A Secure Sequencer and Data Availability Committee for Rollups

Margarita Capretto
margarita.capretto@imdea.org
IMDEA Software Institute
Pozuelo de Alarcón, Spain
Universidad Politécnica de Madrid
Madrid, Spain

Martín Ceresa
martin.ceresa@iohk.io
Input Output
Madrid, Spain

Antonio Fernández Anta
antonio.fernandez@imdea.org
IMDEA Software Institute
Pozuelo de Alarcón, Spain
IMDEA Network Institute
Madrid, Spain

Pedro Moreno-Sanchez
pedro.moreno@imdea.org
IMDEA Software Institute
Pozuelo de Alarcón, Spain
MPI-SP
Bochum, Germany
Visa Research
Foster City, US

César Sánchez
cesar.sanchez@imdea.org
IMDEA Software Institute
Pozuelo de Alarcón, Spain

Abstract

Blockchains face a scalability limitation, partly due to the throughput limitations of consensus protocols, especially when aiming to obtain a high degree of decentralization. Layer 2 Rollups (L2s) are a faster alternative to conventional blockchains. L2s perform most computations offchain using minimally blockchains (L1) under-the-hood to guarantee correctness. A *sequencer* is a service that receives offchain L2 transaction requests, batches these transactions, and commits compressed or hashed batches to L1. Using hashing needs less L1 space—which is beneficial for gas cost—but requires a data availability committee (DAC) service to translate hashes into their corresponding batches of transaction requests. The behavior of sequencers and DACs influence the evolution of the L2 blockchain, presenting a potential security threat and delaying L2 adoption.

We propose in this paper fraud-proof mechanisms, arbitrated by L1 contracts, to detect and generate evidence of dishonest behavior of the sequencer and DAC. We study how these fraud-proofs limit the power of adversaries that control different number of sequencer and DACs members, and provide incentives for their honest behavior. We designed these fraud-proof mechanisms as two player games. Unlike the generic fraud-proofs in current L2s (designed to guarantee the correct execution of transactions), our fraud-proofs are over pre-determined algorithms that verify the properties that determine the correctness of the DAC. Arbitrating over concrete algorithms makes our fraud-proofs more efficient, easier to understand, and simpler to prove correct. We provide as an artifact a mechanization in LEAN4 of our fraud-proof games, including (1) the verified strategies that honest players should play to win all games as well as (2) mechanisms to detect dishonest claims.

CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Blockchain, Layer 2, Fraud Proof, Incentives, LEAN4

ACM Reference Format:
Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Pedro Moreno-Sanchez, and César Sánchez. 2025. A Secure Sequencer and Data Availability Committee for Rollups. In *Proceedings of the 2025 ACM SIGSAC Conf. on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765187

1 Introduction

Distributed ledgers (also known as *blockchains*) were first proposed by Nakamoto in 2009 [34] in the implementation of Bitcoin, as a method to eliminate trusted third parties in electronic payment systems. A current major obstacle for a faster widespread adoption of blockchain technologies in some application areas is the limited scalability of smart contract-enabled blockchains. This is due to (1) the limited throughput of Byzantine consensus algorithms [17, 42], and (2) the limitation in the block size due to the desire of decentralization. Issue (1) limits the number of blocks per second, while the decentralized validation limits the size of the blocks [15]. For example, Ethereum [44]—one of the most popular blockchains—is limited to less than 4 blocks per minute, each containing less than two thousand transactions.

Layer 2 (L2) rollups provide a faster alternative to blockchains, like Ethereum, while still offering the same interface in terms of smart contract programming and user interaction. L2 rollups seek to perform as much computation as possible offchain with the minimal blockchain interaction—in terms of the number and size of invocations—required to guarantee a correct and trusted operation. L2 rollups work in two phases. In the first phase, users inject transaction requests communicating with a service called a *sequencer*,

CCS '25, Taipei

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765187

which orders the transaction requests and packs them into batches. After creating a batch, the sequencer compresses the batch and injects the result into the underlying blockchain (L1). Once the batch is posted to L1 the transaction order is determined. In the second phase, the effects of executing transaction batches are computed offchain by agents called State Transition Functions (STFs). STFs are independent parties that compute L2 blocks from batches and post the resulting state (that is, the state of the L2 blockchain) into L1. There are two main categories of L2 rollups:

- *ZK-Rollups:* STFs post zero-knowledge proofs that encode the correctness of the result obtained after computing the transactions in the batch. These proofs are verified by the L1 contract. Upon successful validation the new L2 block consolidates.
- *Optimistic Rollups:* STFs post L2 blocks which are optimistically assumed to be correct, delegating block validation on fraud-proof mechanisms.

The most prominent Optimistic Rollups based on their market share [37] are Arbitrum One [2], Base [14] and OP mainnet [36]. Popular ZK-Rollups include Starknet [40] and zkSync Era [31].

Optimistic Rollups include an arbitration process to solve *disputes*. STFs place a stake when they propose a new L2 block into L1. Since L2 blocks could be incorrect (that is, STFs could maliciously encode an incorrect outcome of the executed transactions), competing STFs are given a fixed interval of time to challenge proposed blocks. When an STF challenges a block it also places a stake. The arbitration process is a game governed by an L1 contract, which is played between competing STFs. The game consists on bisecting the execution trace of transactions in the disputed L2 block until a dispute over a single instruction is reached, which can be directly verified in L1. In other words, the game arbitrates over the finite execution of an arbitrary program (the sequence of L2 contracts executed). The arbitration process ensures that a single honest participant can always win the dispute, if it plays properly. The losing party loses the stake and the winner receives a portion of the loser's stake as a compensation. Note that winning a game does not necessarily mean that the winner player is right (as a player can also play incorrectly or stop playing and lose on purpose). Therefore, L2 blocks in optimistic rollups consolidate when it has stakes after all challenges end. This guarantees that a single honest participant can enforce that the L2 blockchain only contains honest blocks.

To increase scalability even further, the sequencer in some L2 rollups posts hashes of batches—instead of compressed batches—dramatically reducing the size of the L1 blockchain interaction. However, using hashes to encode batches requires an additional data service—called *data availability committee* (DAC)—to translate hashes back into their corresponding batches.

ZK-Rollups that rely on DACs are known as *Validiums*, which include Sophon [39] and Lens [30]. Optimistic Rollups that use DACs are called *Optimiums*, such as Arbitrum Nova [2]. [1]

To simplify notation, in the rest of the paper we simply use L2 to refer to either Optimistic Rollups or ZK-Rollups that post hashes and have a DAC, (i.e. we use L2 to refer to Optimiums and Validiums). Following [10], *we use the term **arranger** to refer to the combined service formed by the sequencer and the data availability committee.* Therefore, the arranger service receives transaction requests, creates batches containing many requests, commits the hash of these batches to L1, and is responsible for translating the hashes back into batches upon request. Once the arranger has posted batches and translates their contents, the STFs can proceed to compute the effects of executing the transactions in the batch. That is, arrangers are common to Optimiums and Validiums because they do not execute transactions, which is where these L2s differ.

Arrangers have the power to influence both liveness and safety of the L2. For example, arrangers can ignore transactions or users, fail to post hashes or provide data that does not correspond to batches of valid transaction requests and collude with STFs to post invalid L2 blocks that are indisputable. Arrangers can also try to delay the L2 by not replying to translations requests. In order to prevent censorship of transaction requests, some L2s provide mechanisms to bypass the sequencer and add transaction requests directly in L1. However, to the best of our knowledge, there are no mechanism to detect and prevent in L1 safety violations from arrangers.

The Problem. *In this paper we attack the problem of reducing the power of arrangers over the evolution of L2s when trust assumptions are violated.*

Our Solution (overview). *We provide a collection of fraud-proof mechanisms to detect violations of safety properties and to enforce the correct evolution of the L2, and incentives to promote correct behavior.*

We adopt an open permissioned model [16] where permissionless L2 users can issue transaction requests to the permissioned arranger servers (which we call *replicas*). This model can also be adapted to a permissionless setting with committee sortition [22] without significant modifications. We consider a refinement of the Byzantine failure model [28], in which there are two types of faulty replicas: *Byzantine* replicas and *corrupt* replicas. Non faulty replicas are called *honest*. The difference between Byzantine and corrupt replicas is that Byzantine replicas can behave arbitrarily at all points in time, while corrupt replicas cannot interfere with honest replicas in the agreement of batches. Corrupt replicas can misbehave in other ways, for example, they can sign invalid batches or refuse to translate hashes. Essentially, corrupt replicas are less powerful than Byzantine replicas. This distinction between Byzantine and corrupt replicas allows us to study adversarial models with varying degrees of power and evaluate their impact on the properties guaranteed by our system. In particular, we analyze three different adversarial models. The simplest adversary does not violate the trust assumptions of the arranger, and thus the arranger remains correct and all safety and liveness properties hold. In this case, only a fraud-proof mechanism to discard batches without enough signatures is used. At the other extreme, we consider an adversary which can control all arranger replicas. In theory, this compromised arranger offers no guarantees. However, with our fraud-proofs any honest agent can ensure that only correct batches consolidate, and prove in L1 violations to safety properties, exposing that the arranger is compromised. Still, if the compromised arranger satisfies all safety properties but violates liveness, it cannot be detected or proved in L1. Finally, we examine an adversary whose power sits in the middle of the previous two. This adversary controls enough replicas

[1] See [37] for a complete list of L2 rollups on top of Ethereum.

to generate batches with the required signatures, but honest replicas can also agree on batches and have these batches signed with enough signatures, because not all adversary controlled replicas are Byzantine, some are just corrupt. In this case, even if all batches posted by the malicious replicas controlled by the adversary are "correct" but differ from the batches agreed by the honest replicas, honest replicas can expose that the arranger is compromised.

First, we propose fraud-proof mechanisms, based on Refereed Delegation of Computation (RDoC) [8, 9], to prove that a batch posted by the arranger is incorrect or unavailable. If a batch or hash is proven incorrect, all involved replicas lose their stake and the batch is discarded (along with all L2 blocks executing its transactions). Additionally, our fraud-proof mechanisms generate undeniable proofs of fraud, which can be used as evidence for demanding the replacement of faulty replicas. More importantly, a single honest agent (with enough resources to pay the L1 fee to play the game) can use our fraud-proof mechanisms to enforce that incorrect or unavailable batches do not consolidate, ensuring that all L2 blocks can be computed and disputed, *even if all arranger replicas are faulty.* That is, our fraud-proof mechanisms can be used (1) to guarantee safety properties of arrangers even when the trust assumptions are violated and also (2) as a deterrent for replicas from being faulty. As a result, we obtain a L2 solution where a single honest agent can guarantee safety of the entire system, not just the execution part (where the STFs compute the effects), as is the case in current L2s.

