4.3 Case Study: The Arbitrum Nitro Rollup

4.3.1 The Real Arbitrum Nitro Protocol $\mathcal{P}^{\text{Arbitrum}}$. The Arbitrum Nitro [8] is an optimistic protocol. While the real-world implementation includes more complex participant roles and mechanisms to support a broader range of functionalities, here we provide a simplified discussion focusing on the core aspects of the protocol related to scaling L1 execution. To achieve scalability, the protocol primarily involves two types of participants:

- **Operators:** In the Arbitrum Nitro rollup, operators are referred to as sequencers or managers. There may be multiple Operators, but unlike sidechain protocols, these Operators do not need to run a consensus protocol among themselves. And there should be at least one honest operator. Note that in the original protocol, Arbitrum also allows clients to submit requests directly to the L1 ledger to prevent censorship. In such cases, the client effectively acts as an honest operator.
- **Validators:** Validators are responsible for verifying the new rollup state published by the operators. There should be at least one honest validator to verify publication.
- **Clients:** Clients are the users of the rollup protocol who submit transactions to the Operators for execution.

The core logic of the Arbitrum Nitro protocol then operates as follows:

Rollup joining: A client joins the rollup by first depositing coins on the L1 ledger. Then, the client submits a joining transaction to the sequencer via the rollup network. Once the client observes that the joining transaction has been committed to the rollup protocol, the client is considered successfully joined.

Rollup update: To update the rollup state, the operator first collects and batches valid transactions from clients and then executes them. After execution, the sequencer publishes the executed transactions and their results to the L1 ledger. Noted the L1 ledger maintainer does not need to execute the published transactions.

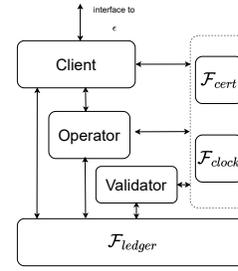


Figure 5: The Arbitrum protocol

Once published, validators can re-execute and verify the state update's correctness based on the published transactions. If the published result is incorrect, fraud-proof can be submitted to invalidate the incorrect update. If no fraud proof is submitted within the designated challenge period, the state is considered valid. If a client notices its transaction has not been executed for a time period, it can also publish it to the L1 ledger by itself.

Rollup exit: A client exits the rollup by first submitting a leaving transaction to the Operator. Once the transaction is successfully executed, the client can then publish a transfer transaction on the L1 ledger, marking the official exit from the protocol.

Based on the protocol description above, we can define the real-world protocol as $\mathcal{P}^{\text{Arbitrum}} = (\text{Client}|\text{Operator}, \text{Validator}, \mathcal{F}_{\text{cert}}, \mathcal{F}_{\text{clock}})$, where Client represents the interaction interface with the environment and processes requests to the rollup protocol. We use $\mathcal{F}_{\text{clock}}$ to model the periodic behavior of operators and validators, including the publishing of state updates on the L1 ledger by operators and the subsequent verification of these updates by validators. Operator includes the machine code that implements the core logic for the sequencer. The structure of the real protocol is shown in Figure 5.

4.3.2 The Ideal Functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$. After showing the real-world protocol of the Arbitrum Nitro rollup, we then propose the formal definition for the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}} = (\mathcal{F}_{\text{layer2}}|\mathcal{F}_{\text{submit}}^{\text{Arbitrum}}, \mathcal{F}_{\text{open}}^{\text{Arbitrum}}, \mathcal{F}_{\text{update}}^{\text{Arbitrum}}, \mathcal{F}_{\text{read}}^{\text{Arbitrum}}, \mathcal{F}_{\text{settlement}}^{\text{Arbitrum}}, \mathcal{F}_{\text{updRnd}}^{\text{Arbitrum}}, \mathcal{F}_{\text{leak}}^{\text{Arbitrum}})$. While the interface $\mathcal{F}_{\text{layer2}}$ remains the same, in the following, we will show how the subroutines are specified in order to capture the security properties. Noted the entities subroutine machine instance can not be corrupted.

$\mathcal{F}_{\text{submit}}^{\text{Arbitrum}}$ checks whether the received requests are one of the following types: (i) the request from the client to join the Arbitrum Nitro rollup, (ii) the transaction request to update the rollup protocol state, and (iii) the request from the client to leave the rollup protocol. $\mathcal{F}_{\text{submit}}^{\text{Liquid}}$ verifies that incoming requests are semantically valid, including all the necessary information for checking. While the content validity of the request is not checked at this stage, the subroutine ensures that the requests are semantically well-formed.

$\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$ verifies that rollup joining request, triggered by the simulator \mathcal{S} , satisfies two conditions. First, a corresponding peg-in transaction must be recorded in the L1 ledger $\mathcal{F}_{\text{ledger}}$, which is published by the operator. Second, a collateral deposit transaction must exist on the L1 ledger published by the joining client.

$\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ verifies any newly proposed state update from the simulator \mathcal{S} by checking the underlying L1 ledger, which consists of a new result and a batch of executed transactions. The verification process follows two possible scenarios: (1) If the request batch is published on the blockchain and lasts a time period T_{period} without any fraud-proof, $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ outputs True, confirming the validity of the update; (2) If any issue is detected in the batch, $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ must check whether a fraud-proof transaction is published before the period T_{period} ends. If valid fraud-proof is provided, the incorrect state update is invalidated, and subroutine outputs False. However, if no fraud proof is submitted within the given timeframe, the functionality halts.

In the Arbitrum Nitro rollup, $\mathcal{F}_{\text{read}}^{\text{Arbitrum}}$ does not determine the read result based on the Internal state maintained by the interface. Instead, the read result is directly obtained by sending a read request to the underlying $\mathcal{F}_{\text{ledger}}$, ensuring that the state retrieved is consistent with the data stored on the L1 ledger.

$\mathcal{F}_{\text{settlement}}^{\text{Arbitrum}}$ verifies the settlement proposals triggered by the simulator \mathcal{S} based on two checks with the underlying blockchain $\mathcal{F}_{\text{ledger}}$: (1) the proposed settlement request is executed and committed to the L1 ledger by the client or the operator; (2) there is also a transaction transferring coins to the leaving client according to the latest state committed on the L1 ledger.

Since the Arbitrum Nitro rollup protocol requires the operator to publish state and request batch periodically, the subroutine $\mathcal{F}_{\text{updRnd}}^{\text{Arbitrum}}$ will not agree the round proceed if there no update more than T_{period} time. These subroutines yield the following result:

THEOREM 4.5. *The ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ guarantees all the security properties of a secure L2 protocol.*

The security of the Arbitrum Nitro rollup protocol can be demonstrated by proving that the real protocol iUC realizes the ideal functionality. This can be formally defined as follows:

THEOREM 4.6. *Let $\mathcal{F}_{\text{ledger}}$ be the idealized L1 ledger functionality, $\mathcal{F}_{\text{cert}}$ be the idealized functionality for EUF-CMA secure signature scheme. Then, the real Liquid Network sidechain protocol $\mathcal{P}^{\text{Arbitrum}}$ iUC-realizes the ideal Arbitrum Nitro rollup functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$.*

5 Comparative Analysis of L2 Designs

In this section, we analyze the structural differences and design trade-offs among Layer 2 (L2) protocols as revealed by our framework. By instantiating the framework for representative constructions, we identify the fundamental differences in parameter subroutines that influence protocol behavior and demonstrate how these differences impact the trade-offs on both security and efficiency properties. Omitted proofs can be found in Appendix E.

5.1 Subroutine Comparison

As observed in the case studies above, the core mechanism by which the framework captures the properties of L2 protocols is through triggering subroutines to determine whether the protocol state, represented by the Internal state, should be updated based on protocol-specific requirements. In this section, we analyze how these subroutines define the structural differences among various

types of L2 protocols, thereby revealing the key design choices that differentiate them. Specifically, we focus on five subroutines: $\mathcal{F}_{\text{open}}$, $\mathcal{F}_{\text{submit}}$, $\mathcal{F}_{\text{update}}$, $\mathcal{F}_{\text{read}}$, and $\mathcal{F}_{\text{settlement}}$, as these encapsulate the essential procedures involved in the lifecycle of an L2 protocol.

5.1.1 The Subroutine Content. The general conditions within each subroutine that determine when a positive output is generated differentiate L2 protocols from each other, as illustrated in Table 1. In the following, we describe how these conditions vary across the three types of L2 protocols.

Protocol opening (joining) $\mathcal{F}_{\text{open}}$. In payment channel factories (PCFs), this procedure requires agreement directly from the clients themselves, represented by their requests sent to the protocol and recorded in RequestQueue. In addition, the agreement from all other participants, including potential third-party entities, must be verified through checks on the L1 ledger. In contrast, for sidechain and rollup protocols, the responsibility for agreement verification is delegated solely to the operators and recorded in ExecutedRequest. Consequently, in PCFs, a client’s joining depends on other participants, including both other clients and third-party entities. However, in sidechain and rollup protocols, the decision depends only on the joining client and the operators.

Request communication $\mathcal{F}_{\text{submit}}$. In our analysis, communication is a shared commonality across all types of L2 protocols. To ensure that requests are executed timely, they must be broadcast to all relevant operators. However, assuming the same number of clients n_C are supported, the communication complexity differs across protocol types. In PCFs, each client is also an operator, meaning each request must be sent to all n_C participants. In contrast, in sidechains and rollups, requests only need to be sent to a designated set of n_{OP} operators, where typically $n_{OP} < n_C$. Consequently, when supporting the same number of clients, PCF protocols may incur higher communication complexity compared to sidechain and rollup protocols.

Request execution $\mathcal{F}_{\text{update}}$. The subroutine $\mathcal{F}_{\text{update}}$ defines the requirements for an L2 protocol to order and execute requests. The PCF protocol differs from the sidechain protocol in that PCF requires full agreement from all operators, whereas the sidechain requires agreement only from a subset of operators, as specified by its consensus algorithm. Notably, both protocols can achieve *safety* without relying on the L1 ledger.

In contrast, the rollup protocol fundamentally differs from both PCF and sidechain protocols due to its reliance on the subroutine $\mathcal{F}_{\text{ledger}}$ during execution to guarantee *safety*. Rollups use the L1 ledger to ensure consistency of executed requests among all participants. Without publishing the necessary data or if the *safety* of the L1 ledger is violated, the *safety* of the rollup also fails. Therefore, for rollup protocols, interaction with $\mathcal{F}_{\text{ledger}}$ is a **necessary condition** for achieving *safety*.

L2 protocol data reading $\mathcal{F}_{\text{read}}$. As a side effect of the update procedure, the data reading mechanism for PCF and sidechain protocols also differs from that of rollups. PCFs and sidechains exhibit similar behavior in data reading, as all data is stored off-chain and can be directly accessed within L2. In contrast, rollups rely on the L1 ledger for data storage, which necessitates interaction with the L1 for data reading.

Table 1: Difference in main parameter subroutines. Assume in each type of protocol there are the same number of n_C clients and n_{OP} operators

	\mathcal{F}_{open}	\mathcal{F}_{submit}	\mathcal{F}_{update}	$\mathcal{F}_{settlement}$	\mathcal{F}_{read}
PCF	All participants agree on opening + Check the L1	Send to n_C clients	Agreed by all clients	Enough clients agreements according to settlement type + Check the L1	Read through L2
sidechain	Request executed by operators + Check the L1	Send to n_{OP} operators	Agreed by $\theta(n_{OP})$ operators + Agreed by $\theta(1)$ operators + Publish on the L1	Request Executed by operators + Check the L1	Read through L1
Rollup					

L2 state settlement $\mathcal{F}_{settlement}$. Similar to the protocol opening (joining) procedure, PCF protocols finalize the protocol state based on direct agreement from the clients themselves, whereas, in sidechain and rollup protocols, this responsibility is delegated to the operators. However, unlike the opening procedure, PCF protocols like the Brick channel additionally support unilateral settlement, which requires agreement only from the initiating client, allowing an individual client to settle the state without the cooperation of other participants.

5.1.2 The Subroutine Connection. From the content differences of subroutines, we can further capture some insights about the connection relationship among subroutines for different types of L2 protocols. The subroutine connection differences can be seen in Figure 6.

To start with, the *direct connection*, represented by the I/O connection, is more apparent. The rollup protocol is distinct from the other two types of L2 protocols in that its subroutines \mathcal{F}_{update} and \mathcal{F}_{read} require direct interaction with the underlying L1 ledger \mathcal{F}_{ledger} . Recall that \mathcal{F}_{update} specifies the requirements for generating a new valid state in the protocol. Unlike PCFs or sidechains, the executed requests received agreements from the L2 participants only for consistency, rollups require the executed requests and the new state to be published on the L1 ledger to realize consistency among L2 protocol participants. Consequently, the way participants learn the executed requests and latest state through \mathcal{F}_{read} is by directly fetching the state from \mathcal{F}_{ledger} .

Additionally, as has been noted, subroutine \mathcal{F}_{open} and $\mathcal{F}_{settlement}$ checks RequestQueue or ExecutedRequest, which are internal state variables updated by other subroutines. This creates an indirect dependency between subroutines through shared state.

5.1.3 Classification and Definition. According to the analysis above, the primary factor that differentiates L2 protocols from each other is the behavior and structure of their subroutines. Based on the characteristics of these subroutines, we propose the following definitions for the three primary types of L2 protocols:

Definition 5.1 (PCF). An L2 protocol Λ is a PCF protocol if the ideal functionality $\mathcal{F}_{layer2}^\Lambda$ realized by Λ satisfies the following:

- Subroutine $\mathcal{F}_{open}^\Lambda$ and $\mathcal{F}_{settlement}^\Lambda$ relies on the result of \mathcal{F}_{submit} .
- \mathcal{F}_{update} and \mathcal{F}_{read} does not need checking with \mathcal{F}_{ledger} to guarantee *safety*.
- \mathcal{F}_{update} requires agreement from all operators.

Definition 5.2. An L2 protocol Λ is a sidechain protocol if the ideal functionality $\mathcal{F}_{layer2}^\Lambda$ realized by Λ satisfies the following:

- Subroutine $\mathcal{F}_{open}^\Lambda$ and $\mathcal{F}_{settlement}^\Lambda$ relies on the result of \mathcal{F}_{update} .
- \mathcal{F}_{update} and \mathcal{F}_{read} does not need checking with \mathcal{F}_{ledger} to guarantee *safety*.
- \mathcal{F}_{update} requires agreement from $\theta(n_{OP})$ participants.

