# Towards a Mechanization of Fraud Proof Games in Lean

**Martín Ceresa**
IMDEA Software Institute, Madrid, Spain

**César Sánchez**
IMDEA Software Institute, Madrid, Spain

## Abstract

Arbitration games from Referee Delegation of Computations are central to layer two optimistic rollup architectures (L2), one of the most prominent mechanisms for scaling blockchains. L2 blockchains operate on the principle that computations are valid unless proven otherwise. Challenging incorrect computations requires users to construct fraud proofs through a dispute resolution process that involves two opposing players. Fraud proofs are objects that establish when proposed computations are invalid, and they are so computationally small and cheap that can be checked by the underlying trusted blockchain. Arbitration games, this challenging process, involve one player posing strategic questions and another player revealing details about computations.

Arbitration games start from the posting of Disputable Assertions (DAs), DAs contain partial information about computations including their result. Since there is no trust between players, hashes are posted as compact witnesses of knowledge. One player provides information decomposing hashes while the other decides which "path" to take navigating the computation trace. When a path is exhausted, all the required information to compute the result from the data provided following the path has been revealed and the path can be proven to be faulty or correct.

We explore in this paper the formalization of arbitration games in Lean4, introducing the first machine-checkable strategies that honest players can play guaranteeing success. These strategies ensure: on one side, the successful debunking of dishonest computations via the construction of fraud proofs, while in the other, the successful navigation of the challenge process through correct answers. In short, these are the winning strategies that honest players (on both sides) can follow. We explore in this paper formal abstractions to capture disputable assertions, arbitration games on finite binary trees asserting data-availability and membership, game transformations, and then discuss how to work towards a general formal framework for referee delegation of computations.

**2012 ACM Subject Classification** Software and its engineering → Formal methods; Software and its engineering → Correctness; Software and its engineering → Software libraries and repositories; Theory of computation → Interactive proof systems; Theory of computation → Program reasoning; Theory of computation → Program constructs

**Keywords and phrases** blockchain, formal methods, layer-2, optimistic rollups, arbitration games

**Digital Object Identifier** 10.4230/OASIcs.FMBC.2025.5

**Supplementary Material** *Software*: `https://gitlab.software.imdea.org/martin.ceresa/leanfpgames` [13], archived at `swh:1:dir:761ba38f606c1b4a0a9e202e6518d092d51ff381`

**Funding** Partially funded by DECO Project (PID2022-138072OB-I00) – funded by MCIN/AEI/10.13039/501100011033 and by the ESF+ – and by grant from Nomadic Labs and the Tezos Foundation.

**Acknowledgements** Thanks to Margarita Capretto for her insightful ideas and help.





## 1 Introduction

Blockchain[1] technology is the first massively adopted decentralized third-party trusted mechanism to perform money exchanges [25]. The second wave of blockchains introduced computational mechanisms, so users were not only capable of performing money transactions but also they could perform computations [9]. Users now can upload small programs, called smart contracts (contracts), whose execution governs the interaction between users, including the transfer of tokens and cryptocurrency. Enabling agents to describe and invoke computations opened a whole new horizon in this field, for the good or bad.

The good part is the fulfilling of a long due dream of having a technology to run trusted third-party code [34, 37]. The contract code describes with precision what is going to be executed, and users can trust and reproduce what the resulting effects of executing transactions are [29]. This spawned a new challenge of formal program verification [33, 28, 2, 26, 6, 15, 5, 3, 31, 32, 17, 4, 23, 11].

The bad part is that this new technology came with a whole new attack surface on contracts. Programs now share the same "memory-space", all agents can invoke contracts and contracts can invoke any other contract, which can potentially produce unexpected transfer of cryptocurrency, or the break of interaction protocol designed between contracts by developers [14].

The massive adoption of contract based blockchains came with another cost. Resources are limited and blockchains are decentralized, so there is limit in the number of transactions per unit of time that can be included in the blockchain, making computation expensive. Expensive computations come in two flavours: expensive execution and space costs. To solve these two problems, several solutions were proposed: scaling the blockchain itself (with techniques like sharding [36] or faster consensus protocols [19, 27, 16]), or devising mechanisms to compute offchain (outside the blockchain), minimizing the blockchain interactions [19].

Rollups are offchain mechanisms to provide a solution for blockchain scalability. These offchain solutions are potentially dangerous since the only trusted computation is what executes in the blockchain using contracts (onchain). Therefore, to provide the same guarantees as the underlying blockchain, L2 introduces new concepts, and they come (broadly speaking) in two flavours.

**Zero-knowledge Rollups.** Zero-Knowledge (ZK) proofs are mechanisms to provide proof of correctness of computations that can be verified onchain by a smart contract [35]. Ideally, committing zk-proofs and checking their correctness is less expensive than actually running the computations they verify. In addition, the system must guarantee that no faulty proofs can be produced, and that all correct computations can be proven (even if crafting their proof is expensive). Therefore, instead of running contracts, zk-proofs are generated and provided to be checked onchain, and if the proof passes the verification process, the L2-blockchain evolves. This approach is implemented by several solutions[2] but a scalable solution based on ZK-proofs is still under research [22].

**Optimistic Rollups.** Computations are assumed correct unless proven otherwise [21, 10]. Instead of verifying zk-proofs, optimistic rollup schemes employ an arbitration mechanism that guarantees a single honest agent (with sufficient resources) is capable of detecting and

[1] We refer to distributed ledgers with crypto-currencies capabilities as *blockchain*.
[2] Readers can check the state of L2 ecosystems at `https://l2beat.com/scaling/summary`

reverting dishonest computations as well as defending correct computations. The correctness of the system is then predicated on a single agent observing the evolution of the system. The scalability predicates on the deterrent of being caught lying preventing dishonest players from posting dishonest computations. Instead of running checks or other mechanisms on transactions, optimistic rollups only keep track of the proposed transactions and block effects produced by executing them. There is a period window where these proposed effects can be challenged. When proposals are challenged, the party proposing the next state and the challenging party play an arbitration game to decide whether the proposal is (in)correct. The proposal is discarded if the challenging party wins the arbitration game or stays alive waiting for the period to end otherwise. Losing parties lose a stake that they must place and winning parties win a reward, which create incentives to participate honestly in the ecosystem and to deter dishonest behavior. Proposals surviving the challenging time period become permanent and the L2-blockchain evolves. That is, the correctness of an L2 optimistic rollup lays on the combination of (1) a correctness criteria that states the ability of a single honest to effectively dispute dishonest allegations (either remove faulty computations or defend honest computation claims), and (2) the economic incentives for agents to follow honest behavior. We focus in this paper only on the computational part of these schemes.