We modeled our solution in LEAN4, and prove that a single honest agent is enough to prevent faulty assertions. LEAN4 is a proof-assistant aimed to close the gap between automated and interacting theorem proving [33]. The mechanization involves modeling assertions, the possible actions agents can take, and mainly, the construction of fraud-proofs when faulty assertions are detected. We can directly run our verified strategies that honest agents can use to win all challenges against faulty agents.

It is important to note that, unlike fraud-proof mechanisms in current L2s, we propose fraud-proofs over pre-determined algorithms which verify specific properties. Fraud-proofs in current L2s are over the execution trace of arbitrary algorithms (as transactions can invoke arbitrary contracts during their execution) and require: (1) reasoning about the execution of an interpreter of smart contracts and (2) the ability to extract the state after each instruction, not just the final result, to provide these states in the trusted L1 arbitrator contract as required by the game. In contrast, our fraud-proofs exploit that we arbitrate over pre-determined algorithms, dividing the execution of these algorithm into well defined high-level blocks. This approach has the following advantages when compared to fraud-proofs for arbitrary algorithms:

- Modularity and clarity: as each building block is well-defined, each move and resulting position in the game is easier to understand and formalize.
- Efficiency: our fraud-proofs can process blocks of instructions instead of having to reason at the instruction level, which reduces the total number of moves of the game.
- Formality: the simplicity of our fraud-proofs, compared to those currently used in L2s, enabled us to formalize and prove the correctness of our games in a LEAN4 library of around 5000 lines of code. As far as we know, fraud-proofs used in L2s have not been formalized yet and would likely be a much more complex endeavor.

Although fraud-proofs are meant to be a deterrent, only to be executed in the worst case scenario, an incorrect implementation may lead to the existence of fraud-proofs that invalidate correct blocks or the impossibility to generate fraud-proofs of incorrect blocks, rendering the whole L2 scheme incorrect. To reason about fraud proofs clear and modular approaches are needed.

Second, we introduce *incentives* for arranger replicas to participate in the protocol, e.g. including payments for generating and signing hashes, posting correct batches into L1, and performing reverse translations. These replicas place stakes when posting batch hashes and are rewarded for batches that consolidate.

By combining incentives with fraud-proof mechanisms, we create a motivation for rational agents to behave honestly. To the best of our knowledge, this is the first work to introduce incentives and fraud-proof mechanisms for arrangers (sequencer and DACs) of L2s. Our incentives and fraud-proof mechanisms are general and do not depend on concrete implementation of the arranger or how the execution part is handled by the STFs. Therefore, current L2s can easily integrate our protocol. L2s just need to (1) replace the existing contract that receives batches from their arranger and (2) deploy new contracts to govern the fraud-proofs of our protocol. These new fraud-proofs focus on properties of the batches posted by the arranger, rather than transaction execution, ensuring that they do not conflict with existing fraud-proofs.

Contributions. In summary, the contributions of this paper are:

(1) Novel fraud-proofs over pre-determined algorithms which verify specific arranger properties in Sections 5;
(2) Economic incentives including payments and fraud-proof mechanisms to detect protocol violations and punishments for faulty replicas in Sections 5 and 6;
(3) An analysis of three adversary models and their limitations and impacts on the evolution of the L2 blockchain in Section 7;
(4) One artifact [12]: a library in LEAN4 mechanizing fraud-proof mechanisms proving that honest players always win.

Structure. The rest of the paper is organized as follows. Section 2 states our assumptions about L1s, presents Optimistic Rollups and Optimiums, describes the computation model, and briefly presents Merkle trees. Section 3 describes the concept of arranger. Section 4 gives an overview of our proposal to limit the power of arrangers using: (1) fraud-proof mechanisms, explained in detail in Section 5, and (2) economic incentives, studied in Section 6. Section 7 presents three threat models and analyzes their impact in the evolution of L2s. Section 8 compares with related work. Finally, Section 9 concludes.

2 Definitions. Model of Computation

We state now our assumptions about L1s, briefly present Optimistic Rollups and Optimiums, describe the computation model and present an overview of Merkle trees.

2.1 Assumptions about L1

We assume that the L1 ensures both liveness and safety. Specifically, while the system tolerates temporary censorship of L1 transaction

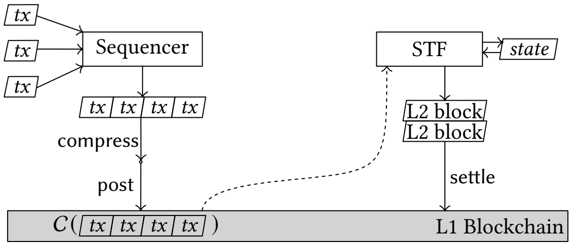

Fig. 1: Optimistic Rollups.

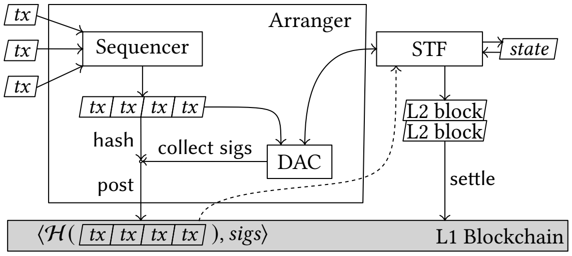

Fig. 2: Optimiums.

requests and reordering of L1 transactions, it guarantees that every transaction submitted to L1 is eventually processed correctly. The L1 includes the following specific smart contracts:

- a **logger** that arranger replicas use to post batches of transaction requests, and
- a set of smart contracts that arbitrate fraud-proof mechanisms (see Section 5).

2.2 L2 Optimistic Rollups and Optimiums

L2 Optimistic Rollups split transaction sequencing from transaction execution. The consolidation of transaction effects is delayed to allow disputes and arbitration. L2 Optimistic Rollups are implemented as two components (see Fig. 1): a *sequencer*, in charge of ordering transactions, and a *state transition function* (STF), responsible for executing transactions (terms introduced by Arbitrum [25]). While currently the role of the sequencer is centralized, anyone can in principle be an STF.[2]

The sequencer collects transaction requests from L2 users, packs them into batches, and posts these batches as a *single* invocation into L1. Once posted, compressed batches are immutable and visible to everyone. Currently, sequencers are centralized processes and do not offer any guarantee regarding transaction orders, whether transaction requests can be discarded or that the data posted in L1 actually correspond to a batch of transaction requests. To prevent censorship from the sequencer, some L2s allow users to perform a more expensive posting of transaction requests straight to L1, which is visible and must be executed.

STFs are independent processes in charge of computing the effects of batches of transaction requests, which is determined by the sequence of transaction requests and the previous L2 state. STFs propose new L2 states as L2 block *assertions* into L1. L2 blocks are optimistically assumed to be correct, but there is a challenge period of typically a week. STFs place stakes in L2 blocks, asserting that the L2 block is correct, and can also place stakes to challenge L2 block assertions posted by other STFs. When STFs challenge assertions, a fraud-proof mechanism is played in L1 between the involved STFs. The mechanism ensures that honest players win. The losing party forfeits their stake, while the winner receives a portion of the loser's stake as compensation. L2 block assertions without stakes are removed, while surviving ones consolidate and the L2 evolves. These fraud-proof mechanisms are games that bisect the execution trace of the transaction in the disputed L2 block, until the challenge is reduced to a single instruction. At this point, players agree on the state before the instruction but disagree on the state after the instruction. During this game, all states in the middle of challenged subtraces and the final instruction must be posted in L1.

Sequencers can post batches in L1 after compressing them, using a reversible compression algorithm [1]. However, the L1 gas associated with posting compressed batches is still significant. To mitigate this cost, Optimiums post hashes instead, and add a *Data Availability Committee* (DAC) storing batches and providing them upon request (see Fig. 2). In Optimiums, the sequencer posts a hash of a batch—which is much smaller than the compressed batch—along with evidence, i.e. signatures, that the batch is available from at least one honest DAC member. STFs then check that posted hashes have enough valid signatures and request the corresponding batch to DAC members. To ensure progress, some implementations provide a fallback mechanism, where the sequencer posts the compressed batch into L1 if it cannot collect enough signatures within a specified time frame. In this work, following [10], we use *arranger* to refer to the service that combines a sequencer and a DAC.

2.3 Model of Computation

Our system comprises arranger replicas and arranger clients. There are two types of clients: (1) L2 users sending L2 transaction requests to arranger replicas, and (2) STFs requesting arranger replicas the translation of hashes (posted through the L1 **logger** smart contract) into batches.

We consider a public-key infrastructure that associates replica and client identities with their public keys, and that is common to all replicas and clients. L2 users can create *valid* transaction requests and arranger replicas cannot impersonate clients. Valid transaction requests are those that have been correctly signed, so arranger replicas can locally validate them using public-key cryptography.

Arranger replicas can be classified as either *honest*, meaning they adhere to the arranger protocol, or *faulty*. Faulty replicas include *Byzantine* replicas, which behave arbitrarily [28]. We analyze other kinds of faulty replicas in Section 7.

Arranger replicas use a known collision-resistant hash function $\mathcal{H}$ to hash transaction requests and create Merkle trees from batches of transactions requests.

2.4 Merkle Trees

Merkle trees [32] are a tree data structure, where each leaf node is labeled with the hash of its content and each node that is not a leaf is labeled with the hash of the concatenation of its children hashes.

[2] In Arbitrum One STFs are allow-listed [18].

Merkle trees provide an efficient (logarithmic) membership authenticated check, which we employ in our fraud-proof mechanisms (see Section 5). Given the hash of the root r of a Merkle tree, also known as *Merkle root*, the proof that the content of the i-th leaf is x consists of index i, element x, and the hashes of all nodes that are neighbors of nodes in the path from the i-th leaf to the root. Verifying the proof involves reconstructing the hash in all nodes in the path from the leaf to the root, which can be done bottom-up with the data provided in the proof. Assuming that the hash function used is collision resistant, the last hash will match the Merkle root r if and only if x is in fact the content of the i-th leaf.

Given a batch b we consider the Merkle tree that has as leaves the elements in b and uses $\mathcal{H}$ as hash function, and denote with $\mathsf{Mroot}(b)$ its Merkle root.

3 Arranger

In this section we briefly explain the concept of arranger, introduced in [10], which seamlessly fits into the model of existing Optimiums (see Fig. 2). Arrangers perform two main functions: (1) *Sequencing*: ordering and batching transaction requests, and then posting the corresponding hashes to L1; and (2) *Data Availability*: ensuring the availability of data corresponding to the posted hashes, enabling the reconstruction of batches when needed. Arranger replicas offer an end point `translate`(id, h) to translate hashes.