Definition 5.3. An L2 protocol Λ is a rollup protocol if the ideal functionality $\mathcal{F}_{layer2}^\Lambda$ realized by Λ satisfies the following:

- Subroutine $\mathcal{F}_{open}^\Lambda$ and $\mathcal{F}_{settlement}^\Lambda$ relies on the result of \mathcal{F}_{update} .
- Checking with \mathcal{F}_{ledger} is the *necessary condition* for \mathcal{F}_{update} and \mathcal{F}_{read} to guarantee *safety*.

5.2 Security and Performance Properties

5.2.1 Comparison of Properties. The key distinction between L2 protocol types lies in the structure of their subroutines, which in turn affects how they satisfy security properties. While all secure L2 protocols must guarantee *correct initialization* and *correct settlement*, the performance and assumptions required to satisfy other properties may differ across designs. In what follows, we compare how PCF, sidechain, and rollup protocols realize three key properties: *f*-Safety, *(f, T)*-Liveness, and Data Availability. The results are summarized in Table 2.

***f*-Safety** characterizes a protocol’s resilience to adversarial corruption while maintaining consistency of the off-chain state. Across all L2 designs, the safety of state transitions depends on the behavior of the operator(s). In PCFs, safety is strongest: each state update must be agreed upon by *all* operators, so the protocol remains safe as long as at least one operator is honest. Sidechains achieve safety through their consensus protocol, typically tolerating up to *f* Byzantine faults (BFT threshold). In rollups, safety depends on the underlying L1 ledger and the existence of at least one honest operator. In zk-rollups, safety depends solely on the safety of the L1, as state transitions are verified through validity proofs. In contrast, optimistic rollups additionally rely on L1’s liveness, as fraud proofs must be submitted and resolved within a bounded time window.

***(f, T)*-Liveness** characterizes the protocol’s resilience to adversarial blocking of honest requests, as well as the latency incurred before such requests are successfully processed. We distinguish between two sources of latency: T_{L2} , caused by execution and communication within the L2 protocol, and T_{L1} , incurred by interactions with the L1 ledger.

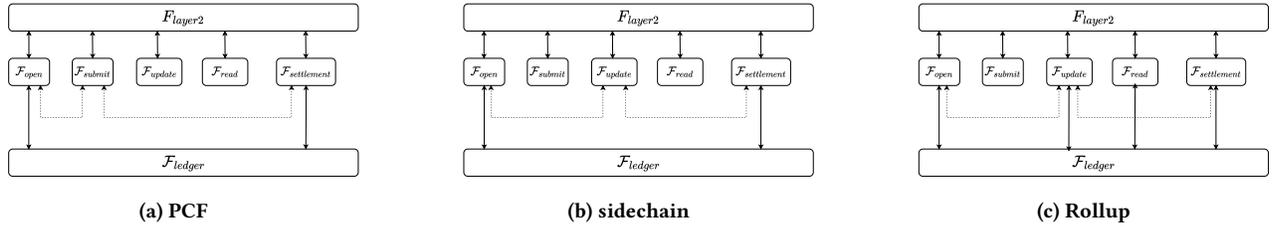


Figure 6: Connection among main parameter subroutines. The solid line represents the direct I/O connection. The dotted line represents the indirect connection.

Table 2: Comparison of security properties and efficiency properties for different types of L2 protocol. Assuming there are $n_p = n_C + n_{OP}$ participants in total for clients and operators. f_{OP} is the maximum corrupted operator assumed by the sidechain consensus algorithm. And there are m executed state update requests after the protocol starts and no clients leave the protocol.

	f-safety	(f,T)-Liveness					Data availability	
		f			T			
PCF	$(n_{OP} - 1)$	$0_p + \text{Layer 1}$	0_{OP}	$= \text{Layer 1}$	$T_{L_2} + T_{L_1}$	T_{L_2}	T_{L_1}	$\{\Omega(m), \Omega(n_p)\}$
sidechain	f_{OP}	$\lfloor \frac{n - (f_{OP} + 1)}{2} \rfloor + \text{Layer 1}$	$\lfloor \frac{n - (f_{OP} + 1)}{2} \rfloor$	$\lfloor \frac{n - (f_{OP} + 1)}{2} \rfloor + \text{Layer 1}$	$T_{L_2} + T_{L_1}$	T_{L_2}	$T_{L_2} + T_{L_1}$	$\{\Omega(m), \Omega(n_p)\}$
Rollup	$= \text{Layer 1}$	$(n_{OP} - 1) + \text{Layer 1}$			$T_{L_2} + T_{L_1}$			$\{\Omega(1), \Omega(m) + \Omega(n_p)\}$

Although our framework defines three types of requests—*open*, *update*, and *settlement*—their liveness varies across L2 designs.

In sidechains and rollups, the update logic is reused across request types via $\mathcal{F}_{\text{update}}$, resulting in uniform liveness behavior. For sidechains, T_{L_2} -bounded liveness is guaranteed for off-chain updates as long as the consensus threshold is met (e.g., a majority of honest operators). However, operations that require anchoring on L1—such as joining or exiting—also depend on the liveness of the L1 ledger, yielding a total latency of $T_{L_2} + T_{L_1}$.

Rollups exhibit uniform liveness across all requests, as every update must be finalized on L1. As long as one honest operator exists and the L1 ledger is live, progress is guaranteed. Thus, rollups are less sensitive to L2 synchrony and exhibit optimal fault tolerance but they are entirely dependent on L1 throughput and liveness guarantees, resulting in a latency of $T_{L_2} + T_{L_1}$ for all operations.

In contrast, PCFs exhibit request-specific liveness behavior. The *open* and *update* procedures require active participation from all involved parties, meaning any crash or delay can block progress. As a result, T_{L_2} is undefined or unbounded unless all participants are honest and responsive. The *settlement* procedure, however, supports unilateral execution, and its liveness depends solely on L1 liveness, with latency dominated by T_{L_1} . In general, PCF protocols are highly sensitive to participant behavior, with liveness varying significantly across subroutines.

Data Availability. The $\mathcal{F}_{\text{read}}$ subroutine models the ability of clients to access the current L2 state. The availability guarantees and storage complexity vary significantly across L2 designs.

In PCFs and sidechains, data availability depends on the assumption that a sufficient number of honest operators are online and responsive. Clients retrieve the current state by combining the initial configuration, stored on the L1 ledger, with off-chain logs of executed requests. As a result, both designs require $\Omega(n_p)$ storage on L1, where n_p is the number of participants, and $\Omega(m)$ storage

off-chain, where m is the number of executed requests representing the L2 ledger size.

Rollups, by contrast, enforce data availability through on-chain publication. All executed requests and resulting states are posted to the L1 ledger, allowing the full state to be reconstructed directly from on-chain data without relying on operator availability. This design yields an L1 storage complexity of $\Omega(n_p + m)$, while removing the need for persistent off-chain storage to guarantee availability. Accordingly, no lower bound on L2 storage is required, since all state-relevant data resides on L1. We emphasize, however, that this is a design choice. In more decentralized rollup architectures, data availability could instead be ensured by relying on existential honesty among L2 archival nodes—i.e., requiring that at least one node faithfully stores the L2 ledger [4, 12]. In such designs, the on-chain storage burden may be reduced at the cost of additional trust assumptions on L2 participants.

Our framework captures this via $\mathcal{F}_{\text{read}}$ depending on local storage (PCFs, sidechains) or $\mathcal{F}_{\text{ledger}}$ (rollups).

5.2.2 Properties Tradeoffs. After outlining how L2 protocols differ in security and efficiency properties, we now examine the tradeoffs these designs entail. Since the core goal of an L2 protocol is to execute state updates efficiently while maintaining consistency, this objective is captured by the *safety* and *liveness* properties. We show that a fundamental trade-off exists between them for state update requests. Omitted proofs appear in Appendix E.

THEOREM 5.4. *A secure PCF and sidechain protocol can only realize f_{OP} -safety and $\{\lfloor \frac{n_{OP} - (f_{OP} + 1)}{2} \rfloor, T_{L_2}\}$ -liveness for state update requests.*

THEOREM 5.5. *A secure rollup protocol can only realize L1-Safety and $\{(n_{OP} - 1) + \text{Layer 1}, T_{L_2} + T_{L_1}\}$ -Liveness for state update requests.*

Beyond the relationship between security properties, our framework also reveals a trade-off between security and efficiency. In

particular, recall that under our definition, *liveness* for a given request holds if the resulting state can be retrieved via a corresponding read request. As demonstrated in the case studies, different L2 protocols implement the $\mathcal{F}_{\text{read}}$ subroutine using distinct mechanisms to determine the final read result. These design choices directly affect how L1 and L2 storage are utilized in practice. We summarize this connection below.

THEOREM 5.6. *Assume m state update requests are executed after the protocol starts. To guarantee liveness for secure PCF protocol and sidechain protocol, the Data availability needs to have $\{\Omega(m), \Omega(n_p)\}$ efficiency.*

THEOREM 5.7. *Assume m state update requests are executed after the protocol starts. To guarantee liveness for secure rollup protocol, the Data availability needs to have $\{\Omega(1), \Omega(m) + \Omega(n_p)\}$ efficiency.*

6 Related Work

Payment channels, e.g., [2, 3, 5, 6, 9, 14, 20, 24, 27], enable two parties to conduct off-chain transactions by locking funds on-chain and exchanging signed messages that update their balance, with only the final state submitted to the blockchain for settlement. Security analyses for these protocols typically focus on the closing phase, capturing guarantees such as *balance security*, which ensures that honest parties can reclaim their rightful funds. However, these analyses are often protocol-specific and do not generalize to broader L2 architectures. In contrast, our framework captures payment channels as a special case of a general L2 functionality, enabling uniform reasoning and comparison across diverse protocol designs.

Sidechains [7, 18, 25] operate as separate blockchains with their own consensus mechanisms, often maintained by federated operators. Clients interact with the sidechain while funds remain escrowed on the L1. Existing work defines security in terms of *ledger safety* and *liveness*, drawing from traditional blockchain theory [17]. These models typically analyze the sidechain in isolation and do not easily support cross-paradigm comparisons or composable reasoning. In our framework, sidechains are treated within the same ideal functionality as other L2 protocols, allowing their assumptions and trade-offs to be systematically compared.

Rollups [8, 12, 15, 20, 23] outsource execution to an off-chain sequencer that posts commitments and proofs to the L1. Optimistic rollups rely on fraud proofs and timeout-based dispute windows, while zk-rollups use succinct validity proofs. A central challenge in rollups is *data availability*, ensuring honest parties can reconstruct and verify protocol state. Although addressed in protocol design and empirical analysis [12, 15], formal models are lacking while existing analyses are protocol-specific. Our framework unifies these models by capturing rollup behavior through a common interface structure, exposing trade-offs between availability, latency, and trust assumptions.

Security Frameworks. Formal frameworks for L2 protocols have primarily focused on specific constructions. Aumayr et al. [1] formalize state channel networks using UC, while Kiayias et al. [22] study composability in Lightning-style channels. These approaches offer strong guarantees but do not extend to sidechains or rollups. In the L1 setting, Graf et al. [19] introduce a general framework for distributed ledgers based on trust and consistency assumptions.

Our work builds on this line of reasoning by presenting the first general security framework for L2 protocols. It supports composable reasoning across L2 paradigms and formalizes shared and diverging properties within a single ideal functionality. [21] formalizes L2 protocols by modeling L1 as responsible solely for participant registration and L2 for execution. However, this abstraction does not accurately capture rollup protocols, which rely on L1 for more than just participant registration. In contrast, [13] focuses on additional functionalities of zk-rollups, such as censorship resistance and protocol upgrades, rather than the core execution logic shared across L2 protocols. Our framework captures the essential logic underlying all types of L2 protocols. Furthermore, while [20] informally discusses the design trade-offs in L2 protocols, it only addresses implementation differences without formally analyzing the corresponding security guarantees.

7 Conclusion

This paper introduced the first general security framework for Layer 2 (L2) blockchain protocols, formalized as a modular ideal functionality in the iUC framework. By structuring protocol behavior through well-defined subroutines—capturing initialization, request submission, state updates, reading, and settlement—our framework abstracts over protocol-specific mechanisms while preserving the key dimensions of L2 security.

This abstraction enables, for the first time, composable reasoning across structurally diverse L2 designs such as payment channels, sidechains, and rollups. It also provides a formal foundation for comparing their safety, liveness, and data availability guarantees within a common model. Our framework not only validates known trade-offs—such as the reliance of rollups on L1 liveness versus the sensitivity of payment channels to participant availability—but also reveals implicit design constraints and minimal trust assumptions previously only understood informally.

Through detailed case studies and a comparative analysis, we demonstrate the generality and applicability of our model, and prove that key properties can be realized under standard assumptions. More broadly, our framework lays the groundwork for formal analysis of emerging challenges in L2 protocol design. Its modular structure enables composable reasoning and supports extensions to other settings, e.g., with rational adversaries by leveraging Rational Protocol Design [16] to capture utility-based guarantees and startegic behavior. Our subroutine-based abstraction is also well-suited to modeling cross-chain interoperability and multi-hop L2 compositions, where clear isolation of roles and assumptions is essential. We see this work as a foundational step toward the security- and incentive-aware design of scalable blockchain infrastructure.