The correct evolution of L2 optimistic rollup blockchains rely on the actions of honest agents. Honest players can locally compute and propose the next blockchain state executing sequentially transaction requests. A single honest player is capable of challenging faulty proposals and winning arbitration games. In this article, we focus on the study of the correctness of arbitration games formally. It is crucial that an honest proposer can propose the next block knowing that it can always defend the proposals. Also, a single honest challenger must be able to challenge a dishonest proposal knowing they can always win, preventing the blockchain from progressing dishonestly.

Previous works capture computations as interactive games between players formally using *interaction trees* [20, 38]. Moreover, there is a big effort in characterizing general computations and proving properties about programs and protocols [7, 8]. We do not follow such a general approach here but a much more concrete one, because of the nature of our model of computation, where authenticated data structures are at the core of our approach. We took a more pragmatic path and trying to maximize the use of two simple building blocks: Merkle tree formation and membership games. An important characteristic of our approach is that our games are finite and players have bounded time to play (as in chess clocks).

The main artifact of our work is a Lean4 library that models the concepts in arbitration games, including DAs, players and strategies. We provide proofs showing the correctness of the strategies for honest players. Moreover, since Lean4 generates executable programs, we exercise our strategies and have them interactively play the arbitration game.

The readers can find the code of the library and all proofs in a git repository `https://gitlab.software.imdea.org/martin.ceresa/leanfpgames` where we created a tag pointing at the current state of the library "FMBC.Final". The repository is not yet complete but will be publicly released as a full library in the near future.

## 2 Preliminaries

In this section, we briefly explain the relevant notions to this article of Optimistic Rollups. Arbitrum Optimistic Rollups [21] are the first implementation of L2 optimistic rollups, implementing Referee Delegation of Computations [10], serving as the leading example in the L2 ecosystem attacking blockchain scalability issues. The main idea is to perform as

much computation as possible outside the blockchain while keeping the same guarantees. Optimistic Rollups propose to optimistically execute transactions, i.e. transaction executions are assumed correct unless someone challenges the correctness of the computation[3].

The evolution of the L2-blockchain goes as follows: one agent proposes a *disputable assertion* (DA) asserting a fact, e.g. what the result of executing the effect of a transaction is, and if the DA survives a fixed period of time, it becomes a committed fact. We can see how this is appealing for blockchains, no expensive computations are done if DAs survive. The brilliant idea comes when DAs are challenged.

When a DA is challenged, the agent proposing the DA, the *proposer*, and the agent challenging, the *challenger*, engage in a turn-based two player game. In this game, called arbitration game, the two players compete against each other to build a witness. Witnesses prove that one of the players is wrong and the other is right. When witnesses prove proposers wrong, they are called fraud-proof and are employed to debunk claims. When witnesses prove challengers wrong, they do not prove proposers right, and depending on the claim and arbitration game, they can provide partial correctness.

Participation in the L2 blockchain evolution is rewarded and misbehaviour penalized and, in arbitration games, it is strictly enforced through penalties. Players stating facts (DAs) place stakes on them, and in the case they cannot support their claims, they lose their stakes. Players challenging DAs place stakes on their (challenging) claim, which they can lose if they cannot debunk the supposedly false claim they are challenging. The entire game is played through the underlying blockchain and the losing party loses their stakes and the winning party is rewarded. We focus on the computation mechanics and not in the monetary analysis of arbitration games.

One may ask why would players lie if they are going to lose money. In this article, we provide a way to prove that lying players are going to get caught by playing the game[4]. We complete our argument with that rational agents will not play to lose money[5], and thus, arbitration games are never played[6]. In other words, arbitration games are deterrent mechanisms never to be played but necessary to guarantee the correct evolution of the blockchain.

### Optimistic Model of Computation for Blockchains

In L2 Optimistic Rollups, the goal is to compute the next state of the blockchain, i.e. the result of applying the effects of executing transactions in a given block. In Refereed Delegation of Computation (RDoC) [10], the authors decompose the execution of transactions as small steps in Turing machines, while in Arbitrum [21], they use small steps in the etherum virtual machine described using WASM. Both approaches map trust of the whole computation to a single computation step run in a trusted computation device, a blockchain.

RDoC and Arbitrum use hashes to represent compact witnesses. Assuming there is a collision resistant hash function $\mathcal{H}$, one can use Merkle Trees, a compact representation of trees into single hashes. Merkle trees are the main example of authenticated data structures, one can locally verify element $e$ belongs to a tree with a path of hashes leading to $e$ without knowing the entire data behind the Merkle tree.

---

[3] In this article, we do not address questions as Sibyl attacks preventing agents from challenging malicious transactions by blocking their access to the blockchain.

[4] Assuming players have enough money and can interact with the blockchain to play the game.

[5] Assuming a close economic world. In open economic worlds, agents may lose at some small close markets while winning somewhere else and actually have bigger net earnings.

[6] To the authors knowledge, there is no evidence of arbitration games ever played in the whole Arbitrum blockchain history nor in similar L2 blockchains like Optimism.

Transactions as checkable sequences of operations. Transactions in blockchain are stored as traces of executed operations, representing all the small steps required to go from the previous state to the current state of the blockchain. When these operations are expanded, we see their name and arguments. Therefore, we can hash each basic operation along with its arguments to create a compact witness, a hash which is the sole result from hashing such operation (because we are using a collision resistant hash function). Now, we can form a balanced tree with the sequence of hashes and form a Merkle tree with them. This hash is a witness of the whole computation generated by the original transaction.