The arranger service receives transaction requests from L2 users. When the arranger collects sufficient transaction requests (or a timeout occurs) all honest arranger replicas agree on a new batch b and assign an identifier id to b. Then, all honest arranger replicas compute hash h, the root of the Merkle Tree [32] whose leaves are the transaction requests in b, and create a *batch tag* (id, h). Once b has been agreed, (id, h) is computed locally. Each honest replica then signs the new batch tag and a compressed version of the batch and propagates its signatures to all other replicas. Once enough signatures of batch tag are collected, a combined signature σ is generated. The resulting *signed batch tag* (id, h, σ) is then posted to L1. The second signature, of the compressed version of the batch, is used in a fraud-proof game that guarantee that the batch is available (see Section 5.1). An attempt to post a batch to L1 becomes an L1 transaction request, which can be reordered. Therefore, we use identifiers in signed batches to order batches. In particular, two consecutive batches agreed by honest arranger replicas have consecutive identifiers.

The **logger** L1 contract accepts signed batch tags without performing any validation check. In this paper, we propose fraud-proof mechanisms—similar to arbitration protocols in Optimistic Rollups—to discard incorrect or unavailable signed batch tags (see Section 5).

STFs monitor the **logger** contract and, after locally validating the signatures of signed batch tags, request the corresponding batch from the arranger to compute the next L2 block. If the signature is invalid, the batch cannot be retrieved or the batch contains invalid transaction requests, the STFs can use a fraud-proof mechanisms to discard the batch tag and penalize misbehaving arranger replicas.

3.1 Arranger Properties

Before given the properties of *correct* arrangers, we first introduce some definitions about signed batch tags. Signed batch tags are considered *certified* when they have at least S signatures of arranger replicas, where S is a static system parameter known by the arranger replicas, the **logger** contract and the set of L1 smart contracts arbitrating fraud-proof mechanisms. A certified batch tag (id, h, σ) is *legal* if its corresponding batch b satisfies the following properties:

- ***Validity***: Every transaction request in b is a valid transaction request added by an L2 user.[3]
- ***Integrity$_1$***: No transaction request appears twice in b.
- ***Integrity$_2$***: No transaction request in b appears in a batch corresponding with a legal batch tag previously posted by the arranger.

All certified batch tags posted by correct arrangers must be legal.

Property 1 (***Legality***). *Every certified batch tag posted by the arranger is a legal batch tag.*

Arranger replicas can post multiple signed batch tags with the same identifier and the **logger** accepts them all. A batch tag can be part of two signed batch tags if each tag is signed by a different subset of arranger replicas. However, to ensure deterministic evolution of L2, two certified batch tags with the same identifier must correspond to the same batch, and thus have the same hash.

Property 2 (***Unique Batch***). *Let (id, h_1, σ_1) and (id, h_2, σ_2) be two posted certified batch tags. Then, $h_1 = h_2$.*

To ensure censorship resistance and data availability, correct arrangers must also satisfy the following properties:

Property 3 (***Termination***). *All valid transaction requests added to honest replicas eventually appear in a posted legal batch tag.*

Property 4 (***Availability***). *Every posted legal batch tag can be translated into its batch.*

Availability is expressed formally as follows. Let (id, h, σ) be a legal batch tag posted by the arranger, s.t. $h = \mathsf{Mroot}(b)$. Then, some honest replica will return b when requested `translate`(id, h). This prevents halting the L2 blockchain and having indisputable L2 blocks by failing to provide batches of transaction requests from hashes. If a faulty replica returns a batch b' with $b' \neq b$, by the assumption of collision resistance, the root of the Merkle tree corresponding with b' cannot be h. Clients can locally verify this mismatch by computing $h = \mathsf{Mroot}(b')$.

Legality, ***Unique Batch*** and ***Availability*** are safety properties and ***Termination*** is a liveness property. Altogether, they characterize correct arrangers. Correct arrangers offer censorship resistance because all valid transaction requests added to honest arranger replicas are eventually executed, guarantee that no transaction is executed twice and that only valid transaction requests are executed. For more details, see [10].

3.2 Arranger Implementations

A straightforward implementation of an arranger, which we refer to as *centralized*, consists of a single replica that performs all actions of arrangers: receiving transaction requests from L2 users, packing them into batches, assigning unique identifiers, posting hashed batches in L1 and translating hashes back into batches of transaction

[3] A transaction request is valid when it is properly formed and signed by the originating L2 user.

requests. Clearly, the correctness of a centralized arranger depends entirely on its single replica.

Most existing L2s do not implement a centralized arranger but instead what we call a *semi-decentralized* arranger.[4] In this architecture, a single replica acts as the sequencer and the remaining replicas implement a decentralized DAC. The centralized sequencer collects and batches transaction requests, assigns unique identifiers to create batch tags, communicates these batches and batch tags to all DAC replicas, collects their signatures and posts signed batch tags to the **logger** in L1. DAC replicas then can provide the inverse resolution of hashes posted by the sequencer. To ensure correctness, implementations of a semi-decentralized arranger assume that the sequencer is honest and the number of faulty replicas in the DAC is less than the signature threshold S. The concrete value of S and its relation with the number of total replicas n vary depending on the system configuration.

To completely remove the single point of trust and failure that is the centralized sequencer in semi-decentralized arrangers, [10] proposes a *fully decentralized* arranger based on Set Byzantine Consensus (SBC) [16]. SBC is a variant of Byzantine Consensus that provides high throughput where replicas agree on a set of elements, instead of a single element. In this implementation all replicas perform both the roles of sequencer and DAC member, and each honest replica includes an honest SBC replica as a building block. L2 users can add transaction requests through any replica. Set consensus guarantees that all honest replicas eventually agree on the same batch and that elements added to honest replicas eventually appear in a set agreed by honest replicas. Then each replica can locally compute the hash of the batch agreed by consensus, use the SBC instance as the batch identifier, and thus create a unique batch tag, sign the batch tag, and later translate hashes back into batches. SBC assumes that less than one-third of all replicas are Byzantine. This assumption is also inherited by the fully decentralized arranger in [10], which also requires that the number of signatures required in certified batch tags is at least $S > n/3$, guaranteeing that all certified batch tags are signed by an at least one honest replica and ensuring the correctness of the arranger.

4 Solution Overview

Arranger implementations assume that a portion of their replicas are honest in order to ensure correctness. However, they neither offer a mechanism to detect when this trust assumption is violated, nor provide guarantees in scenarios where the trust assumption does not hold.

This can have undesired consequences on the evolution of the L2. For example, violations to property ***Availability*** can lead to all L2 tokens to be stolen. Consider a fully-decentralized arranger that is correct under the assumption that at most $f < S$ replicas are faulty, and an adversary that breaks this trust assumption by controlling S arranger replicas. In such case, the replicas controlled by the adversary can generate certified batch tags that are known only by these replicas that the adversary controls. They can post a batch tag in the **logger** contract and then refuse to translate the hash into a batch of transaction requests, violating property ***Availability***. As consequence, the adversary could control an STF that posts a new L2 block encoding a state where all L2 tokens are stolen. Since other STFs do not know the corresponding transaction requests, they are unable to compute the correct L2 block and properly challenge the malicious block. To address this problem, in this paper we propose fraud-proof mechanisms (see Section 5) that can be used to reject batch tags that are illegal or unavailable. These fraud-proofs can also be used to generate evidence that an arranger does not satisfy the safety properties, discouraging faulty behavior by replicas. Furthermore, we provide economic incentives (see Section 6) to motivate arranger replicas to remain active.

Specifically, arrangers replicas post batch tags in L1 as *proposals*. Similar to L2 blocks assertions, batch tag proposals are optimistically accepted, but there can be challenges during a challenging period. During this period, arranger replicas and STFs can stake on a batch tag claiming (1) that the batch tag is legal and unique, and (2) that they can translate the hash. Replicas can also put stakes to challenge other batch tags proposals. There are different kinds of challenges, depending on the claim disputed. For each challenge, there is a fraud-proof game, arbitrated by an L1 contract, played between the replicas involved. Staking agents that fail to defend their claim lose their stake, either because their claim was false or because they did not participate correctly in the fraud-proof game. A batch tag is discarded (along with all L2 blocks executing its transactions) when it has no stake. Conversely, a batch tag consolidates when at least one of its staker survives all challenges.

The different fraud-proof mechanisms (FP) for batch tags are:

1. Certifiability FP mechanism: disputes the legality of the batch tag, claiming that the signature in the batch tag does not contain at least S valid arranger replica signatures.
2. Validity FP mechanism: disputes the legality of the batch tag, claiming that it contains an element that is not a valid transaction request.
3. Integrity FP mechanism 1: disputes the legality of the batch tag, claiming that it contains a duplicate element.
4. Integrity FP mechanism 2: disputes the legality of the batch tag, claiming that it contains an element that appears in a previous consolidated batch tag.
5. Uniqueness FP mechanism: disputes the uniqueness of the batch tag, claiming that there is another certified batch tag with the same identifier but different hash.
6. Data availability FP mechanism: forces the batch to be revealed in L1.

All these FPs and the strategies for honest players, are explained in detail in Section 5. The first five FPs guarantee that an staker making the correcty claim can win the challenge. The data availability FP ensures that either the transaction requests corresponding to the challenged batch tag are revealed or the batch tag is discarded (see Proposition 1). From this, we derive the following key lemma:

Lemma 1. *A single honest agent can ensure that a batch tag consolidates if and only if it is legal, available and, at the end of its challenge period, there is no other certified batch tag posted in L1 with the same identifier but different hash.*

Observe that if an arranger violates one of its safety properties, it means that the arranger either (1) posted a batch tag that is either not legal or unavailable, or (2) posted two certified batch tags with the same identifier but different hash. By the previous lemma, a

[4] For the current decentralization status of L2s running on top of Ethereum see [11]

single honest agent can use fraud-proof mechanisms to discard such batch tag.[5] Moreover, by winning the challenge the honest agent proves that the arranger is violating a safety property and, as consequence, force the replacement of the arranger.

Theorem 1. *A single honest agent can prove in L1 that an arranger violates a safety property.*

Unfortunately, detecting that the arranger is violating property ***Termination*** is impossible as this is a liveness property that may be satisfied later in the future. To mitigate the risk of an arranger censoring transactions requests, L2 users can post their transaction requests directly in L1, although at a higher cost.

Finally, the data availability challenge requires posting the batch compressed, which can be expensive. As an alternative, we provide in Section 6.2 a cheaper protocol to translate hashes into batches using zero-knowledge contingent payments. Although this cheaper protocol is preferable for both arranger replicas and STFs, it does not provide any guarantee when arranger replicas remain silent and thus the data availability challenge is a necessary fallback.

