BoLD: Bounded Liquidity Delay in a Rollup Challenge Protocol

Mario M. Alvarez, Lee Bousfield, Chris Buckland, Yafah Edelman, Edward W. Felten,
Daniel Goldman, Raul Jordan, Mahimna Kelkar, Harry Ng, Terence Tsao, and Preston Van Loon
Offchain Labs, Inc.

ABSTRACT

Optimistic rollup protocols, if not designed carefully, suffer from *delay attacks*, where an adversary sacrifices stakes to delay confirmation of correct results. These attacks are more consequential on rollups built on a Layer 1 system with weak censorship resistance, such as Ethereum, because the attacker can exploit the generous deadlines offered to possibly-censored parties. This paper summarizes previous protocols; presents the first rollup challenge protocol providing a near-constant upper bound on delay; sketches proofs of safety, progress, and delay bound for the protocol (detailed proofs are available in the full paper); and describes an implementation of the protocol for Arbitrum.

1 INTRODUCTION

1.1 Layer 2 protocols

A Layer 2 protocol (or chain) is a blockchain protocol that is built on top of an underlying Layer 1 chain, relying on the Layer 1 chain as the foundation of its security. Layer 2 chains can offer lower cost or more flexibility than their underlying Layer 1 chains, while benefiting from the security of the Layer 1.[1]

Layer 2 protocols typically operate in two phases: sequencing and execution. Sequencing commits to a sequence of input transactions that will be processed by the chain. Execution processes those transactions and produces results. In this paper we will ignore the details of sequencing, and focus on the execution phase.

1.2 The execution phase

The execution phase operates by consuming transactions from the sequence, one at a time, and applying a deterministic *state transition function* F to each one. Given a state S_i and input message M_i, the resulting state is $S_{i+1} = F(S_i, M_i)$. In practice, transactions produce outputs and may emit events, but we incorporate those into the state S to simplify the exposition.

Because F is deterministic and the sequence of messages is established, any honest party can maintain a correct and up-to-date state by computing each iteration of F locally, without any need for coordination other than acquiring the input message sequence.

1.3 Assertion protocols

In practice, though, it is not enough for every honest party to know the correct state. It is also necessary to *settle* the chain state to the underlying Layer 1 chain (such as Ethereum) so that third parties can verify the state trustlessly without needing to run a full replica of the chain themselves. This is the job of an *assertion protocol.*[2]

An assertion protocol maintains an append-only *confirmed history* of the chain, and stores on the Layer 1 chain a commitment to that confirmed history. Different assertion protocols have their own ways of safely growing the confirmed history.

Assertion protocols for current Layer 2 chains broadly come in two flavors: optimistic and zero-knowledge (ZK). For ZK-based protocols, assertions about execution state are accompanied with a proof of correctness which can be verified by honest parties and by a smart contract on the underlying Layer 1 chain. In many cases, however, such a proof can be expensive and time-consuming to compute, especially if there are a large number of transactions.

Instead, another approach is to proceed *optimistically*; here, parties can post on the Layer 1 chain, assertions about the correct history of the Layer 2 chain without a proof but rather along with *stakes* backing those assertions. Assertions are automatically accepted and confirmed after a time interval if no other party has posted a conflicting assertion. If parties do post conflicting assertions, a dispute or challenge protocol will be triggered to resolve the disagreement. In this paper, we focus on such optimistic protocols.

We note that state-of-the-art optimistic protocols do not rely on incentives or rationality assumptions to guarantee safety and progress; they provide such guarantees in all cases but operate at maximum efficiency when participants are rational.

1.4 Goals and Threat Model

We rely on two main assumptions:

- the underlying Layer 1 chain guarantees safety and liveness, that is, the Layer 1 chain produces correct results for each transaction, and that the Layer 1 chain eventually processes every valid transaction submitted to it; and
- at least one honest party is participating in the challenge protocol.

We assume the Layer 1 protocol may be subject to denial-of-service attacks, within limits. In particular, we assume the Layer 1 will accept every valid transaction within a small constant time δ, except for the existence of a notional "censorship mode" during which the adversary can selectively delay the Layer 1 acceptance of any transactions it likes, while allowing other transactions to be accepted normally. The adversary can enter and depart censorship mode at will, but the total time spent in censorship mode is upper-bounded by a constant C, which is typically taken to be seven days[3]. We refer to C as the *challenge period* for reasons that will be evident later.