The main idea of L2 optimistic rollups is to propose DAs asserting the result of the computation to be as compact as possible. Therefore, we have DAs as sequences of hashes plus their Merkle tree representing the skeleton of the computation tree. Other players can challenge DAs by requesting hashes until leaves are found. When a leaf is found, the proposing party needs to reveal the raw data used to compute leaf hashes, i.e. basic operations and their arguments. Therefore, the last basic step has to be run by the trusted computation device. One game is that whenever challengers request hashes, we split the computation in half bisecting the trace, and ask the DA proposer to provide the corresponding hash. This game is called bisection game and they are a subclass of a more general notion of game called arbitration games.

In theory, i.e. in the work of RDoC [10], computations are represented as the steps taken by Turing machines. Therefore, having a trusted single Turing machine step interpreter is enough to simulate one step and provide trusted execution to the whole ecosystem. In practice, i.e. in the work of Arbitrum [21], computations are represented as small steps taken by the etherum virtual machine (EVM). Therefore, in Arbitrum, they instrumented the EVM and adapted it to single step executions using WASM and the bisection game is performed over their machine steps, the Arbitrum Virtual Machine (AVM). As result of executing transactions, we have a list of low-level verifiable atomic operations taking the current state of the blockchain to the next one.

In this work, we abstract these small steps and only focus in the main operations over Merkle trees. This work is part of a bigger research enterprise where we believe there are other ways to trust from small to big computations, we show our first step as an example in Section 5 and a small discussion in Section 6.

Agents observing the blockchain know everything. All information is public, and thus, all agents know what transactions are being executed and can compute their resulting effects. In particular, agents can compute all intermediate steps and the resulting Merkle tree of all computations. Therefore, if an agent is lying other agents know about it and can engage in an arbitration game.

In the protocol just described, two things can go wrong:

- The Merkle tree hash is not the hash resulting from hashing the trace.
- There is something wrong with the data (elements) in the trace.

Therefore, there are two basic building blocks in this protocol.

When posting DAs, the proposing agents are committed to provide (if required) information derived from the witness. Witnesses are Merkle trees, so the hashes provided must hash the parent hash in each step. In Arbitrum, when agents propose the next step in the L2 blockchain placing a DA, they publish a compression of the trace plus the resulting hash. When there is something wrong with the trace (or the trace itself), opposing agents challenge such DA. In both challenges the idea is the same, having a top hash, the agent defending the DA (usually the one proposing it) decomposes the top hash into other hashes, and thus we have a way to link previous proof witnesses into the new ones. The challenger party decides which path to take in case hashes are wrong, repeating the process.

The difference between the two possible dishonest moves is whether or not the challenger party accepts the top hash to be correct. If the top hash is incorrect, the hash does not follow from the data proposed, a *data-availability* arbitration game is played [7](see Section 3.2). If the top hash follows from the data but the data is incorrect, an *membership* arbitration game is played (see Section 4). In the case where the membership arbitration game is played, we also need a trusted validation function so we can test its validity once an element is proved to be part of the data provided. Other properties can be defined using the membership arbitration game, for example, to show that an element appears twice in a block, it suffices to show that it appears at two different places.

# 3 A Generic Fraud Proof Game Formalization

In this section, we present an abstract formalization of fraud proof games that encompasses the arbitration games (and RDoC) presented in Section 2. Then, we instantiate this formalization to other games proposed in [12], which are games used over correct encodings of batches of transactions in Layer 2 blockchains. These games include, for example, specific games to decide whether an element belongs to a Merkle tree (see Section 4).

## 3.1 DAs

Protocols begin when a proposing agent asserts the result of a given computation. We abstract away some details and encode the representation of a computation as a tree, similarly to what algebras with a single binary operation [18] or suspended algebraic effects [30]. DAs can be interpretad as data and the resulting of consuming such data into a resulting value.

```
structure TraceTree (α β γ : Type) where
  mk :: (data : BinaryTree α β) (res : γ)
```

where `BinaryTree` are binary trees with leaves of type $\alpha$ and nodes having information of type $\beta$. We also implicitly (and optimistically) assume the following property for DAs:

```
  def implicit_assumption (comp : ComputationTree α β γ)(leaf_interpretation : α -> γ)
    (node_intrepretation : β -> γ -> γ -> γ) : Prop
    := fold leaf_interpretation node_interpretation comp.data = comp.res
```

The goal of L2 optimistic rollups is to avoid as much computation as possible, and thus, the `implicit_assumption` is never *executed* but it has to be guaranteed by the system. Then, faulty DAs are guaranteed to be discovered, and fault-proofs generated.

### Hashes – Authenticated Data Structures

In fraud-proofs verifiable blame can be assigned to players. The way this is done in RDoC is by creating computations using authenticated data structure (i.e. Merkle trees). When players post DAs, they are committing to a strategy, which states that following the computation the claimed result is obtained. The computation is encoded as a tree with no information, the skeleton of a computation, and the result is encoded as the hash that results from hashing the tree itself. The proposing player can be asked to expose the data in the computation and incrementally validate the data provided (by hashing it) against the submitted hash. In Lean, this means that our trace tree is of the form:

```
abbrev DAs (𝕌 : Type):= TraceTree Unit Unit 𝕌
```

[7] When the data is not public, this game can be employed as a very expensive data retrival mechanism.

Using the Lean class system, we can have a function hashing elements supporting a binary operation combining them:

```
class Hash (α ℍ : Type) where mhash : α -> ℍ

class HashMagma (ℍ : Type) where comb : ℍ -> ℍ -> ℍ
```

However, when a proposing players propose data behind a hash, we assume that they cannot provide a different element that collides with the original element, that is, that hashes have no collisions. Technically, we need to propagate the non-colliding condition to the binary operator as follows.

```
class CollResistant (α ℍ : Type)[op : Hash α ℍ] where
  noCollisions : forall (a b : α), a ≠ b -> op.mhash a ≠ op.mhash b

class SLawFulHash (ℍ : Type)[m : HashMagma ℍ] where
  neqLeft : forall (a1 a2 b1 b2 : ℍ), a1 ≠ a2 -> m.comb a1 b1 ≠ m.comb a2 b2
  neqRight : forall (a1 a2 b1 b2 : ℍ), b1 ≠ b2 -> m.comb a1 b1 ≠ m.comb a2 b2
```

This way of presenting the computational part through classes `Hash` and `HashMagma` and assumptions through classes `CollResistant` and `SLawFulHash` is very useful when having computations on one side and proof on the other. When defining games, functions and executing strategies, we work with their computational counterpart. When proving theorems about such functions and games, we need to also include our theoretical assumptions.