5 Fraud-proofs Mechanisms

We present now fraud-proof mechanisms (also called fraud-proof games, or for conciseness FPs), governed by L1 smart contracts, which enable a single honest agent (replicas and STFs)—with enough tokens to participate—to prevent the consolidation of illegal or unavailable batches, and to expose faulty behavior. Our FPs do not depend on the concrete arranger implementation, the number of faulty replicas or the correctness of the arranger. In Section 7, we study how these mechanisms can limit the power of different adversaries. We assume participating agents have public L1 accounts.

Our FP mechanisms are similar to the ones used in optimistic rollups to dispute L2 blocks, which are based on RDoC [8, 9]. The main difference is that our FPs are over concrete algorithms that check the availability and legality of posted batch tags, and not over the execution of arbitrary smart contracts translated to WASM [43]. Arbitrating over concrete algorithms allows us to divide the algorithms in well-defined high level blocks and arbitrate over these blocks instead of sequences of instructions that are not known upfront. This results into modular and clear FP mechanisms that are amenable for formalization, which are also more efficient. For example, consider an algorithm that verifies a Merkle proof of a Merkle tree with 4096 (= 2^{12}) transaction requests. The execution trace of a WASM program obtained by compiling a Rust implementation contains around 2^{21} instruction. Therefore, a bisection game over its execution trace involves 21 moves per player, where each move requires accessing to intermediate states of the execution trace. On the other hand, the multi-step membership FP that we propose (see Section 5.2) requires 4 moves per player (that only require knowing nodes in the Merkle tree), because this FP mechanism is logarithmic in the height of the Merkle tree. Furthermore, a one-step membership FP (see Section 5.2) requires only 1 step, provided that the gas limit can check the Merkle proof in a single execution.

Our FPs are two-player games arbitrated by L1 contracts. Each player takes turns with a predetermined total time per game, which only decreases in the player's turn (similar to chess clocks, the time mechanism currently implemented in the FPs employed in L2s).[6] Players must put a stake to participate. The losing party loses its stake, while the winner can retrieve its stake and receives a reward.

We describe our FP mechanisms and provide an overview of the strategy for honest players. Formal proofs of the FP of batch tag legality are formalized and proven correct in the companion artifact [12]. The pseudocode for contracts arbitrating the FPs and the strategy for honest players; figures illustrating the states of each FP, the allowed moves in each state, and the data required for each move; and informal proofs can be found in [11].

5.1 Data Availability Fraud-Proof Mechanism

Section 6.2 includes a fast and cheap protocol, based on contingent payments, to obtain the batch corresponding to the the hash of a batch tag. However, arranger replicas can be slow or refuse to participate in this protocol without any punishment. The *data availability FP mechanism* provides an alternative to force agents with a stake in a batch tag t to publish the associated batch b.

Once an agent A initiates a data availability FP mechanism against batch tag t, any agent staking in t can respond by posting to a L1 contract (called *data availability contract*) a compressed version of b along with S signatures from arranger replicas. The data availability contract rejects the data if its signature is not valid or does not contain the same signatures as the batch tag, but does not check that the posted data is the compressed version of b. This last step is delegated to another FP, called *decompress-and-hash FP mechanism*, that consists in bisecting the execution trace of a program P that decompresses the data posted, obtaining a list of elements, creates a Merkle tree with the list of elements as leaves, and checks that the Merkle root is the hash in the previously posted batch tag t. Agent A can initiate the decompress-and-hash FP claiming that program P does not finish successfully with the data provided by the staking agents. If A wins the decompress-and-hash challenge, all staking agents lose their stake and the batch is discarded. Otherwise, A loses its stake. Decompress-and-hash is similar to the arbitration game used in optimistic rollups for L2 blocks, but instead of arbitrating over the execution of an arbitrary smart contract, the decompress-and-hash FP uses a fixed algorithm, whose execution trace is shorter.[7] Since this FP mechanism extends RDoC, we can conclude the following result.

Corollary 1. *Agent* A *has a winning strategy for the decompress-and-hash FP mechanism if and only if the data provided by staking agents does not correspond to a compressed batch.*

Honest Strategies. After a batch tag t is proposed in L1, and before it consolidates, any agent can initiate a data availability FP game. If the staking agents do not respond or only respond with data that does not correspond with the compressed version of b, A can use the compress-and-hash FP mechanism to discard batch tag t. Otherwise, at least one staking agent posts the compressed version

[5] In the second case, if neither batch tag has consolidated yet, then both are discarded. Otherwise, the last one posted is discarded, as it is the one being challenged.

[6] The general consensus among the Ethereum research community and L2 projects about the duration of a game is two weeks (one week per player).

[7] The execution trace of a WASM program obtained by compiling a Rust program that decompress a string representing a compressed batch of 4000 transaction requests contains less than 2^{27} instructions, significantly less than usually expected in L2 blocks in practice.

of b correctly signed, and A (and anyone observing the L1) can decompress the data offchain and obtain b. Hence, either A learns the transaction requests in batch tag t or t is discarded.

Proposition 1. *For each not yet consolidated signed batch tag posted in L1, any honest agent can force to either learn its transaction requests or get the batch tag removed.*

Conversely, honest arranger replicas only stake in batch tag for which they know its batch of transaction requests and have collected at least S signatures of the compressed version of the batch. Therefore, if an honest arranger replica R stake in batch tag t, it can always win any data availability FP mechanism against t. In particular, as response to the initiation of the data availability FP mechanism against t, R posts in L1 the compressed version of the batch associated with t. The challenging agent can either not respond, in which case R (and all agents staking on t) win the FP, or proceed to the compress-and-hash FP mechanism which, by corollary 1, R can win as the data posted is correct.

Even when the data is revealed, some staking agents still lose some tokens due to the cost of publishing data in L1. To encourage replicas to participate and avoid silent replicas (also known as "free-rider" replicas), all replicas that stake in a batch tag also put a "communal stake". For each challenge, part of this communal stake is removed. Paying ahead of time for the cost of posting the challenge response in L1, ensures that the replicas responding to the challenge do not incur in a higher cost than silent replicas.

5.2 Membership Fraud-proof mechanism

Here we explain the *membership FP mechanism*, which is used in some of the FPs designed to prevent the consolidation of illegal batch tags and prove violations of property ***Legality***.

In the membership FP mechanism an agent A claims that element e is the i-th leaf in a Merkle tree mt whose root is h—where h is a hash known in L1—while another agent B denies the claim. We assume that both A and B know all nodes in the Merkle tree mt. We describe here two versions of this FP mechanism, one that consists of only one (expensive) step, and another that consists in multiple simple steps, with a total number of steps that is logarithmic on the height the of Merkle tree mt. The multi-step membership version can be used when the one-step version requires too much gas due to the size of the batch. That is, the multi-step membership mechanism allows to split the one-step version in multiple smaller steps, where each step involves a cheaper interaction with the L1. For simplicity, we only explain the multiple-step version as a bisection. This can be generalized to a k-dissection as long as each step does not exceed the maximum gas limit of a transaction.

In the one-step version, A posts in a L1 contract, called *one-step membership contract*, an element e, a position i, and a sequence of hashes π. The one-step membership contract checks that π is the membership-proof that encodes that element e is the i-th leaf in Merkle tree mt. The check consists in computing an amount of hashes equal to the height of mt. If the proof is valid, A wins the game, otherwise B wins the game. It is easy to see that A has a winning strategy if and only if e is the i-th leaf in Merkle tree mt.

In the multiple step version, A posts in an L1 contract called *multi-step membership contract* an element e, its hash h_e, position i, and a hash h_m, claiming not only that e is the element in the i-th leaf in mt but also that h_m is the hash in the node in the middle of the path π from the i-th leaf to the root in Merkle tree mt. The multi-step membership contract first checks that e hashes to h_e, and is also used to arbitrate the bisection game over π. Agents alternate performing the following operations: B selects one sub-path to challenge, and if the sub-path is longer than two, A responds by providing the middle hash of the sub-path chosen by B. When a sub-path has only two nodes, where the parent has hash h_p and the the child has h_c, A must post a hash h_s. The multi-step membership contract then verifies that hashing the corresponding (left or right) concatenation of h_s with h_c results in h_p. If the hashes are correct, A wins the game, otherwise B wins the game. The multi-step membership contract verifies hashes only during A's first and last turn and thus, each invocation to the contract is cheap. Since the path length is halved at each turn, the maximum number of turns is logarithmic in the path length, which is already logarithmic in the number of elements in the Merkle tree mt.

Honest strategies for the multi-step membership FP mechanism. If A's claim is correct, then A can win the multi-step membership FP, as it knows all nodes in mt. Therefore, A can provide all the hashes requested when bisecting the path and the hash in the final step.

Conversely, if A's claims is incorrect, B can always keep the invariant that the top node in the challenged path is part of the path from the i-th leaf to the root of mt, but not the bottom node. This can be achieved by choosing the top path when the hash proposed by A in its previous turn does not match the hash of the corresponding node in mt, and the bottom path otherwise. Eventually, when the challenged path has only two elements, h_p and h_c, A cannot provide a hash h_s such that hashing the corresponding (left or right) concatenation of h_s with h_c results in h_p. If A were able to provide such a hash this would mean that h_c is the child of h_p but since h_p is in the path h_c would also be in the path.

Proposition 2. *Let mt be a Merkle tree with root h known in L1, and let agents* A *and* B *know all elements in mt. Assume* A *has initiated a membership FP game claiming that e is the i-th leaf in mt. If e is the i-th leaf in mt, then* A *can win the game. Otherwise,* B *can win.*

5.3 Fraud-proofs to Prevent Illegal Batch Tags

We now present FP mechanisms to prevent the consolidation of illegal batch tags. We assume that agents know the original batch (see Section 5.1 or Section 6.2). If a batch tag $t = (id, h, \sigma)$ is illegal, depending on which condition is violated agent A can play one of the following FPs.

Certifiability FP mechanism, violation of condition ***Certified*** (σ does not contain S valid signatures). A function in L1 smart contract that arbitrates certifiability checks whether σ has enough signatures, and another function in the same contract computes the aggregated public key of the claimed signers and checks that the signature σ is correct.[8] If A invokes one of these functions and σ is found to be incorrect, then the batch tag t is discarded.

The replica that posted t, as well as all agents who explicitly staked on t, lose their stakes. However, individual signers do not lose their stakes, as it is the poster responsibility to ensure that

[8] Alternatively, a FP mechanism, arbitrated by an L1 contract, can arbitrate that the algorithm that checks multi-signatures accepts the signature provided.

the batch tag has enough valid signatures and are all correct. As described, this is a one-step FP game.