References

- [1] Lukas Aumayr, Oguzhan Ersoy, Andreas Erwig, Sebastian Faust, Kristina Hostáková, Matteo Maffei, Pedro Moreno-Sanchez, and Siavash Riah. 2021. Generalized channels from limited blockchain scripts and adaptor signatures. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 635–664.
- [2] Lukas Aumayr, Pedro Moreno-Sanchez, Aniket Kate, and Matteo Maffei. 2021. Blitz: Secure {Multi-Hop} payments without {Two-Phase} commits. In *30th USENIX Security Symposium (USENIX Security 21)*. 4043–4060.
- [3] Lukas Aumayr, Sri AravindaKrishnan Thyagarajan, Giulio Malavolta, Pedro Moreno-Sanchez, and Matteo Maffei. 2022. Sleepy channels: Bi-directional payment channels without watchtowers. In *Proceedings of the 2022 ACM SIGSAC*

- Conference on Computer and Communications Security*. 179–192.
- [4] Zeta Avarikioti, Makis Arsenis, Dimitris Karakostas, and Orfeas Stefanos Thyfronitis Litos. 2022. Milkomeda Rollup. <https://milkomeda.com/Milkomeda%20Rollup.pdf>
 - [5] Zeta Avarikioti, Eleftherios Kokoris-Kogias, Roger Wattenhofer, and Dionysis Zindros. 2021. Brick: Asynchronous incentive-compatible payment channels. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II 25*. Springer, 209–230.
 - [6] Zeta Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. 2020. Cerberus channels: Incentivizing watchtowers for bitcoin. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 346–366.
 - [7] Mihailo Bjelic, Sandeep Nailwal, Amit Chaudhary, and Wenxuan Deng. [n. d.]. POL: One token for all Polygon chains. <https://polygon.technology/papers/pol-whitepaper> Accessed: 2025-04-14.
 - [8] Lee Bousfield, Rachel Bousfield, Chris Buckland, Ben Burgess, Joshua Colvin, Edward W Felten, Steven Goldfeder, Daniel Goldman, Braden Huddleston, H Kalonder, et al. 2022. Arbitrum nitro: A second-generation optimistic rollup.
 - [9] Conrad Burchert, Christian Decker, and Roger Wattenhofer. 2018. Scalable funding of bitcoin micropayment channel networks. *Royal Society open science* 5, 8 (2018), 180089.
 - [10] Jan Camenisch, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2019. iUC: Flexible universal composability made simple. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 191–221.
 - [11] Ran Canetti. 2001. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 136–145.
 - [12] Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Pedro Moreno-Sánchez, and César Sánchez. 2024. Fast and Secure Decentralized Optimistic Rollups Using Setchain. *arXiv preprint arXiv:2406.02316* (2024).
 - [13] Stefanos Chaliasos, Denis Firsov, and Benjamin Livshits. 2024. Towards a formal foundation for blockchain rollups. *arXiv preprint arXiv:2406.16219* (2024).
 - [14] Christian Decker and Roger Wattenhofer. 2015. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Stabilization, Safety, and Security of Distributed Systems: 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18–21, 2015, Proceedings 17*. Springer, 3–18.
 - [15] Ben Fisch, Arthur Lazzaretti, Zeyu Liu, and Lei Yang. 2024. Permissionless verifiable information dispersal (data availability for bitcoin rollups). *Cryptology ePrint Archive* (2024).
 - [16] Juan Garay, Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. 2013. Rational protocol design: Cryptography against incentive-driven adversaries. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. IEEE, 648–657.
 - [17] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. 2017. The bitcoin backbone protocol with chains of variable difficulty. In *Annual International Cryptology Conference*. Springer, 291–323.
 - [18] Peter Gaži, Aggelos Kiayias, and Dionysis Zindros. 2019. Proof-of-stake sidechains. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 139–156.
 - [19] Mike Graf, Daniel Rausch, Viktoria Ronge, Christoph Egger, Ralf Küsters, and Dominique Schröder. 2021. A security framework for distributed ledgers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1043–1064.
 - [20] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. 2020. Sok: Layer-two blockchain protocols. In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 201–226.
 - [21] Anurag Jain, Sanidhay Arora, Sankarshan Damle, and Sujit Gujar. 2022. Tiramisu: Layering consensus protocols for scalable and secure blockchains. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–3.
 - [22] Aggelos Kiayias and Orfeas Stefanos Thyfronitis Litos. 2020. A composable security treatment of the lightning network. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 334–349.
 - [23] Matter Labs. 2025. zkSync Documentation. <https://docs.zksync.io/> Accessed: 2025-03-06.
 - [24] Andrew Miller, Iddo Bentov, Surya Bakshi, Ranjit Kumaresan, and Patrick McCorry. 2019. Sprites and state channels: Payment networks that go faster than lightning. In *International conference on financial cryptography and data security*. Springer, 508–526.
 - [25] Jonas Nick, Andrew Poelstra, and Gregory Sanders. 2020. Liquid: A bitcoin sidechain. *Liquid white paper*. URL <https://blockstream.com/assets/downloads/pdf/liquid-whitepaper.pdf> (2020).
 - [26] Optimism Collective. 2024. Optimism Rollup Stack: Overview. <https://docs.optimism.io/stack/rollup/overview> Accessed: 2025-04-09.
 - [27] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

- [28] Erkan Tairi, Pedro Moreno-Sanchez, and Matteo Maffei. 2021. A 2 l: Anonymous atomic locks for scalability in payment channel hubs. In *2021 IEEE symposium on security and privacy (SP)*. IEEE, 1834–1851.

A The iUC Framework

In this section, we provide a brief introduction to the Interactive Universal Composability (iUC) framework, which underpins our proposed framework and analysis. For a complete formal specification, we refer the reader to the original iUC framework paper [10]. The iUC framework is a highly expressive and modular model for universal composability and supports the analysis of various types of protocols in different security settings. It is based on interactive Turing machines, which communicate via designated interfaces.

In the iUC framework, a protocol \mathcal{P} is defined as a set of machines: $\mathcal{P} = \{M_1, \dots, M_n\}$. Each machine M_i implements one or more roles, where a role defines the behavior of an entity executing a specific part of the protocol logic. For instance, in Layer 2 (L2) protocols, distinct roles, such as clients and operators, are instantiated. During the execution of a protocol, multiple instances of each machine may run concurrently. These instances interact with each other and with the adversary via designated I/O and network interfaces (NET). An instance of machine M_i may host one or more entities, each identified by a tuple (pid, sid, role), representing the party ID, session ID, and assigned role, respectively. Communication between entities is permitted only if their respective machine instances are appropriately connected via I/O interfaces (i.e., the callee machine is registered as a subroutine of the caller).

To illustrate the relationship among these elements, consider a signature scheme. In the iUC framework, the real-world protocol is typically modeled with two machines: one implementing the role of signer and the other implementing the verifier. Each machine instance governs an entity representing a signing or verification execution. In contrast, the ideal functionality for such a scheme may use a single machine that implements both roles and contains multiple entities, thereby modeling the shared internal state required to define security properties like unforgeability.

The iUC framework defines two types of roles when describing the protocol: public and private roles. In the iUC framework, only public roles can be accessed by the environment. The protocol specification syntax “(pubrole₁, ..., pubrole_n | privrole₁, ..., privrole_m)” explicitly denotes which roles are public and which are private.

There are some iUC functions that are not specifically mentioned in our machine description. When an entity (pid, sid, role) is addressed for the first time, the request is routed to all instances that implement the specified role until one accepts it. This decision is made using the **CheckID** algorithm, and in our framework by default, **CheckID** accepts all entities from the same session. If no entities have yet been accepted, the instance runs the **Initialization** algorithm to initialize its internal state with the global parameter. If an entity is corrupted by the adversary, the instance will record it in **CorruptionSet** to keep track of the corrupted party and uncorrupted ones.

To define the universal composability (UC) security experiment, the model distinguishes between three types of machines: protocols, environments, and adversaries. As is standard in UC-based models, all machines must run in polynomial time. The core security

experiment compares the real-world protocol \mathcal{P} with an ideal functionality \mathcal{F} . The protocol \mathcal{P} is said to securely realize \mathcal{F} if for every adversary \mathcal{A} interacting with \mathcal{P} , there exists a simulator \mathcal{S} such that the joint executions of $(\mathcal{A}, \mathcal{P})$ and $(\mathcal{S}, \mathcal{F})$ are indistinguishable to any environment \mathcal{E} . In this case, we say \mathcal{P} iUC realizes \mathcal{F} .

B Case Study: The Brick Channel

B.1 Brick Channel Real Protocol

To start with, we first propose the protocol implementation $\mathcal{P}^{\text{Brick}}$ in the real world. The real protocol's structure is shown in Figure. 3, and is formally defined as $\mathcal{P}^{\text{Brick}} = (\text{client}|\text{warden}, \mathcal{F}_{\text{cert}})$. *client* and *warden* are the two participant roles in the Brick protocol, but here, *client* is the only interface to the outside since the clients are the main party who get instructed by the environment to proceed with the protocol, and wardens are the service parties to prevent the incorrect state settlement to the L1 ledger. Here we assume there are $n = 3f + 1$ wardens in total, and the adversary could corrupt no more than f wardens.

B.1.1 Client Machine. The client machine defines the code running by the two main parties of the Brick channel that do offchain transactions.

Description of $\mathcal{M}_{\text{client}}$ of protocol $\mathcal{P}^{\text{Brick}}$
<p>Implement roles: Client (Operator)</p> <p>Internal state :</p> <ul style="list-style-type: none"> • Round • RequestQueue • ExecutedRequest • StateList • OnchainState • Identities <p>Main:</p> <p>Receive {Open, Identities, InitialState} from I/O:</p> <ol style="list-style-type: none"> (1) If request is valid, generates $s = \text{InitialState}$, and $\text{Sig}(s)$ from $\mathcal{F}_{\text{cert}}$; (2) Request \mathcal{A} through NET for message delivery. If receives request from NET, send {Open, Identities, InitialState, Sig} to $(\text{pid}, \text{sid}_{\text{cur}}, \text{client})$; (3) If receive (Open, Sig'), then send $\{s, \text{Sig}, \text{Sig}'\}$ to all $(\text{pid}, \text{sid}_{\text{cur}}, \text{warden})$ after receiving delivery request from \mathcal{A}; (4) If receive $2f + 1$ SigW, then send (Submit, $TX_{\text{open}} = \{w_1, \dots, w_n, f, \text{InitialState}\}$) to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$; (5) Send Read to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ if TX_{open} is committed by all participants, reply {Open, InitialState} to environment \mathcal{E} through I/O; <hr/> <p>Receive {Submit, NewState $\{s, i\}$} from I/O:</p> <ol style="list-style-type: none"> (1) Check validity of the request, if the check passes, then generate $\text{Sig}(s)$ with $\mathcal{F}_{\text{cert}}$, add $\{s, i\}$ to RequestQueue; (2) Send {Submit, $\{s, i\}, \text{Sig}$} to counterparty $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{client})$ after receiving message delivery request from adversary \mathcal{A} through NET; (3) If receive {Submit, $\{s, i\}, \text{Sig}'$} from counterparty client, check the validity with $\mathcal{F}_{\text{cert}}$; (4) If check passes, send $\{\{s, i\}, \text{Sig}, \text{Sig}'\}$ to all $(\text{pid}, \text{sid}_{\text{cur}}, \text{warden})$ after receiving message delivery request from \mathcal{A} through NET;

<p>Receive (Update, SigW) from $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{warden})$ through I/O:</p> <ol style="list-style-type: none"> (1) If already received $2f$ request before. Add $\{s, i\}$ to StateList; (2) Add $\{\text{Sig}, \text{Sig}', \{\text{SigW}\}_{2f+1}\}$ to ExecutedRequest; <hr/> <p>Receive {Settlement, collaborate} from I/O:</p> <ol style="list-style-type: none"> (1) Choose the latest state $s_L = \text{Latest}(\text{StateList})$, generate Sig from $\mathcal{F}_{\text{cert}}$; (2) Send {Settlement, s_L, Sig} to counterparty $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{client})$ after receiving the message delivery notification from \mathcal{A} through NET; (3) If receive {Settlement, s_L, Sig'}, send {Submit, $TX_{\text{close}} = \{s_L, \text{Sig}, \text{Sig}'\}$} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$. (4) Send Read to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$, if TX_{close} is committed, outputs {Settlement, success, s_L} to environment \mathcal{E} through I/O. <hr/> <p>Receive {Settlement, unilateral} from I/O:</p> <ol style="list-style-type: none"> (1) Choose the latest state $s_L = \text{Latest}(\text{StateList})$; (2) Send {Settlement, unilateral} to all $(\text{pid}, \text{sid}_{\text{cur}}, \text{warden})$ after receiving delivery request from \mathcal{A} through NET; (3) Send Read to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$, wait for $2f + 1$ publishements from warden. (4) Send {Submit, $s_L, r_L, \text{fraud-proof}$} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$; (5) Send Read to $\mathcal{F}_{\text{ledger}}$, if latest state is committed, outputs closed to I/O; <hr/> <p>Receive {Read} from I/O:</p> <ol style="list-style-type: none"> (1) Outputs {ExecuteRequest, StateList} to I/O;
--

B.1.2 Warden Machine. The warden machine defines the code run by the third-party wardens, they do not get inputs from the environment but only communicate with the clients through the asynchronous communication channel, in which the message delivery is controlled by the adversary, which simulates the asynchronous communication.

Description of $\mathcal{M}_{\text{warden}}$ of protocol $\mathcal{P}^{\text{Brick}}$
<p>Implement roles: Warden</p> <p>Internal state :</p> <ul style="list-style-type: none"> • ExecutedRequest <p>Main:</p> <p>Receive {Open, InitialState Sig, Sig'} from $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{client})$ through I/O:</p> <ol style="list-style-type: none"> (1) Check the validity of $\{\text{Sig}, \text{Sig}'\}$ with $\mathcal{F}_{\text{cert}}$; (2) Generate SigW(S) from $\mathcal{F}_{\text{cert}}$; (3) Send $\{s, \text{SigW}\}$ to both $(\text{pid}, \text{sid}_{\text{cur}}, \text{client})$ after receiving the message delivery notification from \mathcal{A} through NET; (4) Send {Submit, $TX_{\text{open}} = \{\text{InitialState}\}$} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$; <hr/> <p>Receive {Update, $\{s, i\}, \text{Sig}, \text{Sig}'$} $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{client})$:</p> <ol style="list-style-type: none"> (1) Check the validity of $\{\text{Sig}, \text{Sig}'\}$ with $\mathcal{F}_{\text{cert}}$;

- (2) Check sequence number i for consistency;
- (3) Genrate $\text{Sig}^W(S)$ from $\mathcal{F}_{\text{cert}}$;
- (4) Send $\{\text{Update}, \text{Sig}^W\}$ to both $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \text{client})$ after receiving message delivery request from \mathcal{A} through NET;
- (5) Add $\{\{s, i\}, \text{Sig Sig}'\}$ to ExecutedRequest;

Receive $\{\text{Settlement}, \text{unilateral}\}$ from NET:

- (1) Choose the latest stored $\{\{s, i\}, \text{Sig Sig}'\}$;
- (2) Send $\{\text{Submit}, \{\{s, i\}, \text{Sig Sig}'\}, \text{Sig}^W\}$ to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{cleint})$;

B.2 Brick Channel Ideal Functionality

After proposing the real protocol, we formally define the ideal functionality for the Brick channel based on our framework, the ideal functionality is defined as $\mathcal{F}_{\text{layer2}}^{\text{Brick}} = (\mathcal{F}_{\text{layer2}}^{\text{Brick}} | \mathcal{F}_{\text{init}}^{\text{Brick}}, \mathcal{F}_{\text{submit}}^{\text{Brick}}, \mathcal{F}_{\text{update}}^{\text{Brick}}, \mathcal{F}_{\text{read}}^{\text{Brick}}, \mathcal{F}_{\text{settlement}}^{\text{Brick}}, \mathcal{F}_{\text{updRnd}}^{\text{Brick}}, \mathcal{F}_{\text{leak}}^{\text{Brick}})$.