[1]We refer to these as Layer 2 and Layer 1 for clarity, but in general one could build a Layer 3 on top of a Layer 2 chain, and so on. For our purposes, it matters only that there is a base layer n and an upper layer $n + 1$ that is built on the base layer.

[2]We use the term "assertion protocol" rather than "rollup protocol" because assertion protocols can be, and are, used with Layer 2 chains that do not meet standard definitions of "rollup."

[3]The seven-day number has broad consensus support in the Ethereum community as a conservative upper bound.

We also assume, conservatively, that the adversary has an advantage in getting their transactions on-chain, so that whenever the adversary is racing against honest parties to get a transaction on-chain first, the adversary will always win.

Challenge protocols have the following goals, in decreasing order of importance:

- guarantee safety and liveness of the Layer 2 chain,
- disincent any dishonest behavior that raises cost for honest parties,
- minimize latency to confirm assertions when all parties are behaving honestly,
- upper-bound the maximum confirmation latency regardless of any dishonest behavior, either unconditionally or as a function of the number of stakes that will be confiscated from adversaries,
- minimize total cost of operation when all parties are behaving honestly,
- minimize cost to honest parties in cases where others behave dishonestly.

1.5 Delay attacks

The goals referring to dishonest behavior are primarily affected by *delay attacks*, in which a malicious party (or a group of colluding parties) acts within a challenge protocol, trying to prevent or delay the confirmation of results back to the L1 chain.

This differs from a denial of service attack, in which the attacker tries to prevent any action from being taken in the protocol. By contrast, in a delay attack transactions continue to occur on the Layer 2 chain, but the attacker tries to delay the confirmation of transaction results and force honest parties to burn Layer 1 gas.

2 PAST PROTOCOLS

This section summarizes two versions of the Arbitrum protocol: the protocol from the initial 2018 academic paper, and the protocol deployed on Arbitrum mainnet as of July 2023.

2.1 Original 2018 Arbitrum protocol

The original 2018 paper about Arbitrum [KGC+18] used essentially the following protocol (ignoring a unanimous mode that is not relevant here). Any party could assert a proposed result. During a time window, any subset of the other parties could challenge the assertion, and the asserter would have to defend their assertion against each challenger, one challenger at a time. At the end of each challenge, the losing party would forfeit their stake.

(Note that here, as in many optimistic protocols, it is necessary to allow multiple parties to oppose the assertion, and to give every challenger its own independent opportunity to defeat the assertion. This is necessary because a malicious challenger can deliberately lose a challenge that they could have won.)

If the challenge period elapsed with no challenge being made, or if the asserter won every challenge, then the assertion would be confirmed and the protocol would move forward. But if the asserter lost any of the challenges, its assertion would be rejected and the protocol state would roll back to the state before the assertion was made.

This protocol does not guarantee liveness, because a malicious participant can endlessly make incorrect assertions, sacrificing a stake each time but leading to an endless cycle of the same assertion being made and rejected, leading to continuous rollbacks and lack of progress.

2.2 One-vs-one challenge sub-protocol

This protocol relied on a one-versus-one challenge sub-protocol in which one party ("Alice") defended the correctness of an assertion and the other party ("Bob") challenged that correctness. Alice's assertion of the form (H_0, n, H_n) means that starting in a state with hash H_0, the virtual machine can execute n instructions, and the resulting state will be H_n.

When Bob challenges Alice's assertion, the protocol requires Alice to bisect her assertion by posting the state hash H_m at the midpoint, $m = \lfloor \frac{n}{2} \rfloor$. Now Alice has implicitly made two assertions, (H_0, m, H_m) and $(H_m, n-m, H_n)$. The protocol now requires Bob to choose one of Alice's two smaller assertions to challenge. Once Bob has done this, the protocol has returned to its original state—Bob is challenging an assertion made by Alice—except that the size of Alice's assertion has been cut roughly in half. The bisect-and-choose protocol now repeats, recursively, until there is a one-step assertion $(H_i, 1, H_{i+1})$ which Alice has made and Bob has challenged.

At this point, Alice must submit a *one-step proof*, proving that her one-step assertion is correct. Her proof can be efficiently checked by the Layer 1 chain. If Alice can produce a valid one-step proof, she wins the challenge; otherwise Bob wins the challenge.

This protocol guarantees that the losing party must have behaved dishonestly at some point, so that an honest party will never lose a challenge. However, there is no guarantee that the winning party is honest—it could be that both parties were dishonest. Therefore no single two-party challenge can establish which result is correct. Instead, the protocol relies on a series of challenges to reach the correct result, by eliminating dishonest parties one by one. Relying on the assumption that at least one party is honest, there will always be an honest party present, so as soon as all remaining parties agree on the result, the protocol can accept that result as correct.