Validity FP mechanism, violation of condition ***Validity***. Agent A claims that there is an invalid transaction request e in the batch b of tag t. This FP mechanism is similar to the membership FP explained before except that in the first step the L1 contract verifies that e is an invalid transaction request.

Integrity FP mechanism 1, violation of condition ***Integrity***$_1$. Agent A claims that a transaction request e appears twice in t. Stakers can challenge any of the occurrences of e, by selecting a path to challenge and initiating a membership FP. A wins the Integrity FP mechanism 1 only if A wins a membership FP mechanism against all of the stakers.

Integrity FP mechanism 2, violation of condition ***Integrity***$_2$. Agent A claims that a transaction request e in t appears in a previous legal batch tag t'. As in the previous case, stakers can challenge any of the occurrences of e, initiating a membership FP, and A is required to win all of the FPs in order to win the Integrity FP mechanism 2.

Honest Strategies. An agent A that knows the transaction requests in an illegal batch tag t and all previous batch tags, can play the FP corresponding to the condition violated, win it and force the batch tag to be discarded.

Proposition 3. *Let t be an illegal batch tag not yet consolidated in L1, and let* A *be an agent that knows the batch of t and all previous legal batch tags.* A *can prevent the consolidation of t.*

Conversely, an honest agent staking in a legal batch tag t can win all Certifiability, Validity, Integrity 1 and Integrity 2 FP mechanism initiated against t.

5.4 Unique Batch Fraud-Proof Mechanism

The FP mechanisms described so far can be used to enforce that all batches that consolidate are legal and available, and to prove in L1 when the arranger violates ***Availability*** and ***Legality***. However, these mechanisms are not enough to prove in L1 violations of the ***Unique Batch*** property, which would allow arrangers to propose batch tags that fork the evolution of the L2.

To prevent this kind of attacks from malicious arrangers, we present a simple FP mechanism, called *unique batch* FP mechanism, which consists of only one step: an agent A invokes the unique batch L1 contract submitting two batch tags: (id_1, h_1, σ_1) and (id_2, h_2, σ_2), such that at least one of them has not consolidated yet. The unique batch contract verifies that:

- both batch tags have the same identifier, $id_1 = id_2$,
- each batch tag is certified, both σ_1 and σ_2 have at least S arranger replicas signatures each, and
- the hashes do not match, $h_1 \neq h_2$.

In this case all staking agents lose their stake, agent A is rewarded, and it is established that all arranger replicas whose signature appear in σ_1 or σ_2 should be removed from the arranger, as property ***Unique Batch*** is violated. A protocol to replace replicas is outside the scope of this paper. However, it is not possible to identify the specific replicas that are faulty (the signers of σ_1, the signers of σ_2 or both). This limitation follows from the inability to distinguish between the batches agreed by consensus and batches created by faulty replicas.

Honest Strategies. It is easy to see that if an arranger violates property ***Unique Batch***, any honest agent can use the unique batch FP mechanism to prove the violation in L1 and claim a reward.

Proposition 4. *Violations to* ***Unique Batch*** *can be proven in L1.*

Conversely, if an arranger does not violate property ***Unique Batch***, then all invocations to unique batch contract fail.

5.5 Mechanization using Lean4

In general, in order to prove that a single honest player can prevent the consolidation of illegal and unavailable batch tags and can prove in L1 violations to safety properties, one has to prove all FP mechanisms presented in the previous sections. However, in LEAN4, we focus on proving that only legal batches consolidate, as the correctness of the data-availability FP follows from the correctness of RDoC, and the unique batch FP mechanism is a one-step FP where the unique batch L1 contract perform a simple verification to decide the winner. We focus on the correctness arguments, so we do not model time in our proofs, assuming that players can interact freely with the L1. Similarly to the FPs for STFs in optimistic rollups, we assume that agents have enough time to perform actions. We define claims about batch tags legality as disputable assertions (DAs) composed of a tree where elements are at the leaves, and such that the Merkle root is the hash of the tag.

From Section 5.3, we have four different FP mechanisms, one for each way of making a batch illegal. Certifiability FP mechanism does not require any LEAN check as is a one-step FP which is directly checked by a L1 smart contract. FP mechanisms Validity and Integrity 1 are local to the batch tag, while Integrity 2 depends on the past history of the L2 batch tags. We define two notions of valid DA, one is local and depends only on the elements in a single batch, while the other depends on the entire history.

Definition 2 (Local Valid DA). We say that a batch of transaction requests is local valid if and only if all its transaction requests are valid and there are no duplicated elements.

Definition 3 (History Valid DA). We say that a batch of transactions requests is history valid if and only if it is local valid and no transaction appears in the history.

In LEAN, we write the above definitions as, assuming that $\mathsf{validTr}$ is a predicate indicating when a transaction request is valid:

$$
\begin{aligned}
\mathsf{localValid}(\mathsf{tree}, h) &\overset{\text{def}}{=} \begin{array}{ll} & \mathsf{merkleRoot}(\mathsf{tree}) = h \\ \wedge & \forall e \in \mathsf{tree}, \mathsf{validTr}(e) \\ \wedge & \mathsf{NoDup}(\mathsf{tree}) \end{array} \\
\mathsf{globalValid}(\mathsf{tree}, h, \mathsf{history}) &\overset{\text{def}}{=} \begin{array}{ll} & \mathsf{localValid}(\mathsf{tree}, h) \\ \wedge & \forall e \in \mathsf{tree}, e \notin \mathsf{history} \end{array}
\end{aligned}
$$

The predicate NoDup simply flattens the tree and checks that there are no duplicated elements in the resulting list. Additionally to the previous valid conditions, we have that the proposed hash is in fact the Merkle root of the proposed tree.

Without getting into implementation details, we define two simple protocols, called $\mathsf{LinearL2Protocol}$ and $\mathsf{LinearHistoryL2Protocol}$, where one player proposes a DA, a tree of elements plus its (supposed) Merkle root, and the other player decides what to do next.

The main difference between the protocols is that the protocol LinearHistoryL2Protocol takes into the account the history of the L2 blockchain. The options of the second player are to accept the DA or challenge one of the claims above. Challenging a claim means that one FP mechanism is triggered. We capture the honesty of the second player and prove that only valid DAs are accepted and all faulty ones are challenged and discarded.

The honest second player follows a simple algorithm: first, compute the Merkle root and if it differs from the hash proposed, trigger the decompress-and-hash FP mechanism.[9] If the proposed Merkle root is correct, the honest player checks if all elements are valid. In LEAN4, we find the first invalid element in the tree (if there is one), in which case the honest second player triggers a membership FP mechanism. If all elements are valid, the honest player checks if there are duplicated elements, again obtaining the first repeated element and triggering the Integrity 1 FP mechanism, which consists of *two* membership FPs in sequence. In the case of the honest player for the history protocol, we also check that all elements are not present in the current history.

We have two main FPs: one for data availability and one for membership. In LEAN, since we care about how computations are performed, we defined several similar FP mechanisms, in particular for membership. The multi-step membership FP can be played following the path in the Merkle tree (which is *linear*), one level at a time, either from the root to the leave or from the leave to the root. This game can be played by bisecting the path in a *logarithmic* game, as presented in Section 5.2. All these variant are implemented in the library and we are working on proving them equivalent for honest players.

Honest players in our LEAN implementation are defined as players that know all information, in particular they know all transaction requests in a batch. Using this information, they can compute all correct answers to all possible challenges when they follow the algorithm described above to trigger a FP game and then play the game following the corresponding winning strategy. In the library, we proved that honest players always win the data-availability and membership FP mechanisms. When honest players challenge faulty proposers, they win, leading to the proposed DA being rejected. When honest players propose a new DA, they can defend their claim and win against all possible challengers.

Theorem 4 (Only Valid DAs). *Let* Honest *and* HistHonest *be honest players following the previous algorithms, triggering FPs when needed. Then,* LinearL2Protocol *and* LinearHistoryL2Protocol *with one player being* Honest *only accepts local valid DAs* localValid *and history valid DAs* globalValid *respectively.*

In LEAN the previous result is expressed as follows, where (tree, h) is a DA proposed by an arbitrary player:

$$\mathsf{LinearL2Protocol}\,(\mathsf{tree}, h)\ \mathsf{Honest} \iff \mathsf{LocalValid}(\mathsf{tree}, h)$$

and given history H:

$$\mathsf{LinearHistoryL2Protocol}\,(H, \mathsf{tree}, h)\ \mathsf{HistHonest} \iff \mathsf{globalValid}(\mathsf{tree}, h, H)$$

The proof relies on verifying each function precisely. For example, when finding a duplicated element, the proof reduces to building the proper evidence to guarantee winning the corresponding FP mechanism. In total, the library has approximately 5256 lines of commented LEAN code. The main results can be found in the file "L2.lean" of the companion artifact [12].

6 Incentives

In this section, we present a collection of incentives that provides payments to replicas based on evidence of work, in particular for generating and signing hashes, posting batches into L1, and translating hashes into batches. The FP mechanisms from the previous section are deterrents against incorrect behavior that result in losing the stake. These are last-resort mechanisms to catch faulty behavior, like illegal and unavailable batch tags posted in L1, showing evidence of replicas faulty behavior.

We present now a framework where *rational* behavior (maximizing profit) aligns with honest behavior, i.e. rational behavior corresponds to the the arranger protocol. We also present a simple cost and reward analysis to determine the minimum budget required to guarantee that no illegal or unavailable batch tag consolidates, and to prove that an arranger violates a safety property. We give here an initial understanding of incentives, and introduce the necessary parameters for instantiating a formal game-theoretic model such as [38].

6.1 Incentives to Generate and Post Batches

Each consolidated batch generates the following rewards:

- Every replica receives a constant amount k_1 of L2 tokens.
- Replicas signing consolidated batches receive k_2 additional L2 tokens where k_2 is a monotonically increasing function on the number of signatures and transaction requests.
- The replica posting the batch in L1 receives an additional payment of k_3 L2 tokens to cover the posting costs.

These rewards create incentives to (1) participate in the arranger protocol; (2) sign batches, and communicate and collect signatures; (3) include as many transaction requests as possible in a batch; and (4) post batch tags in L1.

These incentives combined with our FPs make deviating from the arranger protocol irrational when there are no external payments. As result, our system is *incentive compatible* [29]. A more detailed quantitative study including expected utility and a formal game-theoretic security analysis (e.g., using frameworks like [3]), is left as future work.

All payments are charged to L2 users submitting transaction requests and made effective by STFs when computing the effects of batches that consolidated, including the payments as part of new L2 blocks. Optionally, L2 tokens can be minted by the L1 contracts that govern the evolution of the L2.