## 3.2 Generic Arbitration Games

Arbitration games are turn-based two-player games over DAs. One player reveals information, in this case hashes, while the other chooses which path to follow to continued the exploration of the trace tree proposed.

We define the following general game over binary trees abstracting types away:

```
inductive ChooserMoves where | Now | ContLeft | ContRight
def treeCompArbGame {α α' β γ : Type}
    -- Game Mechanics
    (leafCondition : α -> α' -> γ -> Winner)
    (midCondition  : β -> γ -> γ -> γ -> Winner)
    -- Public Information
    (da : ComputationTree α β γ)
    -- Players
    (revealer : BinaryTree (Option α') (Option (γ × γ)))
    (chooser : BinaryTree Unit ((γ × γ × γ) -> Option ChooserMoves))
    : Winner := match da.data, revealer with ...
```

Here, `Winner` is just a two element type to say which player has won, `TraceTree` is the DA defined before, and `ChooserMoves` describes choosers actions either challenge current assertion or chooses what path to take, left or right in binary trees. Players can choose not to play, modeled using Lean `Option` type.

The function `treeCompArbGame` pattern matches the arena in the DA, the player `revealer` and `chooser`, and feeds the chooser function with the information provided by the revealer. Depending on the result of the chooser, the game continues creating a new DA with the information provided by the revealer or the condition `midCondition` is triggered and one player wins[8]. Each pattern matching involving players represent a player interaction with the blockchain. A player that failing to fulfill their part loses the game.

[8] Full implementation in file GenericTree.lean: `https://gitlab.software.imdea.org/martin.ceresa/leanfpgames/-/blob/master/FraudProof/Games/GenericTree.lean`.

Because our arena is a tree and due the nature of Merkle trees, there are two ways to be fraudulent in this scheme. One is to provide faulty information, for example wrong small step in a transactions execution. The other is to give a faulty tree where some elements do not hash to their parents, that is, the computation itself is faulty. The first is what we call `leafCondition`, which is a condition on the leaves of the arena. The second is an intermediary condition, `midCondition`, that is, a condition over the nodes. When challengers engage in arbitration games, they try to find which nodes or leaves violate these conditions. Once we define our two conditions, we define a game.

**Valid Merkle Tree Game.** Instantiating the above game, we have the following game:

```
def cond_hash_elem {𝕄 α : Type}[BEq 𝕄][h : Hash α 𝕄]
  (leaf: 𝕄) (rev : α) (res : 𝕄)
  : Bool :=  h.mhash rev == res && leaf == res

def cond_hash { 𝕄 : Type }[BEq 𝕄][mag : HashMagma 𝕄] (res l r : 𝕄)
  : Bool := mag.comb l r == res

abbrev valid_Merkle_tree := treeCompArbGame cond_hash_elem cond_hash
```

Once the committing hash has been established as well-constructed, agents can play a different (more efficient) game challenging the validity of the claim by pinpointing an invalid element. Depending on the context, we have different validity test, e.g. no duplicated operations or small-step validity. The game then is reduced to show that there is an invalid element in the data proposed by proving that the invalid element belongs to the current DA Merkle tree. Since paths in Merkle trees can be seen as trace trees, we can play this game using `treeCompArbGame`. In Section 4, we define an alternative game for membership which is logarithmic in the length of the path.

## 3.3 Winning Strategies

We focus now on proving that honest players can always win. Depending on their role, players have different winning conditions. Players proposing DAs have the *optimistic advantage*: they are right unless proven otherwise. Honest players proposing DAs know the data they used to create, move first and, if required, defend their claim against all possible challengers, honest or otherwise. When it comes to challenging players, we can build winning strategies against dishonest proposers that submitted a faulty DA. Because of the optimistic advantage, honest challenging players – knowing the data behind the DA – only challenge when they detect there is an invalid DA.

From the definition of `treeCombArbGame`, we get that revealer players (in L2, the ones proposing DAs) can be challenged at any moment, and thus they need to win all possible challenges to make the DA consolidate. Therefore, revealer players, for a given DA, need to win all possible conditions (leaf and node conditions).

Challenger players follow the same reasoning, but in this case, they only challenge when they know they are going to win. In our games, challenger players act as choosers, choosing which path to take. This means they need to know the missing data in the computation tree before playing.

The following definition states that a player strategy follows a given DA and that leaf and node conditions are met:

```
def tree_comp_winning_conditions {α α' β γ : Type}
    -- Game Mechanics
    (leafCondition : α -> α' -> γ -> Prop)
```

```
    (midCondition  : β -> γ -> γ -> γ -> Prop)
    -- Public Information
    (da : ComputationTree α β γ)
    (player : BinaryTree (Option α') (Option (γ × γ)))
  : Prop :=
  match da.data , player with
  | .leaf a' , .leaf (.some a) => leafCondition a' a da.res
  | .node b' gl gr , .node (.some b) pl pr =>
    midCondition b' da.res b.1 b.2
    ∧ tree_comp_winning_conditions leafCondition midCondition ⟨ gl , b.1 ⟩ pl
    ∧ tree_comp_winning_conditions leafCondition midCondition ⟨ gr , b.2 ⟩ pr
  | _ , _ => False
```