B.2.1 Submit Functionality. The submit function handles the requests that would change the protocol status,

- Open request: the request to open the Brick channel with an initial state and all the participants' identities. Other requests will be rejected unless the open request is executed.
- State transition request: since the Brick channel is used for off-chain payments, a state transition request can be understood as a payment request sent by the payer.
- Settlement request: settlement request means the participant requests to settle its state on L1 ledger, and once it is submitted, other requests from the party will be rejected.

Generally, we use subroutine $\mathcal{F}_{\text{submit}}^{\text{Brick}}$ to check whether the request sent from the environment is correct in the format and whether it is valid according to the current protocol status.

Description of $\mathcal{M}_{\text{submit}}^{\text{Brick}}$ of subroutine $\mathcal{F}_{\text{submit}}^{\text{Brick}}$
<p>Implement roles: Submit</p> <p>Main:</p> <p>Receive $\{\text{Submit}, \text{request}, \text{Internal state}\}$ from I/O:</p> <ol style="list-style-type: none"> (1) Check the received request is within one of the three valid types: <ul style="list-style-type: none"> • Open request: $\{\text{Open}, \text{InitialState}, \text{Identities}\}$; • State update request for a new state $\{s, i\}$: $\{\text{Update}, \{s, i\}\}$; • Settlement request: $\{\text{settlement}, \text{collaborate}, \{s, i\}\}$ and $\{\text{settlement}, \text{unilateral}\}$; (2) Check the open request has been executed or request is the open request; (3) Check there is no settlement request recorded in Internal state; (4) Wait for $\mathcal{F}_{\text{leak}}^{\text{Brick}}$ decides the leakage; (5) If all checks pass, return $\{\text{Submit}, \text{True}, \text{leakage}\}$;

B.2.2 Open Functionality. Similar to what is done in the real Brick payment channel protocol, the open subroutine of $\mathcal{F}_{\text{open}}^{\text{Brick}}$ outputs the success request, the initial states, and the necessary attachments

like the warden signatures, only after requests from both parties are received and the state is committed on the L1 ledger, which is same as that's been done in the real Brick protocol $\mathcal{P}^{\text{Brick}}$.

Description of $\mathcal{M}_{\text{open}}^{\text{Brick}}$ of subroutine $\mathcal{F}_{\text{open}}^{\text{Brick}}$
<p>Implement roles: Open</p> <p>Main:</p> <p>Receive $\{\text{Open}, \text{InitialState}, \text{Attachment}, \text{Internal state}\}$ from I/O:</p> <ol style="list-style-type: none"> (1) Check Internal state's RequestQueue and there exist the same open requests from all honest clients. (2) Check there are $2f + 1$ agreements in Attachment from the wardens recorded in Identities of Internal state. (3) Send Read request to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ and check InitialState is commit on L1 ledger by all participants recorded in Identities; (4) Wait for $\mathcal{F}_{\text{leak}}^{\text{Brick}}$ decides the leakage; (5) If all checks passes, reply $\{\text{Open}, \text{True}, \text{leakage}\}$ to I/O;

LEMMA B.1. *The subroutine $\mathcal{F}_{\text{open}}^{\text{Brick}}$ guarantees Correct Initialization and $(1_p + f_{\text{Layer1}}, T_{L_2} + T_{L_1})$ -liveness for the opening procedure.*

PROOF. We prove this by contradiction. Suppose *Correct Initialization* can be violated by the adversary. This would imply that the environment receives a successful initialization request for at least one honest participant under one of the following two conditions: (1) the finalized initial state included in the output differs from the state proposed by the honest client, or (2) the state committed on the L1 ledger is different, or no commitment is made at all.

However, $\mathcal{F}_{\text{open}}^{\text{Brick}}$ only outputs a success request if the initialization check, triggered by the simulator, contains the initial state proposed by all honest clients via $\mathcal{F}_{\text{submit}}^{\text{Brick}}$. Additionally, $\mathcal{F}_{\text{open}}^{\text{Brick}}$ interacts with $\mathcal{F}_{\text{ledger}}$ to verify that this initial state is indeed committed on the L1 ledger. As long as the transaction can not be forged and the L1 ledger is secure, this contradicts the assumption. Thus, *Correct Initialization* is guaranteed by $\mathcal{F}_{\text{open}}^{\text{Brick}}$.

Now assume that $\mathcal{F}_{\text{open}}^{\text{Brick}}$ fails to generate a positive output even when fewer than $1 + f_{\text{Layer1}}$ participants are corrupted and a delay greater than $T_{L_2} + T_{L_1}$ has elapsed. If no participants are corrupted and the liveness of the L1 ledger is guaranteed, then the adversary can, at most, delay the opening procedure but cannot prevent it. This implies that a successful output will be produced within the worst-case delay of $T_{L_2} + T_{L_1}$. This contradicts the assumption. Therefore, $(1 + \text{Layer } 1, T_{L_2} + T_{L_1})$ -liveness for opening is also guaranteed by $\mathcal{F}_{\text{open}}^{\text{Brick}}$. \square

B.2.3 Update Functionality. In the Brick channel, in order to proceed with the protocol state update, the state transition proposal should satisfy the following two requirements:

- Both clients' agreements
- $2f + 1$ wardens' agreements

The update subroutine checks for the update request from the simulator through the NET. Once the check passes, the subroutine $\mathcal{F}_{\text{update}}^{\text{Brick}}$ will suggest $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ to update the Internal state accordingly.

Description of $\mathcal{M}_{\text{update}}^{\text{Brick}}$ of subroutine $\mathcal{F}_{\text{update}}^{\text{Brick}}$
<p>Implement roles: Update Main:</p> <p>Receive {Update, NewState, Attachment, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Check the Internal state and Attachment that there exists agreement from both clients; (2) Check Attachment includes at least $2f + 1$ agreements from participant wardens; (3) Wait for $\mathcal{F}_{\text{leak}}^{\text{Brick}}$ decides the leakage; (4) If all check passes, reply {Update, True, NewState, leakage};

LEMMA B.2. *The subroutine $\mathcal{F}_{\text{update}}^{\text{Brick}}$ guarantees $(1_{OP}, T_{L_2})$ liveness for state updates.*

PROOF. We prove this by contradiction. Suppose $\mathcal{F}_{\text{update}}^{\text{Brick}}$ does not guarantee *liveness* under no corruption or any time delay. Then consider the case where both clients of the Brick channel and more than $\frac{2}{3}$ of the wardens are honest. In this setting, the simulator can always construct a state update that satisfies the checks defined in $\mathcal{F}_{\text{update}}^{\text{Brick}}$ as long as the adversary allows the message to be delivered.

This contradicts the assumption that *liveness* cannot be guaranteed in any case. Therefore, $\mathcal{F}_{\text{update}}^{\text{Brick}}$ guarantees *liveness* for state updates. \square

B.2.4 Read Functionality. The read function decides what information a participant can learn based on the internal state. Because of data availability, the clients directly obtain output based on the latest RequestQueue, ExecutedReuqets, and Statelist that are able to reconstruct the state transition.

Description of $\mathcal{M}_{\text{read}}^{\text{Brick}}$ of subroutine $\mathcal{F}_{\text{read}}^{\text{Brick}}$
<p>Implement roles: Read Main:</p> <p>Receive {Read, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) If LastReadPointer is the previous one before the latest state. Then send responsive request to \mathcal{S} to decide message delivery; (2) If there is no message delay, get the LatestState from Internal state, update LastReadPointer. (3) Otherwise sey LatestState still be previous state; (4) Check the LatestState should not be older than the state stored in Pointer. Otherwise, (5) Check ExecutedReusts, get the TransitionData that includes all the signatures; (6) If reconstruct LatestState from OnchainState based on TransitionDtata, then reply {Read, ReadResult= {LatestState, TransitionData, LastReadPointer}};

(7) Else reply {Read, readResult = \perp };

LEMMA B.3. *The subroutines $\mathcal{F}_{\text{read}}^{\text{Brick}}$ and $\mathcal{F}_{\text{update}}^{\text{Brick}}$ guarantee $((n_{OP} - 1))$ -safety.*

PROOF. For *safety*, which captures the consistency of the current protocol state among honest clients, we consider the behavior of $\mathcal{F}_{\text{read}}^{\text{Brick}}$. The adversary may influence its output due to asynchronous message delivery and thus may either reflect the latest state in Statelist or the previous read result. Therefore, *safety* would only be violated if the two outputs returned to different honest clients are not prefix-related.

However, according to $\mathcal{F}_{\text{update}}^{\text{Brick}}$, a new valid state requires agreement from both clients, as long as there exists one honest client, there should be no inconsistency for different honest clients. This guarantees that outputs from $\mathcal{F}_{\text{read}}^{\text{Brick}}$ for different honest clients are always consistent in the prefix sense. For self-consistency, the LastReadPointer ensures that any earlier read result is always a prefix of the read result obtained at a later time. Hence, *safety* is ensured. \square

B.2.5 Settlement Functionality. Once the settlement is triggered by the simulator, the settlement subroutine will check whether it is a collaborative closing request with the agreement from both parties or a unilateral closing case. In the previous situation, the subroutine will output success after checking it is both agreed upon and published on the L1 ledger. In the latter case, the subroutine will not only check the latest state committed onchain but also the publishments from the wardens, which are decided by the simulator.

Description of $\mathcal{M}_{\text{settlement}}^{\text{Brick}}$ of subroutine $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$
<p>Implement roles: Settlement Main:</p> <p>Receive {Settlement, SettlementType, Attachment, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) If SettlementType is collaborate, and do the following: <ul style="list-style-type: none"> (a) Check Attachment includes agreement from both clients. (b) Check the Internal state, there exist settlement requests from all honest clients in RequestQueue, set the request settlement state to be LatestState; (c) Send Read request to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L_1})$, to check if there exists a corresponding state on the L1 ledger; (2) If SettlementType is unilateral, then do the following: <ul style="list-style-type: none"> • Search the LatestState from Internal state; • Send Read request to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L_1})$ check at least $2f + 1$ wardens publish onchain, in which LatestState is committed; (3) Wait for $\mathcal{F}_{\text{leak}}^{\text{Brick}}$ decides the leakage; (4) If check passes, reply {Settlement, True, LatestState} to I/O;

LEMMA B.4. *The subroutine $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ guarantees correct settlement, $(1_{OP}, T_{L_2} + T_{L_1})$ -liveness for collaborative settlement request, and $(\text{Layer}_1, T_{L_1})$ -liveness for unilateral settlement request.*

PROOF. The *correct settlement* is violated if the environment receives a successful settlement request from $\mathcal{F}_{\text{layer2}}$ to an honest client, but the corresponding settlement state committed on the L1 ledger is neither the latest state nor the state proposed by any honest client. However, the subroutine $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ only outputs a successful settlement if the committed state on the L1 ledger is either (i) the latest state recorded in `StateList`, or (ii) the state proposed by all honest clients, depending on the type of settlement request (collaborative or unilateral). Therefore, *correct settlement* is guaranteed.

Liveness for settlement requests is violated if the subroutine fails to produce a successful output under any corruption and after bounded delay. For collaborative settlement, if both clients are honest and the liveness of the L1 ledger is guaranteed, then the adversary can at most delay message delivery, but cannot prevent $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ from eventually generating a successful output. For unilateral settlement, if at least one honest client initiates the request and the L1 ledger ensures liveness, then the state will eventually be published and accepted. Therefore, *liveness* for settlement requests is also guaranteed. \square

B.2.6 Update Round Functionality. Since we assume an asynchronous communication channel, there will be no further requirement for round updating.

Description of $\mathcal{M}_{\text{updRnd}}^{\text{Brick}}$ of subroutine $\mathcal{F}_{\text{updRnd}}^{\text{Brick}}$
<p>Implement roles: UpdateRound Main:</p> <p>Receive {UpdateRound, Internal state} from I/O: (1) Reply {UpdateRound, True};</p>

B.3 Security Proof

After proposing the ideal functionality and real-world implementation, we now show the security of the Brick channel protocol. To start with we first show the ideal functionality for Brick capture all the security properties with the following conclusion:

THEOREM 4.1. *The ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ guarantees all the security properties of a secure L2 protocol.*

PROOF. According to Lemma B.1 to B.4, the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ guarantees all security properties. \square

Our main proof strategy is to show that there exists a simulator for the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ that internally simulates an alternative version of the real-world protocol $\mathcal{P}^{\text{Brick}}$ and interacts with the ideal functionality in a way that produces the same outputs to the environment. This ensures that the adversary's view in the real world is indistinguishable from that in the ideal world.

We assume there is a simulator \mathcal{S} that internally simulates the same protocol of $\mathcal{P}^{\text{Brick}}$, which is noted with $\mathcal{P}'^{\text{Brick}}$. The simulator \mathcal{S} behaves according to the following rules:

Network connection with ideal functionality and environment.

- Any message the simulator \mathcal{S} received from the environment through the network connection, which can be understood as the adversary instruction, will be forwarded to the internal simulation $\mathcal{P}'^{\text{Brick}}$.
- Any message sent from the internal simulation $\mathcal{P}'^{\text{Brick}}$ to the network connection, which can be seen as leakage to the environment, will be forwarded to the environment by the simulator.
- Whenever there is a request for open protocol, update state, or settlement state inside the simulation, the simulator will trigger the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ to synchronize the change.

Corruption handling.

- The simulator will keep track of the corrupt entities and keep the corrupted parties in $\mathcal{P}'^{\text{Brick}}$ and $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ synchronized.
- The simulator will forward the messages from the corrupted entities in the ideal functionality to the adversary.

Message delivery. For the L2 communication, the Brick channel assumes the asynchronous network; thus, the message delivery will be eventually decided by the adversary. We let the simulator bookkeep all the messages in the simulation and trigger message delivery in the same way as the real-world protocol.

Request submission. Whenever an honest entity in ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ receives a request from the environment after it gets accepted decided by the subroutine $\mathcal{F}_{\text{submit}}^{\text{Brick}}$, it will forward the transaction to the simulator \mathcal{S} . After receiving the forwarded request from the $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$, \mathcal{S} will send the request to its internal simulated protocol $\mathcal{P}'^{\text{Brick}}$.

Protocol opening. The simulator \mathcal{S} monitors both the internal simulation state and the L1 ledger. Whenever it detects that the open procedure has completed in $\mathcal{P}'^{\text{Brick}}$, it prepares a request to be sent to $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ for validation. This request includes the `InitialState` used to initialize the protocol and the corresponding `Attachment`, which contains the signature agreements generated via $\mathcal{F}_{\text{cert}}$ within the simulation.

Protocol state update. The simulator \mathcal{S} monitors the internal simulation for state updates. Based on the requests previously received from $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$, once \mathcal{S} detects that a state update has been completed within the simulation, it prepares an update request and sends it to $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ to synchronize the internal state. This request includes the updated state and an `Attachment` containing the executed requests, along with signatures from both clients and $2f + 1$ wardens, generated via $\mathcal{F}_{\text{cert}}$.