6.2 Incentive to Translate Hashes into Batches

We present now an alternative cheaper protocol for replicas to translate hashes and get paid using zero-knowledge contingent payments [7, 20, 23, 35]. Unlike the data availability FP mechanism from Section 5.1, this protocol operates primarily off-chain although it does not provide evidence of fraud if replicas remain silent.

[9] The decompress-and-hash FP takes a compressed batch and the (claimed) Merkle root, but in our LEAN formalization, we abstract this away for simplicity. The decompression step can be modeled as a partial function triggering an external one-step check.

Consider a client C, such as an STF, who wants to know the batch of transaction requests associated with a batch tag with hash h in order to compute the next L2 block, or to check the validity of an L2 block posted. When C requests the translation of h it can contact directly replica R, which knows the inverse translation b, but in this protocol R does not respond immediately with b. Instead, R creates a fresh secret key k, and computes $y = \mathsf{H}(k)$, where H is an algebraic instance of the hash function H, such as $\mathsf{H}(k) = g^k$, where g is a generator of a cyclic group. Subsequently, R encrypts b under public key y using an algebraic encryption scheme (e.g., ElGamal [21]). The observation in [23] is that it is possible to use an efficient Sigma protocol to create a zero-knowledge proof π proving that w is a valid encryption of b under y and that $y = \mathsf{H}(k)$, without revealing the secrets b or k. Finally, R sends π, w and y to C. Client C verifies the proof and, if it is correct, C generates a contingent payment where R is the only beneficiary. Replica R can collect the payment only by revealing the secret k. When R reveals k to collect the payment C learns k so it can decipher w and obtain b. All communication between R and C is offchain and one-to-one. The only interaction with L1 occurs when C creates the contingent payment and when R post the key k to collect the payment. Moreover, batch b is never posted in L1. Therefore, latency and bandwidth between replicas and clients are likely to not represent a major bottleneck.

A faulty replica R can participate in the first part of the protocol and then refuse to reveal k causing client C to incur in unnecessary costs. To prevent this, R must also sign y and send the signature to C, so C has evidence. If R fails to disclose k after some time has passed, C can use the signature as evidence to accuse R of remaining silent, have R stake removed and receive some compensation.

In contrast to the data availability FP mechanism, this protocol does not offer any guarantee to clients when replicas refuse to participate. However, this protocol is cheaper for clients and more rewarding for replicas than the data availability FP. Arranger replicas get a payment when translating batches using the offchain protocol but they do not receive any reward when the data availability FP is used. On the other hand, the offchain protocol is cheaper for clients than the data availability FP. Moreover, if replicas reveal the data when forced to participate in a data availability FP they lose not only part of the communal stake but also the reward that they would have obtained with this contingent payment. Hence, in practice, the data availability FP is a deterrent for replicas withholding data, and thus, this offchain protocol is always used to translate hashes (see Section 6.3), as both clients and replicas benefit from it.

6.3 Cost Analysis

We perform a simple cost and reward analysis for each FP mechanism presented in Section 5 and the translation protocol (see Section 6.2) to determine the minimum budget required by honest agents to guarantee that no illegal or unavailable batch consolidates, and to prove violations of safety properties.

First, we define the following variables, where x can be *data*, *certifiability*, *validity*, *integrity1*, *integrity2* or *uniqueness*, and y can be a move in a FP mechanism:

- s represents the tokens needed to stake in a batch tag.
- s_{com_data} are the tokens provided in a communal stake by each staker.
- s_x represents the stake needed by a client to start FP x.
- $cc_{translate}$ is the client's costs for using the offchain translation protocol. This includes the gas used to create the contract and the payment done to the staker.
- $sc_{translate}$ and $sr_{translate}$ are stakers cost (resp. reward) for translating a hash using the offchain protocol.
- cc_x and sc_x are client (resp. stakers) cost for participating in the FP mechanism of type x.
- cr_x are client reward for winning the FP of type x.
- C_y represents the cost of performing move y. The cost is covered by the player performing the move, which depending on move y can be the staker or the client.

All of these variables are in L1 tokens.

The cost of performing a move represents the gas required, which depends on the underlying L1 blockchain. The cost for participating in an FP mechanism is upper bounded by the gas needed in the worst case, i.e., the sum of the cost of all moves in the worst case.

Stakers do not receive any reward for winning an FP game, as their motivation for participating is to not lose their stake and eventually collect their reward from consolidated batches. However, they do receive a reward for correctly translating hashes using the offchain protocol as there is no stake at risk there.

The following relations must hold between the parameters:

1. The client costs to participate in FP mechanism x is significantly less than the reward for winning it, $cc_x \ll cr_x$.
2. Clients cover the costs for stakers in all FPs except in the data availability FP mechanism where this cost is covered by the communal stake. In symbols, $\sum s_{com\_data} > sc_{data}$ and for all remaining FP mechanism x, $s_x > sc_x$.
3. The stakes taken from the losing player cover the rewards. For the validity and integrity FP mechanisms, the client reward for winning the FP is less than the challenged agent stake, $cr_x < s$. For the data, certifiability and unique batch FP mechanisms, the client reward is bounded by the sum of all stakes in the batch, i.e. $cr_x < \sum s$, when x is data, certifiability or uniquiness. For the offchain translation protocol, clients cover the stakers reward, $cc_{translate} > sr_{translate}$.
4. The replicas must have a motivation to engage in the offchain translation protocol and gain a profit. That is, the staker costs to translate hashes in the offchain protocol is significantly less than its reward, $sc_{translate} \ll sr_{translate}$.
5. Clients must have a motivation to use offchain translation. That is, the client cost for using the offchain translation protocol is significant less than participating in the data availability FP mechanism, $cc_{translate} \ll cc_{data} + s_{data}$.
6. During the data-availability FP the client must initiate the game and bisect the execution trace of the WASM compilation of program P until the traces is reduced to a single instruction. Therefore, the client cost for participating in the data-availability FP is $cc_{data} = C_{init\_data} + l \times C_{bisect\_subtrace}$, where l is the length of the challenged trace.
7. The client cost for participating in the certifiability FP is the maximum cost between checking the number of signatures and checking the aggregated signature, that is, $cc_{certifiability} = \max(C_{check\_size}, C_{check\_agg})$.

(8) Unique-batch FP is a one-step game, then $cc_{uniqueness} = C_{unique\_batch}$

(9) During validity, integrity 1 and integrity 2 FPs a client must initiate the game, bisect the path until it consists of only two nodes, and reveal the sibling of the lower node. In symbols, the client cost for participating in FP x is $cc_x = C_{init\_x} + (\log\log(SZ) - 1) \times C_{bisect\_subpath} + C_{reveal\_sibling}$, when x is validity, integrity1 or integrity 2.

(10) In the data-availability FP the staker must provide the compressed batch and then select which subtrace to challenge until the subtrace correspond to a single instruction. Therefore, stakers cost for participating in the data-availability FP is $sc_{data} = C_{post\_compressed} + (l - 1) \times C_{select\_subtrace}$.

(11) No staker plays the certifiability and unique-batch FPs, therefore $sc_{certifiability} = sc_{uniqueness} = 0$.

(12) In the validity FP the staker must select which subpath to challenge until the subpath has length 1. Then, stakers cost for participating in the validity FP is $sc_{validity} = \log\log(SZ) \times C_{select\_subpath}$.

(13) In the integrity 1 and 2 FPs the staker must first select which path to challenge and then select which subpaths to challenge until the subpath has length 1. Therefore, stakers cost for participating in the integrity 1 and 2 FPs is $sc_{integrity1} = sc_{integrity2} = C_{select\_path} + \log\log(SZ) \times C_{select\_subpath}$.

We now determine the minimum budget that an honest agent A needs to guarantee that illegal or unavailable batch tags are discarded. This amount is computed from the maximum requirement for playing the corresponding mechanism. In the case of ***Validity***, ***Integrity***$_1$, and ***Integrity***$_2$, A may need to get the data first, and thus, A needs to account tokens for playing the data availability FP game. Agent A has to play these mechanisms against all stakers. The budget proposed is enough if A plays all of them in sequence, accumulating rewards along the way. See [11] for a complete proof.

Proposition 5. *An agent* A *that knows all transaction requests in previous legal batch tags can discard unconfirmed illegal or unavailable batch tags, provided that* A *has at least the following budget* $B = max(s_{certifiability} + cc_{certifiability}, s_{data} + cc_{data} + max(s_{validity} + cc_{validity}, s_{integrity1} + cc_{integrity1}, s_{integrity2} + cc_{integrity2}))$.

Since the required budget to discard illegal or unavailable batch tags, B, depends on clients stake and the size of the batch, any L2 that decides to implement our proposal can set those values to obtain an upper-bound for B. By limiting the number of transaction requests per batch and the stake clients must put to play each FP, any implementation of our proposal can guarantee that an adversary cannot construct an illegal batch tag with arbitrary cost in such a way that no honest agent A has sufficient tokens to successfully challenge its illegality. More generally, our work provides a framework with its proof of correctness, but leaves variables like SZ and s_x open. This allows for concrete implementations to set these parameters according to their specific use cases and requirements.

By discarding an illegal or unavailable batch tag that is certified, an agent also proves that the arranger violates properties ***Legality*** and ***Availability***. Violations of ***Unique Batch*** must be proven by playing the unique batch FP. Therefore, proving that an arranger violates any safety property requires a budget enough to play the unique batch FP and to discard illegal and unavailable batches.

Corollary 2. *An agent* A *that knows all transaction requests in previous legal batch tags can prove that the arranger violated a safety property, provided that* A *has a bugdet of at least* $max(B, s_{uniqueness} + cc_{uniqueness})$ *tokens.*

Finally, the offchain protocol is cheaper for clients and replicas, and more rewarding for replicas than the data availability FP. Therefore, rational agents are inclined to use the offchain protocol.

Furthermore, by adjusting the allocation of the total cost of the data availability FP between clients and stakers, the protocol can influence the equilibrium price of the contingent payment.

6.4 Analysis with Concrete Values

Here we provide representative values for the variables introduced earlier, assuming that the L1 is Ethereum and that the batch size is bounded by $SZ \leq 4096$ transaction requests. For simplicity, we assume that each move consumes at most 100.000 gas, approximately the median gas used in Ethereum transactions, and a gas price of 3 Gwei.[10] Except for transaction posting data compressed, which prior blobs cost 0.06 ETH. [11]

Under these assumptions, the cost of posting compressed data is $C_{post\_compressed} = 0.06$ ETH and the cost of all other moves y is $C_y = 0.0003$ ETH. Using relations 6-9, we obtain that the client cost for each FP is: $cc_{data} = 0.6081$ ETH, $cc_{certifiability} = cc_{uniqueness} = 0.0003$ ETH, $cc_{validity} = cc_{integrity1} = cc_{integrity2} = 0.0039$ ETH. These values give a lower bound for clients rewards (see relation 1), which in turn provide a lower bound for staker stakes in batch tags (see relation 3). Assuming clients rewards for each FP x is 1 ETH plus the client cost for FP x, we obtain: $cr_{data} = 1.0084$ ETH, $cr_{certifiability} = cr_{uniqueness} = 1.0003$ ETH, $cr_{validity} = cr_{integrity1} = cr_{integrity2} = 1.0039$ ETH. Given these bounds, a stake of 10 ETH is more than enough to be used as an stake for arranger replicas.