For revealer players, if they know the data and computed the final result properly – that is, `tree_comp_winning_conditions` is true – they win against all chooser players. In L2 terms, revealers can defend their claim against all possible dishonest agents[9].

```
theorem winning_prop_hashes {𝕌 α : Type}
    [DecidableEq 𝕌]
    [Hash α 𝕌][HashMagma 𝕌]
    -- Public Information
    (da : ComputationTree 𝕌 Unit 𝕌)
    -- Players
    (revealer : BinaryTree (Option α) (Option (𝕌 × 𝕌)))
    (good_revealer : revealer_winning_condition
                        cond_hash_elem (fun _ => cond_hash) da revealer)
    : forall (chooser : BinaryTree Unit ((𝕌 × 𝕌 × 𝕌) -> Option ChooserMoves)),
      valid_merkle_tree da revealer chooser = Player.Proposer
    := winning_proposer_wins _ _ da revealer good_revealer
```

When it comes to challengers, first we need to generate the strategy and then prove a similar theorem but working over the assumption that the challenger knows the data and that the computation leads to a different hash.

```
theorem winning_gen_chooser {𝕌 α : Type}
    [hash : Hash α 𝕌][HashMagma 𝕌] [DecidableEq 𝕌]
    -- Public Information
    (pub_data : BinaryTree 𝕌 Unit)
    -- Players
    (revealer : BinaryTree (Option α) (Option (𝕌 × 𝕌)))(rev_res : 𝕌)
    (chooser : BinaryTree (Option α) (Option (𝕌 × 𝕌)))(ch_res : 𝕌)
    (good_chooser: winning_condition_player cond_hash_elem cond_hash
                     (const id) ⟨ pub_data , ch_res ⟩ chooser)
    (hneq : ¬ rev_res = ch_res)
    : valid_merkle_tree ⟨ pub_data, rev_res ⟩
          revealer (chooser.map (fun _ => ()) gen_chooser_opt)
    = Player.Chooser := by ...
```

The above proof needs to be sure that when the revealer provides the data (as a hash) the corresponding element is publicly known and no other element can be produced with the same hash. In this presentation, we are using `DecidableEq 𝕌` hiding this fact[10]. The same goes for intermediary steps, the chooser player needs to have some guarantee when choosing paths, because otherwise the revealer may produce elements hashing to the same hash, invalidating the challenge.

[9] Honest challengers will not challenge honest DAs.

[10] See file DataStructures/Hash.lean at `https://gitlab.software.imdea.org/martin.ceresa/leanfpgames/-/raw/master/FraudProof/DataStructures/Hash.lean`.

## 4 Membership Games for Merkle Trees

Specific fraud proof games have also been employed [12] to guarantee that Layer 2 sequencers propose valid batches of transactions. These fraud proof games do not verse about the outcome of generic computations (as in RDoC) but, they correspond instead to concrete programs that evaluate certain aspects of data-types, in particular of batches of transactions encoded as Merkle trees. The most fundamental building block for such games is a membership game that allows to prove that a given element is in the batch proposed. In turn, membership games can be used by a challenger to show that the batch is illegal because it contains repeated elements (providing proofs of the same element in two different positions). We detail membership games in the rest of this section.

The definition of a claim is similar to the DAs in the previous sections, but now the DA has the form of a Merkle tree path (instead of a trace tree), and additional indication of the source and destination hashes.

```
inductive Direction where | Left : Direction | Right : Direction
-- Sequence of length |n| indicating Left or Right
abbrev Path (n : Nat) := Sequence n Direction

structure ElemInMTree (ℍ : Type) (n : Nat) where
  path : Path n
  src : ℍ
  dst   : ℍ
```

The above claim encodes the idea that if hash `src` is a the root of Merkle tree, there is a path `path` of length `n` from `src` to `dst`. There are two (equivalent[11]) games we can play: a path from elem to `dst` and a path from `dst` to elem. For simplicity, we focus on paths starting from the element up to the root. The implicit property is the proof of an element belonging to a Merkle tree, which is the sequence of intermediate hashes lead to the root.

In this game, the arena is a list where one player reveals the missing data while the other either chooses to challenge the current step or continues up on the path. The missing data is (1) the next hash in the path from the current element and (2) the hash used to compute it– in the case of Merkle trees, this encodes a Merkle subtree. However, instead of defining a new game, we can use our previous definitions. We map the arena (a path) to a trace tree, and to map the players strategies, the usual way to map a sequence into a tree with one deep child and the other child being a leaf. We use the arena guiding their strategies indicating which child belongs to the path and which one is an unexplored subtree. We map the move `Continue` depending on the side `Direction` dictated by the path to `Left` or `Right`. Conditions check (if required) that hashes match, i.e. if the proposer player gave a subpath whose last element is the next hash in the path to the root.

```
inductive ChooserSmp : Type where | Now | Continue
def elem_in_tree_forward_gentree {ℍ : Type}
  [BEq ℍ][mag : HashMagma ℍ]{n : Nat}(da : ElemInMTree ℍ n)
  (proposer : Sequence n (Option (ℍ × ℍ)))
  (chooser : Sequence n (ℍ × ℍ × ℍ -> Option ChooserSmp))
  := treeCompArbGame leaf_condition_range mid_condition_range_one_step_forward
      {data := skl_to_tree da.data, res:= da.res}
      (build_proposer' da.res.1 da.data proposer)
      (build_chooser' da.data chooser)
```

[11] We prove them in Lean as there is a transformation to go and come back from both games resulting in the same player winning.

## Logarithmic FraudProof

We now introduce an alternative more efficient membership game, which requires a logarithmic number of moves on the length of the path. To prove both games equivalent, we define a transformation of the arena and prove that corresponding players that know the data win one game if and only if they win the other game. This game mimics the bisection game used in Arbitrum.

The linear and logarithmic games are different from the point-of-view of the challenger. When building the fraud-proof, we are verifying the existence of a path from a leaf to the root. In the linear game, we ask to the revealer to reveal each element along the path verifying that it is correct by checking that hashes match. In the logarithmic game, we ask for the hash of the element in the middle of the path, effectively bisecting the path in two, and then, choosing which half to challenge next. The main difference is that to choose whether to challenge the upper or lower sub-paths, the challenger needs to know the path upfront. On the other hand, in the linear games, the challenger does not need to know the path and can run the check and challenge until hashes do not match. Honest challengers playing the linear game only need to know that that the path is invalid. In fact, the path can be provided fully by the proposer and checked in one shot of computation (requiring to check a linear number of hashes in the size of path).