Protocol state read. When $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ queries \mathcal{S} the message delivery to decide the read result, \mathcal{S} replies according to its interaction with the adversary about the message delivery.

Protocol state settlement. After receiving the settlement request from $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$, the simulator \mathcal{S} executes the settlement procedure as defined in the real protocol for all non-corrupted entities. \mathcal{S} continuously monitors the simulation, and once the settlement procedure is completed, it prepares a request containing the `SettlementType` and an `Attachment` that includes the relevant

signatures generated via $\mathcal{F}_{\text{cert}}$, and sends this request to $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ for checking.

Further details. The simulator \mathcal{S} also maintains synchronization of the internal clock/round with the ideal functionality. Whenever \mathcal{S} is instructed by the adversary \mathcal{A} to advance the round, it sends an `UpdateRound` request to $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ and triggers the corresponding round update.

Then, we prove the security of the protocol by showing that for each procedure, the environment can not distinguish between the output of ideal functionality and the output of the real protocol.

THEOREM 4.2. *Let $\mathcal{F}_{\text{ledger}}$ be the idealized L1 ledger functionality and $\mathcal{F}_{\text{cert}}$ be the idealized functionality for EUF-CMA secure signature scheme. Then, the real Brick payment channel protocol $\mathcal{P}^{\text{Brick}}$ realizes the ideal Brick payment channel functionality $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$.*

PROOF. We analyze the output for each step of the protocol in both worlds.

Request submission. For any received request from the environment, the subroutine $\mathcal{F}_{\text{submit}}^{\text{Brick}}$ will check whether the format is correct. Therefore, the real protocol and ideal functionality realize the same functions. After the checks pass, a request along with the accepted request will be forwarded to the simulator to simulate the same action.

Protocol open. In the real protocol, the open procedure of the Brick channel is not finished immediately, even though an honest party starts the procedure. This is because some parties are corrupted and want to prevent the open procedure from happening, or the messages and actions are delayed or reordered, including publishing on the Layer 1 ledger. Thus, in the ideal world, we let the simulator trigger the open procedure of the ideal functionality to simulate the possible influence. Since we assume the simulator keeps track of the adversary’s instruction in the real world, whenever the open procedure is finished in the simulation $\mathcal{P}'^{\text{Brick}}$, the $\mathcal{F}_{\text{open}}^{\text{Brick}}$ will be triggered by the simulator, where all parties agreement and the onchain committed value will be checked. Once all the check passes, the `Internal` state of $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ will be updated accordingly and notify all the participants. The output for protocol open will only be sent to all the participants if all the requirements are satisfied, which is the same in both worlds. Besides, all the messages in the open procedure are broadcast to the public on the L1 ledger. Thus, the simulator can generate the same message leakage to the environment. Therefore, the environment can not tell the difference.

Protocol state update. Once the protocol is opened, the state of the L2 protocol can be updated. The adversary could influence this procedure by delaying the update procedure or proposing an incorrect update request. We let the simulator simulate all the possible misbehavior, whenever there is an instruction from the adversary, the simulator will act accordingly in the inside simulation $\mathcal{P}'^{\text{Brick}}$, and the simulator will trigger the $\mathcal{F}_{\text{update}}^{\text{Brick}}$ whenever there is a state update in the simulation $\mathcal{P}'^{\text{Brick}}$. The $\mathcal{F}_{\text{update}}^{\text{Brick}}$ will check the update includes the agreement from both parties and at least $2f + 1$ wardens if so it will let the $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ to update the `Internal` state accordingly. The same check is also realized in the real protocol by the honest client machine. Thus, the `Internal` state in the ideal functionality is synchronized with the real protocol as long

as the adversary is not able to forge a signature. Although this procedure does not generate direct output to the environment, the update to the `Internal` state could influence the outputs of other procedures like `read` and `settlement`, which will be discussed in the following.

Protocol state read. In the real protocol, each machine will locally store the information it receives during the protocol execution. Although in the state update procedure, the states are synchronized between the ideal functionality and the real-world protocol, the adversary could still influence the read result. Specifically, since the message delivery is decided by the adversary, although there might be a valid update approved by the ideal function, the honest participants could not know it because the message has not been delivered yet. To simulate such influence, the ideal functionality $\mathcal{F}_{\text{read}}^{\text{Brick}}$ will check with the simulator for message delivery. If the latest message is delivered, it will output the latest state and the data to reconstruct the state transition. Otherwise, the previous state will be output. By doing so, the simulator helps the ideal functionality to generate the same read result as the real protocol does.

Protocol state settlement. Similar to the real protocol, the ideal functionality also takes two types of settlement requests: collateral and unilateral. In the ideal world, the settlement procedure is also triggered through the network connection to simulate the malicious parties’ possible behaviors, like settling the state unilaterally without notifying the counterparty. Once there is an incoming collateral settlement request, $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ checks both two parties agree on the settlement request, and it has already been committed on the L1 ledger. Note that the settlement state, in this case, does not need to be the latest state due to the possible message delay; it is valid as long as there are agreements from both clients (operators). As for the unilateral close, $\mathcal{F}_{\text{settlement}}^{\text{Brick}}$ will check the request and the L1 ledger that the latest state in `Internal` state has been committed onchain. Once these checks pass, all the participants will be notified by $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$ for successful closure. As long as the adversary is not able to forge a signature and construct a valid transaction itself, the outputs of the real protocol and ideal functionality during the settlement procedure are indistinguishable.

Based on the analysis of all the procedures of the Brick channel, we can know that the Brick channel $\mathcal{P}^{\text{Brick}}$ iUC realizes $\mathcal{F}_{\text{layer2}}^{\text{Brick}}$. \square

C Case Study: The Liquid Network Sidechain

C.1 Liquid Network Real Protocol

To analyze the security of the Liquid Network, we first describe the real-world protocol of Liquid Network: $\mathcal{P}^{\text{Liquid}} = (\text{Client} | \text{Operator}, \mathcal{F}_{\text{com}}, \mathcal{F}_{\text{cert}})$. The Liquid Network protocol in the real world is realized based on two types of machines as main components: the client machine $\mathcal{M}_{\text{client}}^{\text{Liquid}}$ and the operator machine $\mathcal{M}_{\text{operator}}^{\text{Liquid}}$. Here we assume there are $n = 3f + 1$ operators in total, and the adversary could corrupt no more than f operators.

C.1.1 Client Machine. The client machine specifies the behavior of an honest client in the Liquid Network protocol as follows:

Description of $\mathcal{M}_{\text{client}}^{\text{Liquid}}$ of protocol $\mathcal{P}^{\text{Liquid}}$
<p>Implement roles: Client</p> <p>Internal state :</p> <ul style="list-style-type: none"> • Round • ExecutedRequest • StateList • OnchainState • Identities <p>Main:</p> <p>Receive {Open, Identities, InitialState} from I/O:</p> <ol style="list-style-type: none"> (1) According to InitialState, prepare the peg-in transaction $TX_{\text{peg-in}}$, and parse the destination account address Address from Identities; (2) Send {Submit, TX_{deposit}, Address} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$; (3) Send Read to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$, wait for TX_{deposit} committed by 100 blocks; (4) Obtain Sig from ideal functions $\mathcal{F}_{\text{cert}}$ on peg-in transaction $TX_{\text{peg-in}}$; (5) Send {Open, $TX_{\text{peg-in}}$, Sig} to all $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$ through \mathcal{F}_{com}; (6) Send {Read} to all $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$, and check $TX_{\text{peg-in}}$ is included in the ReadResult; (7) Reply {Open, InitialState} through I/O; <hr/> <p>Receive {Update, TX} from I/O:</p> <ol style="list-style-type: none"> (1) Obtain Sig on transaction TX from the ideal functions $\mathcal{F}_{\text{cert}}$; (2) Send {Submit, TX, Sig} to to all $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$; <hr/> <p>Receive {Settlement} from I/O:</p> <ol style="list-style-type: none"> (1) Prepare the peg-out transaction $TX_{\text{peg-out}}$ and obtain Sig from $\mathcal{F}_{\text{cert}}$; (2) Send {Settlement, $TX_{\text{peg-out}}$, Sig} to to all $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$; (3) Send {Read} to both $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ and to all $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$; (4) If LatestState is recorded in both ReadResult, reply {Settlement, LatestState}; <hr/> <p>Receive {Read} from I/O:</p> <ol style="list-style-type: none"> (1) Send {Read} to all $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$; (2) Set ReadResult be the chain, in which each block contains $2f+1$ operator signatures; (3) Rely ReadResult through I/O;

C.1.2 Operator Machine. The operator machine defines the code for an honest operator who is responsible for maintaining the Liquid sidechain:

Description of $\mathcal{M}_{\text{operator}}^{\text{Liquid}}$ of protocol $\mathcal{P}^{\text{Liquid}}$
<p>Implement roles: Operator</p> <p>Internal state :</p> <ul style="list-style-type: none"> • Round • RequestQueue

<ul style="list-style-type: none"> • ExecutedRequest • StateList • OnchainState • Identities <p>Main:</p> <p>Receive {Open, $TX_{\text{peg-in}}$} from I/O:</p> <ol style="list-style-type: none"> (1) Verify the validity of $TX_{\text{peg-in}}$; (2) Send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ and check $TX_{\text{peg-in}}$ is committed for 100 blocks; (3) If check passes, add $TX_{\text{peg-in}}$ to RequestQueue and broadcast to all other $(\text{pid}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$ through \mathcal{F}_{com}; <hr/> <p>Receive {Submit, TX, Sig} from I/O:</p> <ol style="list-style-type: none"> (1) Verify the validity of TX and Sig with $\mathcal{F}_{\text{cert}}$; (2) If check passes, add {TX, Attachment} to RequestQueue and broadcast to all other $(\text{pid}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$ through \mathcal{F}_{com}; <hr/> <p>Receive {Settlement, $TX_{\text{peg-out}}$, Sig} from I/O:</p> <ol style="list-style-type: none"> (1) Check the validity of the received Sig with $\mathcal{F}_{\text{cert}}$, which guarantees the requester is a valid client participant; (2) If the check passes, add settlement request $TX_{\text{peg-out}}$ to RequestQueue and broadcast to all other $(\text{pid}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$ through \mathcal{F}_{com}; <hr/> <p>Receive {Read} from I/O:</p> <ol style="list-style-type: none"> (1) Rely StateList and ExecutedRequest through I/O; <hr/> <p>Receive {UpdateLeader} from I/O:</p> <ol style="list-style-type: none"> (1) Generate a new valid block with all the valid requests recorded in RequestQueue; (2) The block should not contain invalid requests cause double-spending; (3) Send {UpdatePropose, Block} to all other $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$ through \mathcal{F}_{com}; <hr/> <p>Receive {UpdatePropose, Block} from I/O:</p> <ol style="list-style-type: none"> (1) Check the validity of Block, which indicates following: <ul style="list-style-type: none"> • Block is in the correct form. • Included transactions are in the correct form and have valid signatures. • There are no double-spending transactions. (2) If all the check passes, obtain Precommitment from $\mathcal{F}_{\text{cert}}$. (3) Send {UpdatePrecommitment, Block, Precommitment} to all other $(\text{pid}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}} : \text{operator})$ through \mathcal{F}_{com}; <hr/> <p>Receive {UpdatePrecommitment, Block, Precommitment} from I/O:</p> <ol style="list-style-type: none"> (1) If already received $2f$ Precommitments before, generate Sig for Block with $\mathcal{F}_{\text{cert}}$. (2) Send {UpdateFinal, Block, Sig} to all other $\mathcal{M}_{\text{operator}}^{\text{Liquid}}$; <hr/> <p>Receive {UpdateFinal, Block, Sig} from I/O:</p> <ol style="list-style-type: none"> (1) For the received Block, if there already received $2f$ Sigs before, then add all the requests in the Block to StateList;
--

C.2 Liquid Network Ideal Functionality

After proposing the real protocol, we formally define the ideal functionality for the Liquid Network sidechain based on our framework, the ideal functionality is defined as $\mathcal{F}_{\text{layer2}}^{\text{Liquid}} = (\mathcal{F}_{\text{layer2}}^{\text{Liquid}} | \mathcal{F}_{\text{init}}^{\text{Liquid}}, \mathcal{F}_{\text{submit}}^{\text{Liquid}}, \mathcal{F}_{\text{update}}^{\text{Liquid}}, \mathcal{F}_{\text{read}}^{\text{Liquid}}, \mathcal{F}_{\text{settlement}}^{\text{Liquid}}, \mathcal{F}_{\text{updRnd}}^{\text{Liquid}}, \mathcal{F}_{\text{leak}}^{\text{Liquid}})$.

C.2.1 Submit Functionality. The submit function handles the requests that would change the protocol status:

The $\mathcal{F}_{\text{submit}}^{\text{Liquid}}$ to check whether the request sent from the clients, who are instructed by the environment, is correct in the format and valid according to the current status of the Liquid Network protocol.

Description of $\mathcal{M}_{\text{submit}}^{\text{Liquid}}$ of subroutine $\mathcal{F}_{\text{submit}}^{\text{Liquid}}$
<p>Implement roles: Submit</p> <p>Main:</p> <p>Receive {Submit, request, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Check the received request is within one of the three valid types: <ul style="list-style-type: none"> • Open request: {Open, InitialState, Identities}; • State update request: {Submit, TX}; • Settlement request: {Settlement}; (2) Check the open request has been executed or request is the open request; (3) Check there is no settlement request from the caller participants; (4) Wait for $\mathcal{F}_{\text{leak}}^{\text{Liquid}}$ decides the leakage; (5) If all checks pass, return {Submit, True, leakage};

C.2.2 Open Functionality. In the Liquid Network, whether a client is considered as successfully participating in the protocol is decided based on two requirements: (1) the commitment transaction needs to be committed on the L1 ledger for 100 blocks; (2) there is also a claiming transaction in the sidechain maintained by the operators.

Description of $\mathcal{M}_{\text{open}}^{\text{Liquid}}$ of subroutine $\mathcal{F}_{\text{open}}^{\text{Liquid}}$
<p>Implement roles: Open</p> <p>Main:</p> <p>Receive {Open, InitialState, Attachment= $TX_{\text{peg-in}}$, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Send Read request to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ and check InitialState is commit on L1 ledger for 100 blocks; (2) Check Internal state that StateList and ExecutedRequest includes the $TX_{\text{peg-in}}$; (3) Wait for $\mathcal{F}_{\text{leak}}^{\text{Liquid}}$ decides the leakage; (4) If all checks passes, reply {Open, True, leakage} to I/O;

LEMMA C.1. $\mathcal{F}_{\text{open}}^{\text{Liquid}}$ guarantees correct initialization and $(f_{OP} + \text{Layer}_1, T_{L_2} + T_{L_1})$ -liveness for open requests.