Similarly, using relations 10-13, the staker cost for each FP is: $sc_{data} = 0.0084$ ETH, $sc_{certifiability} = sc_{uniqueness} = 0$ ETH, $sc_{validity} = 0.0036$ ETH, and $sc_{integrity1} = sc_{integrity2} = 0.0039$ ETH. By relation 2, these values give lower bounds for both the communal stake and clients stake in each FP. Hence, the communal stake can be set to $s_{com\_stake} = 0.009$ ETH, while client stake $s_x = 10$ ETH for all FPs.

Therefore, in this scenario, by Proposition 5, the minimum budget needed to discard illegal or unavailable batches is $B = 20.6141$ Regarding the offchain translation protocol, replicas cost is the cost of a single transaction, $sc_{translate} = 0.0003$, while clients cost is the cost of single transaction plus the replica reward, $cc_{translate} = 0.0003 + sr_{translate}$. To satisfy relations 4 and 5, replicas reward must satisfy: $0.0003 \ll sr_{translate} \ll cc_{data} + s_{data} - cc_{translate} = 10$ ETH. Thus, a reasonable reward in this setting is 1 ETH.

Finally, when adding transactions requests L2 users must pay a fee enough to cover replicas rewards upon batch consolidation (see Section 6.1). Suppose a batch tag b with S' signatures consolidates. The total reward distributed to replicas is given by: $k_1 \times n + k_2 \times S' + k_3$. Since the concrete value of S' and the batch size are unknown at the time of transaction submission, we propose that each L2 user pays a flat fee of $(k_1 \times n + k_2 \times S + k_3)/SZ$. If the total collected fees are

[10] Data obtained from https://studio.glassnode.com/charts/fees.GasUsedMean?a=ETH and https://studio.glassnode.com/charts/fees.GasPriceMean?a=ETH, respectively, on July 2025.

[11] See, for example, transaction 0x4276c81d7b51...X0693e0a6bbade5, where the Arbitrum Sequencer post a batch in Ethereum.

insufficient—e.g., due to more signatures than S—L2 tokens can be minted to make up the difference, introducing inflation. Consider an scenario of a fully decentralized arranger consisting of $n = 31$ replicas where the cost of executing a transaction in L1 corresponds to 10 L2 tokens. By setting parameters $k_3 = 10$, $k_2 = 200$, and $k_1 = 1$, the resulting fee for L2 users would be approximately 0.5. This is roughly $\frac{1}{20}$-th the cost of executing a transaction on L1, providing a significant cost advantage for L2 users.

7 Threat Models

We consider three types of adversarial models, all computationally bounded, as in [5, 6]. We analyze how two of these adversary can impact the evolution of the L2 under the different implementations of the arranger described in Section 3.2 (centralized, semi-decentralized, and fully-decentralized). The third adversarial model is specific to the fully-decentralized implementation, as it reasons about the batch tags consented by honest replicas.

BFT Adversary. In this adversarial model, all replicas are either honest or Byzantine, and we assume that the trust assumptions of the arranger implementation are not violated.

In particular, for the centralized implementation this means that the only replica is honest. In the semi-decentralized implementation the sequencer and at least $n - S + 1$ DAC members are honest. Finally, for the fully-decentralized implementation the adversary can control less than one third of the replicas.

Therefore, under this adversary, the arranger implementation is correct (see [10] for a proof for the semi-decentralized and fully decentralized implementations). However, Byzantine replicas can still post batch tags that are not certified, so the certifiability FP should be utilized to discard such batch tags.

Arranger Adversary. This is the strongest adversary, capable of controlling all arranger replicas. This adversary can decide the content of all legal batch tags created and posted by the arranger. In particular, the adversary has the power to exclude valid transaction requests from posted legal batch tags, potentially violating property ***Termination***. Unfortunately, detecting that the arranger is violating ***Termination*** is impossible as this is a liveness property that might be satisfied later in the future. Then, this adversary can censor transactions without any consequence. To mitigate this risk, L2 users can post their transaction requests directly in L1 although at a higher cost.

This adversary can also violate all safety properties, but this offers no advantage to the adversary, as this violations can be detected using our FPs (see Theorem 1), and merely exposes the corruption of the arranger. Furthermore, even under this extremely powerful adversary, Lemma 1 holds and a single honest agent can guarantee the legality and data availability of consolidated batch tags.

DAC Adversary. The power of this adversary lies between the previous two and only applies to the fully-decentralized arranger. The previous two adversaries capture what happens with arranger implementations either when no trust assumption is violated or when the adversary has full control over the arranger. Here, we consider a more practical adversary where the trust assumptions of the arranger are violated but honest replicas still have the power to create certified batch tags.

This adversarial model refines the Byzantine failure model [28] by introducing a new type of faulty replicas: *corrupt* replicas. Corrupt replicas behave arbitrarily without violating any property of SBC, that is, they only misbehave in the DAC part of the arranger. For example, corrupt replicas may sign incorrect batches or refuse to translate batches, but corrupt replicas can neither prevent honest replicas for agreeing on the content of batches nor prevent honest replicas from including elements added by clients to honest replicas. The portion of Byzantine replicas is less than one-third, the total number of honest replicas is at least $S = \frac{n}{3}$, where S is the minimum number of signatures required for a batch to be considered certified, and the remaining replicas are corrupt. As there are at least S honest replicas, honest replicas can create certified batches without requiring signatures from faulty replicas.

Unlike in the previous adversarial model, property ***Termination*** still holds under this adversary. Since the SBC works properly, honest replicas can agree on the content of each batch, and all valid transaction requests added to honest replicas eventually appear in an agreed batch. As there are enough honest replicas to certify batch tags, each batch agreed by them will be posted as a legal batch tag. Consequently, all valid transaction requests added to honest replicas eventually appear in a legal batch tag posted in L1.

As with the Arranger Adversary, if this adversary violates any safety property, it can be detected using our FP mechanisms. However, detecting violations of property ***Unique Batch*** requires the action of honest replicas. Since this adversary controls more replicas than the certification threshold S, faulty replicas can create certified batch tags without the signature of any honest replicas. Therefore, faulty replicas can post a certified batch tags whose batch is not the one agreed by consensus. In this case, honest replicas can post a legal batch tag with the same identifier but different batch content, violating property ***Unique Batch*** and exposing that the arranger has been corrupted. This violation can be detected simply by observing L1 and the uniqueness FP mechanism can then be used to trigger the replacement of arranger replicas. As a consequence, if the arranger under this adversary does not behave as under adversary BFT then its corruption can be exposed.

8 Related Work

To the best of our knowledge, our work is the first to provide economic incentives for arrangers replicas to behave honestly, and fraud-proof mechanisms arbitrated by L1 contracts to demonstrate safety violations by arrangers, and thus, generate evidence of fraud from faulty replicas.

Current L2s do not provide any mechanism to guarantee that batches posted by their arranger are legal or available, because their arranger is *assumed* to be correct. Arrangers in Optimistic Rollups and ZK-Rollups can post data that does not correspond to a compressed batch of valid transaction requests. Similarly, arrangers of Optimiums and Validiums can post a hash whose inverse does not represent a batch of new valid transactions. The implications of these actions are significant, e.g., malicious STFs could post illegal new L2 blocks that cannot be disputed by other STFs as they do not know the transaction requests to play the fraud-proof over the L2 block. Hence, malicious arrangers in current L2s can block the L2 or, even worse, create indisputable incorrect L2 blocks. By contrast, our

FPs prevent such issues under all adversarial scenarios outlined in Section 7, including an adversary that controls all arranger replicas.

One of the FP we provide is to translate hashes, which has also been studied in [41]. However, they assume that the translation can be directly verified in L1 using SNARK proofs, while we provide an interactive FP that consists on posting the (claimed) compressed batch and a game over the execution trace of a program that checks whether the posted data corresponds with a given hash. In other DAC protocols, such as Celestia [27], clients can accuse replicas outside of L1 of withholding data, and thus suffer from the following problem. They cannot distinguish between the following two situations: (1) malicious clients falsely accuse replicas of withholding data; and (2) malicious replicas reveal data only after being accused. In protocols like [27], this problem has security implications depending on the reward received by the accusing client:

(a) If the net reward is positive even when the data is revealed, rational clients can take advantage by falsely accusing replicas just to collect the reward.
(b) If the net reward is zero (independently of whether the data is revealed or not), denial-of-service attacks are possible because malicious clients can issue several accusations without penalty.
(c) If the net reward is negative, only altruistic clients would accuse malicious stakers.

In our protocol, there is no detection and accusation of withholding data. Instead, the data availability FP is used by clients to ensure access to the data (or remove stakes). Furthermore, clients incur on a cost when playing the data availability FP, ensuring that it is only used when necessary, so points (a) and (b) do not apply. Regarding the third point, in our protocol clients always pay to access the data, either through the data availability FP or by the cheaper offchain protocol. However, if the data is malicious or unavailable, clients receive significant compensation, and if the data is revealed and it is correct, clients can generate L2 blocks executing the transaction requests and be rewarded for it. Replicas, on the other hand, lose their stake if the data is malicious and earn nothing if the data is correct and revealed only in a data availability FP.

Other works that deal with the detection of malicious behavior include accountability in consensus [13], and MPC with identifiable abort [24]. The main difference between our work and MPC with identifiable abort is that we not only identify faulty replicas but also generate evidence of their faulty behavior that anyone can verify, like in accountability in consensus. However, an advantage of our work over accountability in consensus is that anyone observing the posts done by arranger replicas can generate evidence of faulty behavior (when safety properties are violated), rather than being restricted to the replicas running the protocol.

9 Conclusions and Future Work

Layer 2 blockchains aim to improve the scalability of current smart contract blockchains, offering a much higher throughput without modifying the programming logic and interaction of blockchains. An important component of L2s is the arranger, which batches transaction requests, commits these batches hashed, and translates hashes into batches. Current implementations of arrangers in most L2s assume an upper bound in the number of faulty replicas in order to guarantee correctness. However, these implementations lack mechanisms to detect when this trust assumption is violated, and they provide no guarantees in cases where the assumption does not hold. As consequence, arrangers have the power to influence the evolution of the L2.