In the logarithmic games, the revealer – instead of decomposing the parent hash into two children hashes – given two hashes (corresponding to the extremes of the path), proposes the hash in the middle of the path.

First, we transform the arena. From a sequence of `Direction` of length `2^n` for some `n`, we build a tree having as leaves the sequence (in the same order) and no information at the nodes.

```
def built_up_arena {n : Nat} : Sequence (2^n) Direction -> BinaryTree Direction Unit
   := gen_info_perfect_tree (seq_constant ())
```

Then, we transform the strategy of the revealer in a similar way. From the missing data, we can compute all intermediary hashes along the path (spine hashes) and the auxiliary hashes (representing Merkle subtrees in the original computation tree). We take all spine hashes but the last (the Merkle tree root hash) and place them at the nodes and subtree hashes at the leaves.

```
def forward_proposer_to_tree {𝕄 : Type}{ n : Nat}
  (prop : Sequence (2^n) (𝕄 × 𝕄)) : BinaryTree 𝕄 𝕄
  := gen_info_perfect_tree
       ( Fin.init -- Drop last hash  (top hash [forward])
       $ sequence_coerce (by have pg := @pow_gt_zero n; omega)
       $ seqMap (fun p => p.fst) prop) -- Spine hashes
       ( seqMap (fun p => p.snd) prop) -- leaves matching subtrees
```

Finally, we show that the above transformations map linear revealer winning players into winning logarithmic revealer players.

```
theorem proposer_winning_mod_forward {𝕄 : Type} {lgn : Nat}
       [DecidabeEq 𝕄][HashMagma 𝕄] (da : ElemInTree (2^lgn) 𝕄)
       (proposer : Sequence (2^lgn) (𝕄 × 𝕄))
       (wProp : elem_in_revealer_winning_condition_forward
                          da (seqMap (.Next) proposer))
       (chooser : BinaryTree Unit (Range 𝕄 -> 𝕄 -> Option ChooserMoves))
       : spl_game ({data := built_up_arena da.data , res := da.mtree})
         ( BinaryTree.map .some .some $ forward_proposer_to_tree proposer)
         chooser = Player.Proposer := by ...
```

Where game `spl_game` is essentially the same as `treeCompArbGame` but instead of disclosing a pair of hashes from a hash, the revealer is ask to provide a hash in the middle of two hashes plus there is no node conditions (if triggered the proposer player wins the game.) In this game, both players have to play until a leaf is reached, since there is no way to know that intermediary steps are correct. Intuitively, we are not following small verifiable steps, but jumping around in the trace tree. To build fraud-proofs is enough, since we only need one witness to show that the computation is invalid.

When it comes to the challenger, we do something similar to what we did before. The main difference is the winning condition. We cannot transform choosers as defined in the previous games. We used functions since they have to handle all possible hashes revealed by the other player. Therefore, we define choosers knowing the data and generate their strategies.

```
theorem range_chooser_wins {𝕄 : Type}
    [BEq 𝕄][LawfulBEq 𝕄][HashMagma 𝕄][hash_props : SLawFulHash 𝕄]
    -- DA elements
    (comp_skeleton : BinaryTree SkElem Unit)
    (input_rev input_ch: 𝕄)(output : 𝕄)
    -- Players says that path starts at different places
    (hneq : ¬ input_rev = input_ch)
    -- Players
    (revealer : BinaryTree (Option 𝕄) (Option 𝕄))
    (chooser : BinaryTree 𝕄 𝕄)
    -- Chooser computation is fold plus invariants.
    (chooser_wise : knowing comp_skeleton chooser input_ch output)
    : spl_game { data:= comp_skeleton , res := (input_rev , output) }
      reveler (gen_to_fun_chooser (BinaryTree.map .some .some chooser))
      = Player.Chooser := by ...
```

The above theorem proves that honest choosers win, but does not say anything about how long games are. In the case the arena is a binary complete tree, finding the fraud-proof is logarithmic in the path length. Predicate `knowing` states that the data the chooser has faithfully describes a path from hash `input_ch` to hash `output`, similar to `winning_condition_player`. What it is missing is to connect the winning chooser players in the linear game with the above logarithmic game, we leave that to future work.

## 5 Example: A Simple Protocol

Recent work [12] introduces arbitration games to guarantee properties of batches of transactions proposed by sequencers. These arbitration games now correspond to the execution of concrete specific algorithms known a-priori and not (as in RDoC) to games where one must reason about arbitrary traces of computation from a universal machine.

In [12], a block $b$ is proven to be valid if and only if (1) all transactions in $b$ are valid (which can be check locally by a function valid), (2) there are no duplicates transactions in $b$, (3) no transaction appears in a previous accepted block [12, Section 3.2 (*certified legal batch tag*)]. The definition characterizes the notion of validity completely, and thus, we can also detect invalid blocks by detecting when (at least) one of the above conditions does not hold playing specific games. To see the application of our approach, we focus on the first two properties. The third one can be modeled by encoding the history of accepted blocks as a large Merkle tree or multiple signed Merkle trees, and we leave it as future work.

In our Lean library, we define a structure `Valid_DA` mapping the definition of a valid block as the first two properties plus the correctness of the Merkle tree.

```
structure Valid_DA {α 𝕄 : Type}[DecidableEq α][Hash α 𝕄][HashMagma 𝕄]
          (data : BinaryTree α)(mk : 𝕄)(P : α -> Bool) where
  -- Merkle tree is correct.
  MkTree : data.hash_BTree = mk
  -- All Elements are valid.
  ValidElems : data.fold P Bool.and = true
  -- There are no duplicates.
  NoDup : List.Nodup data.toList
```

We then model the interaction between the two players by their actions. Proposing players generate the DA from a sequence of values (as a tree) and also provides all their strategies before hand. Because of the two kinds of games that can be played there is one data-availability strategy and one strategy for each possible paths. Outside of this model, strategies are played interactively, but we do not have reactive components in our model.

```
structure P1_Actions (α 𝕄 : Type) : Type
 where
 da : BinaryTree α Unit × 𝕄
 dac_str : BinaryTree (Option α) (Option (𝕄 × 𝕄))
 gen_elem_str : {n : Nat} -> Path n -> (Sequence n (Option (𝕄 × 𝕄)) × Option α)
```