PROOF. The *correct initialization* property is violated if the initial state committed on the L1 ledger differs from the state recorded in StateList, yet the environment still receives a successful initialization output from $\mathcal{F}_{\text{layer2}}$. However, $\mathcal{F}_{\text{open}}^{\text{Liquid}}$ generates a positive output only if the updated StateList and ExecutedRequest matches the state and the peg-in transaction that have been committed on the L1 ledger. As long as there only exists less than $f_{OP} = \frac{1}{3}n_{OP}$ corrupted operators, and the L1 ledger is secure, there will not be a mismatch. Therefore, *correct initialization* is guaranteed.

Now assume that the *liveness* for an open request from an honest client is broken, meaning that a positive output is never generated under any corruption setting or time delay. However, as long as the *liveness* of the L1 ledger is guaranteed and more than $\frac{2}{3}$ of the operators are honest, $\mathcal{F}_{\text{open}}^{\text{Liquid}}$ will eventually generate a positive output. The worst-case time delay is bounded by $T_{L_2} + T_{L_1}$, representing the time required for communication among L2 operators and for committing a value on the L1 ledger. Thus, the *liveness* of the open request is also guaranteed by $\mathcal{F}_{\text{open}}^{\text{Liquid}}$. \square

C.2.3 Update Functionality. In the liquid network, the StateList and ExecutedRequest in the internal state of $\mathcal{F}_{\text{layer2}}$ is updated based on the BFT protocol that operator running to maintain the sidechain. If all the check passes, the update function generates positive output and

Description of $\mathcal{M}_{\text{update}}^{\text{Liquid}}$ of subroutine $\mathcal{F}_{\text{update}}^{\text{Liquid}}$
<p>Implement roles: Update</p> <p>Main:</p> <p>Receive {Update, NewState={StateList, Block}, Attachment, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Check the Internal state and Attachment that there exists agreements from $2f + 1$ operators; (2) Check all the transactions in the Block that there is no double spending; (3) Wait for $\mathcal{F}_{\text{leak}}^{\text{Liquid}}$ decides the leakage; (4) If all check passes, reply {Update, True, NewState, leakage};

LEMMA C.2. $\mathcal{F}_{\text{update}}^{\text{Liquid}}$ guarantees (f_{OP}, T_{L_2}) -liveness for update requests.

PROOF. Suppose that the *liveness* for update requests is violated. This would imply that $\mathcal{F}_{\text{update}}^{\text{Liquid}}$ never generates a positive output under any corruption scenario or time delay. However, as long as there are only less than $f_{OP} = \frac{1}{3}n_{OP}$ of the operators are corrupted and not responding, the rest of the honest operators can still reach consensus, and the simulator can therefore produce a valid state update to the ideal functionality. In this case, $\mathcal{F}_{\text{update}}^{\text{Liquid}}$ will generate a positive output for the update request within a time delay bounded by T_{L_2} , which is the latency caused solely by the L2 communication. Therefore, *liveness* for update requests is guaranteed by $\mathcal{F}_{\text{update}}^{\text{Liquid}}$. \square

C.2.4 Read Functionality. The read function in the Liquid Network represents the client's procedure for fetching the latest chain status from the operator. Here, we assume each client is connected to all

the operators, and the communication is under the synchronous assumption. Thus, in the ideal function, only the blocks from the previous read result till the latest block should be replied through I/O.

Description of $\mathcal{M}_{\text{read}}^{\text{Liquid}}$ of subroutine $\mathcal{F}_{\text{read}}^{\text{Liquid}}$
<p>Implement roles: Read Main:</p> <p>Receive {Read, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Get LatestState from Internal state. (2) Check the LastReadPointer ExecutedReusts, get the TransitionData that includes all the blocks till the latest one, including all the agreements from operators; (3) If reconstruct LatestState from LatestState based on TransitionDtata, then reply {Read, ReadResult={LatestState, TransitionData}}; (4) Else reply {Read, ReadResult = \perp};

LEMMA C.3. $\mathcal{F}_{\text{update}}^{\text{Liquid}}$ and $\mathcal{F}_{\text{read}}^{\text{Liquid}}$ guarantee $(\frac{1}{3}n_{OP})$ -safety.

PROOF. Since $\mathcal{F}_{\text{read}}^{\text{Liquid}}$ always selects the latest state recorded in StateList as the read result, and each new state along with the according ExecutedRequest must be agreed upon by more than $\frac{2}{3}$ of the operators (as specified in $\mathcal{F}_{\text{update}}^{\text{Liquid}}$), thus conflicting read results cannot occur for different honest clients, which guarantees view-consistency. Additionally, the synchronous communication assumption, together with the assumption that at least one honest operator will respond with the latest state of the sidechain to the client, guarantees self-consistency for honest clients. Therefore, all read results are prefix-comparable, satisfying the *safety* requirement. \square

C.2.5 Settlement Functionality. Once the settlement is triggered by the simulator, similar to the open procedure, the $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$ will also check whether two requirements are satisfied: (1) the settlement request $TX_{\text{settlement}}$ is committed in the StateList of the Internal state; (2) the settlement request $TX_{\text{settlement}}$ should be committed on the L1 ledger as well.

Description of $\mathcal{M}_{\text{settlement}}^{\text{Liquid}}$ of subroutine $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$
<p>Implement roles: Settlement Main:</p> <p>Receive {Settlement, Attachment=$TX_{\text{peg-out}}$, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Search Internal state, check $TX_{\text{peg-out}}$ is included in ExecutedRequest; (2) Obtain the LatestState from the StateList; (3) Send Read request to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ check LatestState is committed on the L1 ledger; (4) Wait for $\mathcal{F}_{\text{leak}}^{\text{Liquid}}$ decides the leakage; (5) If check passes, reply {Settlement, True, LatestState} to I/O;

LEMMA C.4. $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$ guarantees correct settlement and $(f_{OP} + \text{Layer}_1, T_{L_2} + T_{L_1})$ -liveness for settlement requests.

PROOF. *Correct settlement* is considered violated if the environment receives a successful settlement output from $\mathcal{F}_{\text{layer2}}$ to an honest client, but one of the following holds: (1) the client did not propose a settlement request on the L1 ledger, or (2) the state committed on the L1 ledger differs from the latest L2 protocol state. However, $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$ only generates a positive output if: (1) the client explicitly proposes to settle the state on the L1 ledger, and (2) the committed state on the L1 ledger matches the latest state recorded in ExecutedRequest. Therefore, *correct settlement* is guaranteed.

Liveness for settlement requests is considered violated if $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$ does not generate a positive output under any corruption status or time delay. However, as long as more than $\frac{2}{3}$ of the operators are honest and the liveness of the L1 ledger is guaranteed, $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$ will produce a positive output after verifying both the L2 and L1 conditions. The total delay is bounded by $T_{L_2} + T_{L_1}$. Thus, *liveness* is guaranteed. \square

C.2.6 Update Round Functionality. Here, the communication is assumed to be synchronous. Thus, the round update should have certain requirements. Specifically, for any request from the client, there should be a time delay upper bound δ , if the request is not executed with more delay than δ , the round can not be updated.

Description of $\mathcal{M}_{\text{updRnd}}^{\text{Liquid}}$ of subroutine $\mathcal{F}_{\text{updRnd}}^{\text{Liquid}}$
<p>Implement roles: UpdateRound Main:</p> <p>Receive {UpdateRound, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Check Internal state, there exists no request that has not been executed more than δ time. (2) If the check passes, reply {UpdateRound, True};

C.3 Security Proof

After proposing the ideal functionality and real-world implementation, we now show the security of the Liquid Network sidechain. We first prove that the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$.

THEOREM 4.3. *The ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ guarantees all the security properties of a secure L2 protocol.*

PROOF. According to Lemma C.1 to C.4, it can be concluded $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ guarantees all the security properties. \square

Our main proof strategy is to show that there exists a simulator for the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ that internally simulates an alternative version of the real-world protocol $\mathcal{P}^{\text{Liquid}}$ and interacts with the ideal functionality in a way that produces the same outputs to the environment. This ensures that the adversary's view in the real world is indistinguishable from that in the ideal world.

Here we assume there is a simulator \mathcal{S} that internally simulates the same protocol of $\mathcal{P}^{\text{Liquid}}$, which is noted with $\mathcal{P}'^{\text{Liquid}}$. The simulator \mathcal{S} behaves according to the following rules:

Network connection with ideal functionality and environment.

- Any message the simulator \mathcal{S} received from the environment through the network connection, which can be understood as the adversary instruction, will be forwarded to the internal simulation $\mathcal{P}'^{\text{Liquid}}$.
- Any message sent from the internal simulation $\mathcal{P}'^{\text{Liquid}}$ to the network connection, which can be seen as leakage to the environment, will be forwarded to the environment by the simulator.
- Whenever there is a request for open protocol, update state, or settlement state inside the simulation, the simulator will trigger the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ to synchronize the change.

Corruption handling.

- The simulator will keep track of the corrupt entities, and keep the corrupted parties in $\mathcal{P}'^{\text{Liquid}}$ and $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ synchronized.
- The simulator will forward the messages from the corrupted entities in the ideal functionality to the adversary.

Message delivery. For the Liquid Network, we assume the communication is under a synchronous setting. We assume in the real protocol there is a communication function \mathcal{F}_{com} that takes the delay input from the adversary. And the simulator bookkeeps all the received sending through \mathcal{F}_{com} in the simulated $\mathcal{P}'^{\text{Liquid}}$, whenever the adversary \mathcal{A} triggers the message delivery, the simulator will do the same.

Request submission. Whenever an honest entity in ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ receives a request from the environment after it gets accepted decided by the subroutine $\mathcal{F}_{\text{submit}}^{\text{Liquid}}$, it will forward the transaction to the simulator \mathcal{S} . After receiving the forwarded request from the $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$, \mathcal{S} will send the request to its internal simulated protocol $\mathcal{P}'^{\text{Liquid}}$.

sidechain joining. The simulator \mathcal{S} monitors both the internal simulation state and the L1 ledger. Whenever it detects that the open procedure has completed and received output from the simulated entities in $\mathcal{P}'^{\text{Liquid}}$, it prepares a request to be sent to $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ for validation. This request includes the `InitialState` used to initialize the protocol and the corresponding `Attachment`, which contains the signature agreements from the client and operators generated via $\mathcal{F}_{\text{cert}}$ within the simulation, as well as the on-chain committed transaction required for verification.

Protocol state update. The simulator \mathcal{S} simulates the real protocol by allowing the simulated operators to generate new blocks according to the timing signals from the inner clock defined in \mathcal{F}_{com} . It monitors the internal simulation for state updates, and once a new block is generated in the simulated sidechain, \mathcal{S} prepares an update request and sends it to $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ to synchronize the internal state. This request includes the updated state and an `Attachment` containing the newly generated block with the executed requests,

along with the quorum certificate consisting of operator signatures generated via $\mathcal{F}_{\text{cert}}$.

Protocol state read. Because of the synchronous communication assumption, the $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ will not query \mathcal{S} to decide the read result.

Protocol state settlement. After receiving the settlement request from $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$, the simulator \mathcal{S} wait for its execution. \mathcal{S} continuously monitors the simulation and the L1 ledger, and once the settlement procedure is completed, it prepares a request containing the `SettlementType` and an `Attachment` that includes the peg-out transaction published on the L1 ledger and sends this request to $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ for checking.

Further details. The simulator \mathcal{S} also maintains synchronization of the internal clock/round with the ideal functionality. Whenever \mathcal{S} is instructed by the adversary \mathcal{A} to advance the round through \mathcal{F}_{com} in the simulation, it sends an `UpdateRound` request to $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ and triggers the corresponding round update.

Then, we prove the security of the protocol by showing that for each procedure, the environment can not distinguish between the output of ideal functionality and the output of the real protocol.

THEOREM 4.4. *Let $\mathcal{F}_{\text{ledger}}$ be the idealized L1 ledger functionality, \mathcal{F}_{com} be the synchronous communication channel, $\mathcal{F}_{\text{cert}}$ be the idealized functionality for EUF-CMA secure signature scheme. Then, the real Liquid Network sidechain protocol $\mathcal{P}^{\text{Liquid}}$ iUC-realizes the ideal Liquid Network sidechain functionality $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$.*

PROOF. Here we prove it by showing that the outputs for each step of the protocol in both worlds are indistinguishable.

Request submission. For any received request from the environment, the subroutine $\mathcal{F}_{\text{submit}}^{\text{Liquid}}$ will check whether the format is correct, and after the checks pass, a request and the accepted request will be sent to the simulator to simulate the same action. As a result, in both the real world and the ideal world, the same types of valid requests will be accepted.

sidechain join. In the real protocol, the open procedure of the Liquid Network first requires verifying that the commit transaction has been confirmed on the L1 for a sufficient duration, and that the client can observe the corresponding peg-in transaction on the Liquid sidechain. The adversary may influence this procedure by corrupting operators and deviating from the protocol, such as proposing an incorrect peg-in transaction to the sidechain or generating an invalid block containing the peg-in transaction and falsely claiming that the procedure has completed successfully. Therefore, in the ideal world, we allow the simulator to trigger the open procedure of the ideal functionality to simulate this possible adversarial influence. Since the simulator is assumed to keep track of the adversary's instructions in the real world, whenever the open procedure completes in the simulation, the functionality $\mathcal{F}_{\text{open}}^{\text{Liquid}}$ is triggered. In this procedure, it is required that the open request is agreed upon by the majority of all operators, and that the corresponding on-chain value is correctly committed on the L1. Once all checks pass, the `Internal` state of $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$ is updated accordingly and all participants are notified. The output for the open request is only sent if all the requirements are satisfied, ensuring that the real and

ideal executions are indistinguishable, as long as the assumption of a majority of honest operators holds and the adversary cannot forge signatures.

Protocol state update. Once a client is considered as having joined the sidechain, it can propose to update the state of the Liquid Network protocol by submitting transactions. The adversary may influence this procedure by proposing an invalid block containing incorrect transactions. In the ideal world, we let the simulator simulate all such potential misbehavior. Whenever there is an instruction from the adversary, the simulator acts accordingly within the internal simulation $\mathcal{P}^{\text{Liquid}}$, and triggers the subroutine $\mathcal{F}_{\text{update}}^{\text{Liquid}}$ whenever a state update occurs within the simulation. The subroutine $\mathcal{F}_{\text{update}}^{\text{Liquid}}$ performs checks to validate the proposed block. These checks mirror those carried out in the real protocol. Therefore, the Internal state of the ideal functionality remains synchronized with the real protocol execution, provided that the adversary cannot forge signatures and that the assumption of a majority of honest operators holds. Although this update procedure does not generate direct output to the environment, changes to the Internal state can affect the outputs of other procedures, such as read and settlement, which are discussed in the following sections.