To address this, we described a collection of incentives for replicas to behave honestly and penalties for posting incorrect or unavailable batches based on fraud-proof mechanisms. These fraud-proof mechanisms are over properties of concrete algorithms rather than the execution trace of arbitrary algorithms, as is the case in RDoC and in the fraud-proofs for used for the correctness of L2 blocks in Optimistic Rollups and Optimiums. Using these mechanisms a single honest agent can guarantee that only valid and available batches of transaction requests are executed in L2, regardless of the number of faulty replicas in the arranger and the correctness of the arranger, limiting the power of adversaries. Furthermore, these fraud-proof games can also be used to prove in L1 violations of safety properties from the arranger, therefore also acting as deterrent for faulty behavior. Our incentives and fraud-proof mechanisms are independent from the arranger implementation.

Artifacts. [12] This paper is accompanied by a library of mechanized formal proofs of all our fraud-proof mechanisms from Section 5.3. developed in the proof assistant Lean4 [33]. This includes a proof that there is a winning strategy for the honest player.

9.1 Future Work

One promising direction for future work is to extend our fraud-proof mechanisms to other properties, such as order fairness [26].

Another interesting line of research is enhancing the L1 to support more efficient fraud-proof mechanisms. Most L1 contracts arbitrating fraud-proof mechanisms first perform some kind of validation (e.g. verify signatures or compute hashes of input values and compare them with stored hashes), and then update their state accordingly. Interestingly, the validation step executed by these L1 contracts is typically a *pure computation*: its result depends only on the inputs and not on the blockchain state. As consequence, these pure computations could be executed in parallel with all other transactions and other pure computations as they do not interfere with their outcome. To leverage this, we propose extending the semantics of L1 contracts in blockchains like Ethereum to support function parameters of a special type, representing the result of pure computations. Functions with these kind of parameters should be only invoked by external users initiating the transaction, to ensure that the input to the pure computation do not depend on the blockchain state. As these pure computations are part of transactions, their results is consented just like regular transaction. Unlike regular transactions, which must be executed sequentially, pure computations could be executed concurrently with other pure computations and regular transactions, improving the throughput. Therefore, the gas consumed by pure computations could be lower than the gas for transactions, similarly as the cost of posting data in Ethereum is cheaper when using blobs [4]. This approach would not only reduce the cost of existing fraud-proof games but also allow to simplify more complex ones. For example, consider the decompress-and-hash FP, that consists in bisecting the execution trace of a program that decompresses the data posted, obtains a list of elements, creates a Merkle tree with the list of elements as leaves, and finish successfully if the Merkle root is a given hash. This FP game could be implement as a one-step FP mechanism that consists of executing the whole program as a pure computation, because its computation only depends on the data provided.

Finally, to completely characterize the system in the Lean4 library we need to add the notion of time, possibly as bounded interaction trees [19] modeling the effects of agents interacting.

Acknowledgments

Funded by project PID2022-140560OB-I00 (DRONAC) funded by MICIU/AEI /10.13039/501100011033 and ERDF, EU; grant CEX2024-001471-M/ funded by MICIU/AEI/10.13039/501100011033, grant PID2022-138072OB-I00 and grant PID2022-142290OB-I00, funded by MCIN/AEI/10.13039/501100011033/ FEDER, UE.

References

[1] Jyrki Alakuijala, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka, and Lode Vandevenne. 2018. Brotli: A General-Purpose Data Compressor. *ACM Transactions on Information Systems* 37, 1 (2018), 1–30.

[2] L. Bousfield, R. Bousfield, C. Buckland, B. Burgess, J. Colvin, E. Felten, S. Goldfeder, D. Goldman, B. Huddleston, H. Kalonder, F. Lacs, H. Ng, A. Sanghi, T. Wilson, V. Yermakova, and T. Zidenberg. 2022. Arbitrum Nitro: A Second-Generation Optimistic Rollup. https://github.com/OffchainLabs/nitro/blob/master/docs/Nitro-whitepaper.pdf.

[3] Lea Salome Brugger, Laura Kovács, Anja Petkovic Komel, Sophie Rain, and Michael Rawson. 2023. CheckMate: Automated Game-Theoretic Security Reasoning. In *Proc. of the 2023 ACM SIGSAC Conf. on Computer and Communications Security* (Copenhagen, Denmark) *(CCS'23)*. ACM, New York, NY, USA, 1407—-1421. https://doi.org/10.1145/3576915.3623183

[4] Vitalik Buterin, Dankrad Feist abd Diederik Loerakker, George Kadianakis, Matt Garnett, Mofi Taiwo, and Ansgar Dietrichs. [n. d.]. EIP-4844: Shard Blob Transactions. https://eips.ethereum.org/EIPS/eip-4844, Accessed: 2025-04-11.

[5] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology — CRYPTO 2001*. Springer, 524–541.

[6] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2000. Random oracles in constantipole: practical asynchronous Byzantine agreement using cryptography (extended abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '00)*. ACM, New York, NY, USA, 123–132. https://doi.org/10.1145/343477.343531

[7] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. 2017. Zero-Knowledge Contingent Payments Revisited: Attacks and Payments for Services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 229–243. https://doi.org/10.1145/3133956.3134060

[8] Ran Canetti, Ben Riva, and Guy N Rothblum. 2011. Practical delegation of computation using multiple servers. In *Proceedings of the 18th ACM conference on Computer and communications security*. 445–454.

[9] Ran Canetti, Ben Riva, and Guy N. Rothblum. 2013. Refereed delegation of computation. *Information and Computation* 226 (2013), 16–36.

[10] Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Pedro Moreno-Sánchez, and César Sánchez. 2025. A Decentralized Sequencer and Data Availability Committee for Rollups Using Set Consensus. arXiv:2503.05451 [cs.DC] https://arxiv.org/abs/2503.05451

[11] Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Pedro Moreno-Sánchez, and César Sánchez. 2025. A Secure Sequencer and Data Availability Committee for Rollups (Extended Version). arXiv:2509.06614 https://arxiv.org/abs/2509.06614

[12] Martin Ceresa. 2025. *LeanFraudProofs: L2HistoryCCS*. https://zenodo.org/records/16993467

[13] Pierre Civit, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, and Jovan Komatovic. 2023. As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy! *J. Parallel and Distrib. Comput.* 181 (2023), 104743. https://doi.org/10.1016/j.jpdc.2023.104743

[14] Coinbase. [n. d.]. base. https://base.org/, Accessed: 2025-07-13.

[15] Ethereum Community. [n. d.]. Ethereum development documentation - Blocks. https://ethereum.org/en/developers/docs/blocks/#block-size, Accessed: 2025-07-09.

[16] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2021. Red Belly: A Secure, Fair and Scalable Open Blockchain. In *Proc. of S&P'21*. 466–483. https://doi.org/10.1109/SP40001.2021.00087

[17] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. 2016. On Scaling Decentralized Blockchains. In *Financial Crypto. and Data Security*. Springer, 106–125.

[18] Arbitrum Foundation. [n. d.]. The state of Arbitrum's progressive decentralization. https://docs.arbitrum.foundation/state-of-progressive-decentralization#allowlisted-validators, Accessed: 2024-11-25.

[19] Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL, Article 12 (Jan. 2024), 30 pages. https://doi.org/10.1145/3632854

[20] Georg Fuchsbauer. 2019. WI Is Not Enough: Zero-Knowledge Contingent (Service) Payments Revisited. Cryptology ePrint Archive, Paper 2019/964. https://doi.org/10.1145/3319535.3354234 https://eprint.iacr.org/2019/964.

[21] Taher El Gamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory* 31, 4 (1985), 469–472. https://doi.org/10.1109/TIT.1985.1057074

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

[23] Javier Gomez-Martinez, Dimitrios Vasilopoulos, Pedro Moreno-Sanchez, and Dario Fiore. 2025. Algebraic Zero Knowledge Contingent Payment. In *Applied Cryptography and Network Security - 23rd Int'l Conf., ACNS 2025, Proc., Part II (LNCS, Vol. 15826)*. Springer, 224–254. https://doi.org/10.1007/978-3-031-95764-2_10

[24] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. 2014. Secure Multi-Party Computation with Identifiable Abort. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8617)*. Springer, 369–386. https://doi.org/10.1007/978-3-662-44381-1_21

[25] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium*. USENIX Assoc., 1353–1370. https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

[26] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-Fairness for Byzantine Consensus. Cryptology ePrint Archive, Paper 2020/269. https://eprint.iacr.org/2020/269 https://eprint.iacr.org/2020/269.

[27] Celestia Labs. [n. d.]. Celestia. https://celestia.org, Accessed: 2024-05-21.

[28] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[29] John O. Ledyard. 1989. *Incentive Compatibility.* Palgrave Macmillan UK, London, 141–151. https://doi.org/10.1007/978-1-349-20215-7_15

[30] Lens Labs. [n. d.]. Lens. https://lens.xyz/, Accessed: 2025-07-13.

[31] Matter Labs. [n. d.]. zkSync. https://zksync.io/, Accessed: 2025-07-13.

[32] Ralph C. Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO '87*, Carl Pomerance (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–378.

[33] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[34] Satoshi Nakamoto. 2009. Bitcoin: a peer-to-peer electronic cash system.

[35] Ky Nguyen, Miguel Ambrona, and Masayuki Abe. 2020. WI is Almost Enough: Contingent Payment All Over Again. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, 641–656. https://doi.org/10.1145/3372297.3417888

[36] Optimism Foundation. [n. d.]. Optimism. https://www.optimism.io/, Accessed: 2025-07-13.

[37] L2BEAT research team. [n. d.]. L2BEAT. https://l2beat.com/scaling/summary Accessed: 2024-05-21.

[38] Giulia Scaffino, Lukas Aumayr, Mahsa Bastankhah, Zeta Avarikioti, and Matteo Maffei. 2025. Alba: The Dawn of Scalable Bridges for Blockchains. https://doi.org/10.14722/ndss.2025.241286

[39] Sophon. [n. d.]. Sophon. https://sophon.xyz/, Accessed: 2025-07-13.

[40] Starknet [n. d.]. Starknet. https://www.starknet.io/en, Accessed: 2025-07-13.

[41] Ertem Nusret Tas and Dan Boneh. 2024. Cryptoeconomic Security for Data Availability Committees. In *Financial Cryptography and Data Security*. Springer Nature Switzerland, Cham, 310–326.

[42] Shobha Tyagi and Madhumita Kathuria. 2021. *Study on Blockchain Scalability Solutions*. ACM, 394–401. https://doi.org/10.1145/3474124.3474184

[43] WebAssembly Working Group. [n. d.]. WebAssembly. https://webassembly.org/.

[44] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.