The player choosing and challenging dishonest claims have one action per invalid property of the DA. In this example, and because of simplicity, we show the linear games.

```
inductive P2_Actions (α 𝕄 : Type)  : Type where
   -- Player 2 challenging the Merkle tree formation
   | DAC (str : BinaryTree Unit ((𝕄 × 𝕄 × 𝕄) -> Option ChooserMoves))
   -- Player 2 accepts the Markle is well form but there is an invalid element
   | Invalid {n : Nat} (p : α)(path : Path n)
      (str : Sequence n ((𝕄 × 𝕄 × 𝕄) -> Option ChooserSmp))
   -- Player 2 accepts the Markle is well form and all elements are valid
   -- but there is a repeated element
   | Duplicate (n m : Nat) -- There are two paths
      (path_p : Path n) (path_q : Path m)
      -- Strategies to force proposer to show elements.
      (str_p : Sequence n ((𝕄 × 𝕄 × 𝕄) -> Option ChooserSmp))
      (str_q : Sequence m ((𝕄 × 𝕄 × 𝕄) -> Option ChooserSmp))
   -- Player 2 accepts the DA proposed
   | Ok
```

Now we have all the pieces to define the protocol. The protocol is simply an intermediary mechanism invoking games when required and indicates when a proposal should be accepted or no. When implemented in the real-world, this is implemented in a smart contract governing the computational aspects of the system. Here we show a fragment of the protocol when the chooser challenges the creation of the Merkle tree[12].

```
def linear_l2_protocol{α 𝕄 : Type} [BEq α][BEq 𝕄][o : Hash α 𝕄][HashMagma 𝕄]
   (val_fun : α -> Bool) (playerOne : P1_Actions α 𝕄)
   (playerTwo : (BTree α × 𝕄) -> P2_Actions α 𝕄) : Bool
   := match playerTwo playerOne.da with | .DAC ch_str =>
            -- Challenging Sequencer (Merkle tree is not correct)
            match data_challenge_game
                ⟨ playerOne.da.fst.map o.mhash , playerOne.da.snd ⟩
                playerOne.dac_str ch_str with
            | .Proposer => true
            | .Chooser => false
         ...
```

Finally, we prove only valid blocks survive the protocol in presence of honest choosers.

[12] The rest of the protocol can be found in the file "L2.lean" at `https://gitlab.software.imdea.org/martin.ceresa/leanfpgames/-/blob/master/FraudProof/L2.lean`

```
theorem honest_chooser_valid {α 𝕄}
   [BEq 𝕄][LawfulBEq 𝕄][DecidableEq α]
   [o : Hash α 𝕄][m : HashMagma 𝕄][InjectiveHash α 𝕄][InjectiveMagma 𝕄]
   (P : α -> Bool) (p1 : P1_Actions α 𝕄)
   : linear_l2_protocol P p1 (honest_chooser val_fun) ⟺ valid_da p1.da P
```

where `valid_da` states that all properties are valid, i.e. building a `Valid_DA`.

## 6 Conclusions and Future Work

Arbitration games in L2 Optimist Rollups systems are deterrents to prevent fraudulent blocks to consolidate. Such systems rely on the argument that if malicious agents lie, they are caught and penalized. They also rely on agents knowing the public data.

We defined the concepts of DAs, games, players, winning strategies and transformations between games in Lean4. Moreover, we proved that honest players knowing the data, have winning strategies to defend honest claims and to challenge dishonest claims. This is (to the best of our knowledge) the first work to mechanize and prove winning strategies for honest players in the computational model of L2 optimistic rollups. We explored simple notions of equivalence between games: same winning players and mapping winning strategies to winning strategies. Finally, we defined a simple Layer2 protocol and proved it correct.

As future work, we propose the following paths.

**Polish the library.** All proofs not provided due to the limited space are proved. The library is a proof-of-concept, so the first step is to refactor it to get a cleaner code base. Additionally, with the intuition gained, we want to properly define games borrowing formal concepts from Combinatorial Game Theory and Operational Game Semantics [8].

**Generalization.** The main idea of DAs is to hide data and computation and to use Merkle trees to build (verified) blaming chains, fraud proofs. In this work, we focused on formalizing these concepts on trees, but we plan on explore different authenticated-data structures [24]. Another generalization is to have *container* data-types as the arena, computations as *folds* and DAs as predicates over these computations[1].

**Game Description and Interaction Language.** RDoC performs arbitration over the trace of traces of computation from arbitrary progams. However, we can play arbitration games over higher abstractions or programs fixed a-priori. If we are able to decompose validity of bigger DAs into smaller ones, we may be able to play specific games over different algorithms more efficiently. Once players accept the hash to be a Merkle tree, they can engage into specific games. Game `elem_in_tree_is_invalid(path, hash)` challenges the agent posting the DA that element in `path` is invalid. Game `elem_in_tree_is_twice(path_1,path_2, hash)` challenges the agent posting the DA that element in `path_1` is the same as the one in `path_2`, and thus, the block is invalid for repeating elements. To describe all these different situations, we would like to have a nice game language, probably a subset of the *Game Description Language (GDL)*. Ideally, we want to verify the basic components of these games and derive proofs to the more general games.

## References

1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. Applied Semantics: Selected Topics. doi:10.1016/j.tcs.2005.06.002.

2 Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In *Proc. of ISoLA (3)*, number 12478 in LNCS, pages 9–24. Springer, 2020. doi:10.1007/978-3-030-61467-6_2.

3 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. ConCert: a smart contract certification framework in Coq. In *Proc. of the 9th ACM SIGPLAN Int'l Conf. on Certified Programs and Proofs (CPP'20)*, pages 215–218. ACM, 2020. doi:10.1145/3372885.3373829.