Protocol state read. Here since the data is stored in the operators, we assume a client is connected to all operators, along with the assumption of synchronous communication, it guarantees the client is always able to obtain the latest block and state of the sidechain. To simulate such read method, the ideal functionality $\mathcal{F}_{\text{read}}^{\text{Liquid}}$ will directly output the latest state recorded in StateList, as long as the transaction list stored in ExecutedRequest starting from the LastReadPoint. In that case, the read result in both the ideal world and the real world will be the same.

Protocol state settlement. In the ideal world, the settlement procedure is also triggered through the network connection in order to simulate the malicious parties' possible behaviors, $\mathcal{F}_{\text{settlement}}^{\text{Liquid}}$ checks the settlement transaction is both committed on the sidechain and the L1 ledger. Once these checks pass, all the participants will be notified by $\mathcal{F}_{\text{layers}}^{\text{Liquid}}$ for successful closure. As long as the adversary is not able to forge a signature or a valid transaction itself, the outputs of the real protocol and ideal functionality during the settlement procedure are indistinguishable.

Based on the analysis of all the procedures of the Liquid Network, we can know that the Liquid Network protocol $\mathcal{P}^{\text{Liquid}}$ iUC realizes $\mathcal{F}_{\text{layer2}}^{\text{Liquid}}$. \square

D Case Study: The Arbitrum Nitro Rollup

D.1 Arbitrum Nitro Real Protocol

We here first propose the real-world protocol of the Arbitrum Nitro rollup protocol $\mathcal{P}^{\text{Arbitrum}} = (\text{Client}|\text{Operator}, \text{Validator}, \mathcal{F}_{\text{cert}}, \mathcal{F}_{\text{clock}})$. In the Arbitrum Nitro rollup protocol, we focus on three roles of participants: client, operator, and validator. Here we assume there is at least one honest operator and one honest validator.

D.1.1 Client Functionality. Clients are the users of the rollup protocol; after joining the protocol, they can propose transactions to the operator for execution.

Description of $\mathcal{M}_{\text{client}}^{\text{Arbitrum}}$ of protocol $\mathcal{P}^{\text{Arbitrum}}$
<p>Implement roles: Client</p> <p>Internal state :</p> <ul style="list-style-type: none"> • Round • ExecutedRequest • StateList • OnchainState • Identities <p>Main:</p> <p>Receive {Open, Identities, InitialState} from I/O:</p> <ol style="list-style-type: none"> (1) According to InitialState, prepare the deposit transaction TX_{deposit}, and parse the destination account address Address from Identities; (2) Send {Submit, TX_{deposit}, Address} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$; (3) Send Read to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$, wait for committed in the L1 ledger; (4) Prepare the peg-in transaction $TX_{\text{peg-in}}$ and generate according Sig with $\mathcal{F}_{\text{cert}}$ for signatures; (5) If receive message delivery request from \mathcal{A} through NET, send {Open, $TX_{\text{peg-in}}$, Sig} to the $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}}^{\text{Arbitrum}} : \text{operator})$; (6) Send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$, and check $TX_{\text{peg-in}}$ is included in ReadResult without any fraud-proof; (7) Reply {Open, InitialState} through I/O; <hr/> <p>Receive {Submit, TX} from I/O:</p> <ol style="list-style-type: none"> (1) Obtain Sig for TX from the ideal functions $\mathcal{F}_{\text{cert}}$; (2) Send {Submit, TX, Sig} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}}^{\text{Arbitrum}} : \text{operator})$ after receive the message delivery request from \mathcal{A} through NET; <hr/> <p>Receive {Settlement} from I/O:</p> <ol style="list-style-type: none"> (1) Prepare the peg out transaction $TX_{\text{peg-out}}$ and generates Sig with $\mathcal{F}_{\text{cert}}$; (2) Send {Settlement, $TX_{\text{peg-out}}$, Sig} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{M}_{\text{operator}}^{\text{Arbitrum}} : \text{operator})$ after receive the message delivery request from \mathcal{A} through NET; (3) Send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$; (4) If LatestState is recorded in ReadResult, reply {Settlement, LatestState}; <hr/> <p>Receive {Read} from I/O:</p> <ol style="list-style-type: none"> (1) Send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$; (2) Wait for ReadResult; (3) Rely ReadResult through I/O;
Description of $\mathcal{M}_{\text{operator}}^{\text{Arbitrum}}$ of protocol $\mathcal{P}^{\text{Arbitrum}}$

D.1.2 Operator Functionality. The operator is responsible for executing the received transaction request TX. After the execution, the operator is responsible for publish the requests and final result state to the L1 ledger every T_{period} time, which is triggered by the ideal clock functionality $\mathcal{F}_{\text{clock}}$ with request {Updaterequest}.

Description of $\mathcal{M}_{\text{operator}}^{\text{Arbitrum}}$ of protocol $\mathcal{P}^{\text{Arbitrum}}$

Implement roles: Operator

Internal state :

- Round
- RequestQueue
- Identities

Main:

Receive {Open, $TX_{\text{peg-in}}$, Sig} from I/O:

- (1) Verify the validity of $TX_{\text{peg-in}}$ and Sig with $\mathcal{F}_{\text{cert}}$;
- (2) Send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ and check $TX_{\text{peg-in}}$ is committed;
- (3) If check passes, add $TX_{\text{peg-in}}$ to RequestQueue;

Receive {Submit, TX , Sig} from I/O:

- (1) Verify the validity of TX and Sig with $\mathcal{F}_{\text{cert}}$;
- (2) If check passes, add TX to RequestQueue;

Receive {Settlement, $TX_{\text{peg-out}}$ } from I/O:

- (1) Check the validity of the transaction $TX_{\text{peg-out}}$;
- (2) If the check passes, add settlement request to RequestQueue ;

Receive {Updaterrequest} from I/O:

- (1) Send {write, RequestQueue, ExecuteResult} to the $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$;

D.1.3 Validator Functionality. The validator in Arbitrum Nitro rollup protocol is responsible for verifying the published executed requests whether match up with the resulting state. If there is a mismatch, the validator should propose a fraud-proof to prevent the state update from happening during the challenge period. The ideal clock functionality $\mathcal{F}_{\text{clock}}$ should trigger the validator for verification.

Description of $\mathcal{M}_{\text{validator}}^{\text{Arbitrum}}$ of protocol $\mathcal{P}^{\text{Arbitrum}}$

Implement roles: Validator

Internal state :

- Round

Main:

Receive {UpdateCheck} from I/O:

- (1) Send {Read} request to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$.
- (2) Check the correctness of the newly published RequestQueue and ExecutedResult;
- (3) If check does not pass, send {Submit, FraudProof} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$;

D.2 Arbitrum Nitro Ideal Functionality

After proposing the real protocol, we formally define the ideal functionality for the Liquid Network sidechain based on our framework, the ideal functionality is defined as $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}} = (\mathcal{F}_{\text{layer2}} | \mathcal{F}_{\text{init}}^{\text{Arbitrum}}, \mathcal{F}_{\text{submit}}^{\text{Arbitrum}}, \mathcal{F}_{\text{update}}^{\text{Arbitrum}}, \mathcal{F}_{\text{read}}^{\text{Arbitrum}}, \mathcal{F}_{\text{settlement}}^{\text{Arbitrum}}, \mathcal{F}_{\text{updRnd}}^{\text{Arbitrum}}, \mathcal{F}_{\text{leak}}^{\text{Arbitrum}})$.

D.2.1 Submit Functionality. The submit function handles the requests that would change the protocol status: (1) Open request; (2) State update request that includes a transaction TX ; (3) Settlement request. The $\mathcal{F}_{\text{submit}}^{\text{Arbitrum}}$ to check whether the request sent from

the clients, who are instructed by the environment, is correct in the format and valid according to the current status of the Liquid Network protocol.

Description of $\mathcal{M}_{\text{submit}}^{\text{Arbitrum}}$ of subroutine $\mathcal{F}_{\text{submit}}^{\text{Arbitrum}}$

Implement roles: Submit

Main:

Receive {Submit, request, Internal state} from I/O:

- (1) Check the received request is within one of the three valid types:
 - Open request: {Open, InitialState, Identities};
 - State update request: {Submit, TX };
 - Settlement request: {Settlement};
- (2) Check the received request is within one of the three valid types:
 - Open request: {Open, InitialState, Identities};
 - State update request: {Submit, TX };
 - Settlement request: {Settlement};
- (3) Check the open request has been executed or request is the open request;
- (4) Check there is no settlement request from the caller party;
- (5) Wait for $\mathcal{F}_{\text{leak}}^{\text{Arbitrum}}$ decides the leakage;
- (6) If all checks pass, return {Submit, True, leakage};

D.2.2 Open Functionality. Once triggered by the simulator, the open subroutine will help to check whether the peg-in transaction and the initial state that is proposed by the honest client are committed on the L1 ledger.

Description of $\mathcal{M}_{\text{open}}^{\text{Arbitrum}}$ of subroutine $\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$

Implement roles: Open

Main:

Receive {Open, InitialState, Attachment= $TX_{\text{peg-in}}$, Internal state} from I/O:

- (1) Send Read request to $\mathcal{F}_{\text{ledger}}$ and check both InitialState, $TX_{\text{peg-in}}$ are committed on L1 ledger;
- (2) Wait for $\mathcal{F}_{\text{leak}}^{\text{Liquid}}$ decides the leakage;
- (3) If all checks passes, reply {Open, True, leakage} to I/O;

LEMMA D.1. *The subroutine $\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$ guarantees correct initialization and $((n_{\text{OP}} - 1) + \text{Layer}_1, T_{L_2} + T_{L_1})$ -liveness for the open request.*

PROOF. *Correct initialization* is considered violated if the initialization state or the corresponding request is not properly committed on the L1 ledger, while $\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$ still generates a positive output. However, $\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$ only produces output when all required data is correctly committed on the L1 ledger. Besides, as long as the transaction can not be forged by forging a signature and there exists an honest validator for any incorrect published on the L1 ledger from the operator, it will be invalidated by fraud-proof. Thus, *correct initialization* is guaranteed.

Liveness is considered violated if $\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$ fails to generate a positive output under any corruption level or time delay. However,

since $\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$ relies on the publishment from the operator on L1 ledger to generate positive output, thus as long as the L1 ledger guarantees liveness and there exists one honest operator, $\mathcal{F}_{\text{open}}^{\text{Arbitrum}}$ will eventually generate a positive output within the bounded delay of $T_{L_2} + T_{L_1}$. Therefore, *liveness* for the open request is also guaranteed. \square

D.2.3 Update Functionality. To update the protocol state, the update subroutine will check on the L1 ledger whether the published new state is correctly computed based on the executed requests that should also be published on the L1 ledger. If the punishment is correct, the update finality generates positive output to update the `Interan1` state accordingly. If the publishment is incorrect, there should also be a corresponding fraud-proof.

Description of $\mathcal{M}_{\text{update}}^{\text{Arbitrum}}$ of subroutine $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$
<p>Implement roles: Update Main:</p> <p>Receive {Update, NewState, Attachment= {RequestBatch}, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Send {Read} to $\mathcal{F}_{\text{ledger}}$ to check the NewState is committed on-chain; (2) Check all the requests in Attachment and the new state in the NewState is correctly executed. (3) If the check passes, wait for $\mathcal{F}_{\text{leak}}^{\text{Arbitrum}}$ decides the leakage, and reply {Update, True, NewState, leakage} (4) If execution is incorrect, check FraudProof is published, if so, reply {Update, False, leakage}; (5) If no fraud-proof published, HALT;

LEMMA D.2. $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ guarantees $((n_{\text{OP}} - 1) + \text{Layer}_1, T_{L_2} + T_{L_1})$ -liveness for update requests.

PROOF. The *liveness* for update requests is considered violated if $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ fails to generate a positive output under any corruption status or time delay. However, as long as the liveness of the L1 ledger is guaranteed and there exists one honest operator honestly published on the blockchain periodically, $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ —which verifies the publishment of the updated state by checking with the L1 ledger—will be able to generate a positive output. Therefore, *liveness* for update requests is guaranteed. \square

D.2.4 Read Functionality. In the Arbitrum Nitro rollup protocol, instead of deciding the read result based on the `Internal` state stored by the $\mathcal{F}_{\text{layer2}}$ machine, the read result will be the output of sending read request to the underlying L1 ledger functionality $\mathcal{F}_{\text{ledger}}$.

Description of $\mathcal{M}_{\text{read}}^{\text{Arbitrum}}$ of subroutine $\mathcal{F}_{\text{read}}^{\text{Arbitrum}}$
<p>Implement roles: Read Main:</p> <p>Receive {Read, Internal state} from I/O:</p>

- (1) Send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L_1})$, wait for `ReadResult={LatestState, TransactionData}`;
- (2) If reconstruct LatestState from LatestState based on TransitionData, then reply {Read, ReadResult};
- (3) Else reply {Read, ReadResult = \perp };

LEMMA D.3. The subroutines $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ and $\mathcal{F}_{\text{read}}^{\text{Arbitrum}}$ guarantee (Layer1)-safety.

PROOF. The *safety* property is considered violated if there exist conflicting read results among honest parties. However, $\mathcal{F}_{\text{read}}^{\text{Arbitrum}}$ obtains its output by querying the underlying L1 ledger. Therefore, as long as the safety of the L1 ledger is guaranteed—meaning both self-consistency and view-consistency hold for the underlying ledger—the read outputs will be consistent, and thus, *safety* is guaranteed. \square

D.2.5 Settlement Functionality. Once triggered, the settlement function verifies whether the peg-out transaction has been executed and committed on the L1 ledger and whether the latest state of the L2 protocol has also been correctly committed on the L1 ledger.

Description of $\mathcal{M}_{\text{settlement}}^{\text{Arbitrum}}$ of subroutine $\mathcal{F}_{\text{settlement}}^{\text{Arbitrum}}$
<p>Implement roles: Settlement Main:</p> <p>Receive {Settlement, Attachment= $TX_{\text{peg-out}}$, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Find the LatestState from Internal state; (2) Send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L_1})$ and check $TX_{\text{peg-out}}$, LatestState are committed without any FraudProof; (3) Wait for $\mathcal{F}_{\text{leak}}^{\text{Arbitrum}}$ decides the leakage; (4) If check passes, reply {Settlement, True, LatestState} to I/O;

LEMMA D.4. $\mathcal{F}_{\text{settlement}}^{\text{Arbitrum}}$ guarantees correct settlement and $(\text{Layer}_1, T_{L_2} + T_{L_1})$ -liveness for settlement requests.