4 Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: ContractLarva and open challenges beyond. In *Proc. of the 18th International Conference on Runtime Verification (RV'18)*, volume 11237 of *LNCS*, pages 113–137. Springer, 2018. doi:10.1007/978-3-030-03769-7_8.

5 Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In *Proc. of the FM 2019 International Workshops, Part I*, volume 12232 of *LNCS*, pages 368–379. Springer, 2019. doi:10.1007/978-3-030-54994-7_28.

6 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fourneta, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short paper. In *Proc. of Workshop on Programming Languages and Analysis for Security (PLAS@CCS'16)*, pages 91–96. ACM, 2016. doi:10.1145/2993600.2993611.

7 Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski. Games and strategies using coinductive types. In *International Conference on Types for Proofs and Programs*, 2023.

8 Peio Borthelle, Tom Hirschowitz, Guilhem Jaber, and Yannick Zakowski. An abstract, certified account of operational game semantics. In *European Symposium on Programming*, (to appear in) 2025.

9 Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. Accessed: May 6, 2025. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

10 Ran Canetti, Ben Riva, and Guy N. Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013. doi:10.1016/J.IC.2013.03.003.

11 Margarita Capretto, Martín Ceresa, and César Sánchez. Transaction monitoring of smart contracts. In Thao Dang and Volker Stolz, editors, *Proc. of the 22nd Int'l Conf. on Runtime Verification (RV'22)*, volume 13498 of *LNCS*, pages 162–180. Springer, 2022. doi:10.1007/978-3-031-17196-3_9.

12 Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Pedro Moreno-Sánchez, and César Sánchez. A decentralized sequencer and data availability committee for rollups using set consensus, 2025. doi:10.48550/arXiv.2503.05451.

13 Martín Ceresa and César Sánchez. Fraud Proof Games. Software, version 1., swhId: swh:1:dir:761ba38f606c1b4a0a9e202e6518d092d51ff381 (visited on 2025-05-06). URL: https://gitlab.software.imdea.org/martin.ceresa/leanfpgames, doi:10.4230/artifacts.23003.

14 Martín Ceresa and César Sánchez. Multi: A Formal Playground for Multi-Smart Contract Interaction. In Zaynah Dargaye and Clara Schneidewind, editors, *4th International Workshop on Formal Methods for Blockchains (FMBC 2022)*, volume 105 of *Open Access Series in Informatics (OASIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.FMBC.2022.5.

15 Sylvain Conchon, Alexandrina Korneva, and Fatiha Zaïdi. Verifying smart contracts with Cubicle. In *Proc. of the 1st Workshop on Formal Methods for Blockchains (FMBC'19)*, volume 12232 of *LNCS*, pages 312–324. Springer, 2019. doi:10.1007/978-3-030-54994-7_23.

16 Tyler Crain, Christopher Natoli, and Vincent Gramoli. Redbelly: A secure, fair and scalable open blockchain. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 466–483, 2021. doi:10.1109/SP40001.2021.00087.

17 Joshua Ellul and Gordon J. Pace. Runtime verification of Ethereum smart contracts. In *Proc. of the 14th European Dependable Computing Conference (EDCC'18)*, pages 158–163. IEEE Computer Society, 2018. doi:10.1109/EDCC.2018.00036.

18 J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *J. ACM*, 24(1):68–95, January 1977. doi:10.1145/321992.321997.

19 Abdelatif Hafid, Abdelhakim Senhaji Hafid, and Mustapha Samih. Scaling blockchains: A comprehensive survey. *IEEE Access*, 8:125244–125262, 2020. doi:10.1109/ACCESS.2020.3007251.

20 Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1):189–239, 2006. doi:10.1016/j.apal.2005.05.022.

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

22 Ryan Lavin, Xuekai Liu, Hardhik Mohanty, Logan Norman, Giovanni Zaarour, and Bhaskar Krishnamachari. A survey on the applications of zero-knowledge proofs, 2024. doi:10.48550/arXiv.2408.00243.

23 Ao Li, Jemin Andrew Choi, and an. Long. Securing smart contract with runtime validation. In *Proc. of ACM PLDI'20*, pages 438–453. ACM, 2020. doi:10.1145/3385412.3385982.

24 Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. *SIGPLAN Not.*, 49(1):411–423, January 2014. doi:10.1145/2578855.2535851.

25 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, December 2008. Accessed: May 6, 2025. URL: https://bitcoin.org/bitcoin.pdf.

26 Zeinab Nehaï and François Bobot. Deductive proof of industrial smart contracts using Why3. In *Proc. of the 1st Workshop on Formal Methods for Blockchains (FMBC'19)*, volume 12232 of *LNCS*, pages 299–311. Springer, 2019. doi:10.1007/978-3-030-54994-7_22.

27 Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making fast consensus generally faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 156–167, 2016. doi:10.1109/DSN.2016.23.

28 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety verification of smart contracts. In *Proc of the 41st IEEE Symp. on Security and Privacy (S&P'20)*, pages 1661–1677. IEEE, 2020. doi:10.1109/SP40000.2020.00024.

29 Daian Phil. Analysis of the DAO exploit, 2016. URL: https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

30 Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). doi:10.1016/j.entcs.2015.12.003.

31 Jonas Schiffl, Wolfgang Ahrendt, Bernhard Beckert, and Richard Bubel. Formal analysis of smart contracts: Applying the KeY system. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2020. doi:10.1007/978-3-030-64354-6_8.

32 Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level Language. *CoRR*, abs/1801.00687, 2018. arXiv:1801.00687.

33 Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. SmartPulse: Automated checking of temporal properties in smart contracts. In *Proc. of the 42nd IEEE Symp. on Security and Privacy (S&P'21)*. IEEE, May 2021. URL: https://www.microsoft.com/en-us/research/publication/smartpulse-automated-checking-of-temporal-properties-in-smart-contracts/.

34 Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 16, 1996.

**35** Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain scaling using rollups: A comprehensive survey. *IEEE Access*, 10:93039–93054, 2022. doi:10.1109/ACCESS.2022.3200051.

**36** Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. SoK: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 41–61, 2019. doi:10.1145/3318041.3355457.

**37** Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

**38** Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371119.