PROOF. The *correct settlement* property is considered violated if a settlement request committed on the L1 ledger results in a final committed value that differs from the submitted request and is not the latest valid state, yet still triggers a positive output. However, $\mathcal{F}_{\text{settlement}}^{\text{Arbitrum}}$ generates a positive output only after verifying that the state committed on the L1 ledger is consistent with the submitted request and corresponds to the latest valid state. Therefore, as long as the L1 ledger is secure and valid transactions can not be forged by forging signatures, the *correct settlement* is guaranteed.

For *liveness*, since $\mathcal{F}_{\text{settlement}}^{\text{Arbitrum}}$ only needs to check the state committed on the L1 ledger to generate a positive output, and clients are allowed to publish the transaction to L1 ledger by itself, the liveness of the settlement request depends solely on the liveness of the underlying blockchain. Thus, as long as the L1 ledger provides liveness, the settlement request will be processed within time $T_{L_2} + T_{L_1}$, and the *liveness* is guaranteed. \square

D.2.6 Update Round Functionality. Although the Arbitrum Nitro protocol does not assume any bound on communication latency between clients and operators, the operator is expected to publish a batch of executed requests and results every T_{period} .

Description of $\mathcal{M}_{\text{updRnd}}^{\text{Arbitrum}}$ of subroutine $\mathcal{F}_{\text{updRnd}}^{\text{Arbitrum}}$
<p>Implement roles: UpdateRound Main:</p> <p>Receive {UpdateRound, Internal state} from I/O:</p> <ol style="list-style-type: none"> (1) Every T_{period} time, send {Read} to $(\text{pid}_{\text{cur}}, \text{sid}_{\text{cur}}, \mathcal{F}_{\text{ledger}} : \text{Client}_{L1})$ and check if there any publishement from operator; (2) If the check passes, reply {UpdateRound, True};

D.3 Security Proof

After proposing the ideal functionality and real-world implementation, we now show the security of the Arbitrum Nitro rollup protocol. We first prove that the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ guarantees all the security properties.

THEOREM 4.5. *The ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ guarantees all the security properties of a secure L2 protocol.*

PROOF. According to Lemma D.1 to D.4, we can conclude that $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ guarantees all the security properties. \square

Our main proof strategy is to show that there exists a simulator for the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ that internally simulates an alternative version of the real-world protocol $\mathcal{P}^{\text{Arbitrum}}$ and interacts with the ideal functionality in a way that produces the same outputs to the environment. This ensures that the adversary's view in the real world is indistinguishable from that in the ideal world.

Here we assume there is a simulator \mathcal{S} that internally simulates the same protocol of $\mathcal{P}^{\text{Arbitrum}}$, which is noted with $\mathcal{P}'^{\text{Arbitrum}}$. The simulator \mathcal{S} behaves according to the following rules:

Network connection with ideal functionality and environment.

- Any message the simulator \mathcal{S} received from the environment through the network connection, which can be understood as the adversary instruction, will be forwarded to the internal simulation $\mathcal{P}'^{\text{Arbitrum}}$.
- Any message sent from the internal simulation $\mathcal{P}'^{\text{Arbitrum}}$ to the network connection, which can be seen as leakage to the environment, will be forwarded to the environment by the simulator.
- Whenever there is a request for open protocol, update state, or settlement state inside the simulation, the simulator will trigger the ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ to synchronize the change.

Corruption handling.

- The simulator will keep track of the corrupt entities, and keep the corrupted parties in $\mathcal{P}'^{\text{Arbitrum}}$ and $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ synchronized.
- The simulator will forward the messages from the corrupted entities in the ideal functionality to the adversary

Message delivery. For the Liquid Network, we assume the communication among participants in Arbitrum Nitro is asynchronous; thus, the message delivery will be eventually decided by the adversary. Here, we let the simulator bookkeep all the messages in the simulation and trigger message delivery the same as the real-world protocol.

Request submission. Whenever an honest entity in ideal functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ receives a request from the environment after it gets accepted decided by the subroutine $\mathcal{F}_{\text{submit}}^{\text{Arbitrum}}$, it will forward the transaction to the simulator \mathcal{S} . After receiving the forwarded request from the $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$, \mathcal{S} will send the request to its internal simulated protocol $\mathcal{P}'^{\text{Arbitrum}}$.

Rollup joining. The simulator \mathcal{S} monitors both the internal simulation state and the L1 ledger. Whenever it detects that the open procedure has completed in $\mathcal{P}'^{\text{Arbitrum}}$, it prepares a request to be sent to $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ for validation. This request includes the InitialState used to initialize the protocol and the corresponding Attachment, which contains the peg-in transaction that must be published on the L1 ledger, as generated using $\mathcal{F}_{\text{cert}}$ within the simulation $\mathcal{P}'^{\text{Arbitrum}}$.

Protocol state update. The simulator \mathcal{S} monitors the internal simulation and the L1 ledger for state updates. Once \mathcal{S} detects that a state update has been published on the L1 ledger, the challenging time period T_{period} ends. It prepares an update request and sends it to $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ to synchronize the internal state. This request includes the updated state and an Attachment containing the executed requests, along with signatures generated via $\mathcal{F}_{\text{cert}}$.

Protocol state read. $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ directly gets the read result by querying the underlying $\mathcal{F}_{\text{ledger}}$, \mathcal{S} will not influence the read result.

Protocol state settlement. After receiving the settlement request from $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$, the simulator \mathcal{S} executes the settlement procedure as defined in the real protocol for all non-corrupted entities. \mathcal{S} continuously monitors the simulation, and once the settlement procedure is completed, it prepares a request containing the Attachment that includes the valid peg-out transaction generated with $\mathcal{F}_{\text{cert}}$ that needs to be published on the L1 ledger, and sends this request to $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ for checking.

Further details. The simulator \mathcal{S} also maintains synchronization of the internal clock/round with the ideal functionality. Whenever \mathcal{S} is instructed by the adversary \mathcal{A} to advance the round, it sends an UpdateRound request to $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$ and triggers the corresponding round update.

Then, we prove the security of the protocol by showing that for each procedure, the environment can not distinguish between the output of ideal functionality and the output of the real protocol.

THEOREM 4.6. *Let $\mathcal{F}_{\text{ledger}}$ be the idealized L1 ledger functionality, $\mathcal{F}_{\text{cert}}$ be the idealized functionality for EUF-CMA secure signature scheme. Then, the real Liquid Network sidechain protocol $\mathcal{P}^{\text{Arbitrum}}$ iUC-realizes the ideal Arbitrum Nitro rollup functionality $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$.*

PROOF. We prove it by showing the outputs of each procedure of the protocol in both worlds are indistinguishable.

Request submission. For any received request from the environment, the subroutine $\mathcal{F}_{\text{submit}}^{\text{Arbitrum}}$ will check whether the format is

correct. The accepted requests are forwarded to the simulator for the following simulation. Since the real-world protocol and the ideal functionality accept the same types of requests, there are no differences to distinguish both worlds.

Rollup join. In the real protocol, the adversary could influence the joining procedure by deviating from the expected steps. However, since both the ideal functionality and the real protocol perform the same checks, the output for client joining remains consistent across both worlds. Additionally, the adversary may delay the communication between the joining client and the operator, causing the joining procedure not to be completed immediately. We simulate such adversarial influence by allowing the simulator to trigger the corresponding check in the ideal functionality once the joining procedure is completed in its internal simulation—either when the adversary permits the message to be delivered, or when the client independently submits the peg-in transaction to the L1 ledger. Therefore, the outputs of the rollup joining procedure in both the real and ideal worlds are indistinguishable.

Protocol state update. To begin with, in both the ideal and real worlds, state update proposals are periodically published on the L1 ledger; otherwise, the protocol round does not proceed. The adversary may attempt to influence this procedure by publishing an invalid batch of requests on the L1 ledger. In the real protocol, as long as there exists at least one honest validator, a fraud-proof will be submitted, rendering the proposed state update invalid. In the ideal world, since the simulator internally runs the same protocol and all previously accepted requests by $\mathcal{F}_{\text{layer2}}$ are forwarded to the simulator, the adversary learns the same leakage and can publish the same invalid batch. Similarly, the simulator can simulate the response of an honest validator by generating a fraud-proof and triggering $\mathcal{F}_{\text{update}}^{\text{Arbitrum}}$ to perform the corresponding check, which will result in a rejection and no state change. Therefore, the update procedure remains indistinguishable between the two worlds.

Protocol state read. In both the ideal world and the real world, the read result is directly fetched from the L1 ledger. Therefore, as long as the update procedure ensures that the state transitions in both the real and ideal world protocols are consistent, the outputs for read requests will be indistinguishable.

Protocol state settlement. Similar to the open procedure, the same checks for settlement are conducted in both the real and ideal worlds. Any incorrect settlement request will either not be committed on the L1 ledger or will be invalidated via a fraud-proof, resulting in the same output behavior in both worlds. On the other hand, the adversary can delay the settlement request from the client to the operator. This delay is simulated in the ideal world by allowing the simulator to emulate the communication delay, without immediately forwarding the request to the ideal functionality. Once the simulator determines that the settlement is successfully completed in its internal simulation, it triggers the ideal functionality to perform the corresponding checks and generate a positive output, consistent with the behavior of the real-world protocol. Consequently, the outputs of the settlement procedure in both worlds are indistinguishable.

Based on the analysis of all the procedures of the Arbitrum Nitro rollup protocol, we know that the Arbitrum Nitro rollup protocol $\mathcal{P}^{\text{Arbitrum}}$ iUC realizes $\mathcal{F}_{\text{layer2}}^{\text{Arbitrum}}$. \square

E Missing Proofs of Comparative Analysis

THEOREM 5.4. *A secure PCF and sidechain protocol can only realize $f_{\text{OP-safety}}$ and $\{\lfloor \frac{n_{\text{OP}} - (f_{\text{OP}} + 1)}{2} \rfloor, T_{L_2}\}$ -liveness for state update requests.*

PROOF. To begin, observe that in PCF and sidechain-based protocols, all execution occurs off-chain. Consequently, the liveness latency is determined solely by the off-chain environment and is always T_{L_2} .

Suppose the protocol simultaneously realizes $f_{\text{OP-safety}}$ and

$$\left\{ \left\lfloor \frac{n_{\text{OP}} - (f_{\text{OP}} + 1)}{2} \right\rfloor + 1, T_{L_2} \right\}\text{-liveness.}$$

This implies that to execute a request, it suffices to obtain agreement from only

$$n_{\text{OP}} - \left\lfloor \frac{n_{\text{OP}} - (f_{\text{OP}} + 1)}{2} \right\rfloor - 1$$

operators.

Assume the adversary corrupts f_{OP} operators. Then, by coordinating with up to half of the remaining honest operators, the adversary can potentially form a coalition of size

$$f_{\text{OP}} + \left\lfloor \frac{n_{\text{OP}} - f_{\text{OP}}}{2} \right\rfloor.$$

We now show that:

$$f_{\text{OP}} + \left\lfloor \frac{n_{\text{OP}} - f_{\text{OP}}}{2} \right\rfloor \geq n_{\text{OP}} - \left\lfloor \frac{n_{\text{OP}} - (f_{\text{OP}} + 1)}{2} \right\rfloor - 1.$$

Let $x := n_{\text{OP}} - f_{\text{OP}} > 0$. Then the inequality becomes:

$$\left\lfloor \frac{x}{2} \right\rfloor + \left\lfloor \frac{x-1}{2} \right\rfloor \geq x-1.$$

This identity holds for all integers $x > 0$, with equality:

$$\left\lfloor \frac{x}{2} \right\rfloor + \left\lfloor \frac{x-1}{2} \right\rfloor = x-1.$$

Therefore, the adversary is capable of gathering sufficient votes to satisfy the liveness condition. This implies that the adversary could cause inconsistent state transitions, contradicting the assumption that the protocol realizes $f_{\text{OP-safety}}$.

Thus, there exists a fundamental trade-off between liveness and safety under adversarial corruption thresholds. \square

THEOREM 5.5. *A secure rollup protocol can only realize L1-Safety and $\{(n_{\text{OP}} - 1) + \text{Layer 1}, T_{L_2} + T_{L_1}\}$ -Liveness for state update requests.*

PROOF. We prove this by contradiction. Assume that a secure rollup protocol realizes *Layer 1-safety* while also achieving a stronger liveness tolerance of $n_{\text{OP}} + \text{Layer 1}, T_{L_1}$ -liveness. This implies that even if all operators are corrupted, the protocol should still guarantee that a request from an honest client will eventually be executed within time T_{L_1} . However, in practice, the execution of any request requires it to be published on the L1 ledger by at least one operator. If all operators are corrupted, they can simply ignore the request from the honest client, thereby preventing it from being published and executed. This violates the liveness property, contradicting the assumption. Hence, such a liveness guarantee is not achievable under full operator corruption. \square

THEOREM 5.6. *Assume m state update requests are executed after the protocol starts. To guarantee liveness for secure PCF protocol and sidechain protocol, the Data availability needs to have $\{\Omega(m), \Omega(n_p)\}$ efficiency.*

PROOF. To start with, since all the n participants are required to publish their joining transactions on the L1 for the protocol to begin, the L1 storage efficiency must be at least $\Omega(n_p)$.

For the L2 storage efficiency, we prove it by contradiction. Assume that *liveness* still holds even if fewer than m executed requests are stored off-chain. In that case, there must exist at least one request that cannot be retrieved through the read interface at a certain round based solely on the L2 storage. Since PCF and sidechain protocols do not retrieve execution data from the L1, the missing request would violate the definition of *liveness*. Thus, the assumption leads to a contradiction, and storing fewer than m requests off-chain breaks *liveness*. \square

THEOREM 5.7. *Assume m state update requests are executed after the protocol starts. To guarantee liveness for secure rollup protocol, the Data availability needs to have $\{\Omega(1), \Omega(m) + \Omega(n_p)\}$ efficiency.*

PROOF. The L1 storage efficiency consists of two parts. The first part is the $\Omega(n_p)$ storage requirement for the participants' joining requests, all of which are published on the L1.

For the second part, assume that *liveness* still holds for state update requests in a rollup protocol even if fewer than m requests are stored on the L1. Since the rollup protocol's read result is derived directly from the L1, this would imply that at least one request cannot be retrieved by a read query at a certain round. This contradicts the definition of *liveness*, and therefore the assumption must be false. Thus, storing fewer than m requests on the L1 would break *liveness*, and the L1 storage efficiency must also be at least $\Omega(m) + \Omega(n_p)$ in total. \square