Although the series of challenges is guaranteed to terminate, this does not imply that the initial Arbitrum protocol guarantees progress, because the result of the series of challenges could be to (correctly) reject an attempt to advance the state.

2.3 Production Arbitrum protocol

The original *commercial* Arbitrum protocol—which has been deployed on every version of Arbitrum since 2020—improves the previous protocol in several ways, providing guarantees of liveness in addition to safety. Many details of this protocol are described in the Arbitrum Nitro whitepaper [BBB+]. To our knowledge, this is the only optimistic assertion protocol in production use.

One improvement is to introduce branching. The idea is to allow multiple parties to make competing assertions, and to treat competing assertions as forking the chain. A series of one-versus-one challenges then pits advocates of the branches against each other, and prunes off a branch if all of its advocates have lost a challenge. Eventually only one branch will remain, and this branch will be confirmed.

Timekeeping works as follows. Each assertion in the chain tracks the timestamp when its first child (i.e., successor) was created. Other parties can create additional children. Each child assertion implicitly claims that all of its older siblings are incorrect.

The party who creates an assertion is required to stake on it, and other parties have the option of also staking on it. A party who is staked on an assertion is implicitly also staked on its chain of predecessors, all the way back to the genesis assertion. Because of this implicit staking, a party who is staked on an assertion A can move its stake to any assertion whose predecessor is A.

If two parties are staked on assertions that are siblings, and neither of the two parties is already in a challenge, then a challenge can be initiated between the two parties, where the party staked on the older of the two siblings is defending the correctness of that older sibling, while the other party is challenging that correctness. The party who loses the challenge forfeits their stake and is removed from the protocol.

This protocol includes deadlines for action. First, the deadline for creating child assertions of a parent is one challenge period after the first child has been created. Second, the deadline for staking on an assertion is one challenge period after that assertion is created.

If the deadline for staking on an assertion has passed, and there are no parties staked on that assertion, the assertion is pruned off. (One or more parties must have been staked there in the past, but all of them must have lost challenges.). Any children, grandchildren, or other descendants of the pruned assertion are also pruned away at the same time.

If an assertion is older than one challenge period and has no unpruned siblings, that assertion can be confirmed, representing progress in the protocol.

To force progress, an honest party can post a correct child, if one does not already exist. After that, a limited time will pass before a deadline ensures that no more siblings can be created and no more parties can stake on the siblings that exist. From that point on, the honest party will engage in a series of challenges, defeating and removing parties staked incorrectly, one party at a time. (If multiple honest parties are staked, they can defeat adversaries in parallel.) Once all such parties are removed, the correct child assertion can be confirmed.

2.3.1 Deadlines and the chess clock model. The production protocol imposes time limits on parties in challenges, based on a "chess clock" model. Each party has a conceptual clock, which is initially set to a fixed value (seven days in the production system). When it is a party's turn to act next in a challenge, that party's clock is ticking downward. If a party's clock reaches zero, that party automatically loses the challenge.

The logic behind this approach stems from the threat model. If the adversary is not engaged in censorship, a party should be able to make each move within a short time δ. The only exception is if the adversary is using censorship mode, which can only be the case for a total time of C. It follows that if a challenge requires a maximum of m actions from an honest party, that party will need a total of $C + m\delta$ time for all of its actions.

Because $C \gg \delta$, this chess clock model is much more efficient than the obvious alternative of allowing $C+\delta$ for each action, which was the approach taken in the 2018 Arbitrum protocol.

2.3.2 Delay attacks against the production protocol. The most effective delay attack against this protocol has one malicious party stake on an incorrect sibling, and $N - 1$ malicious parties stake on the correct sibling. No matter how many honest parties are staked on the correct sibling, the attacker will be able to arrange that the incorrectly staked party engages in challenges against its confederates before engaging in challenges against honest parties. (Recall that we are assuming conservatively that the attacker can always get in transactions before the honest parties can.) The confederates will also deliberately lose their challenges, taking as long as possible to do so. (Due to the delay rules, this could take as long as one or two challenge periods per confederate.). Only after all $N - 1$ confederates had sacrificed themselves would the incorrectly staked party be required to fight a challenge against an honest party, which the honest party would win, finally eliminating the incorrectly staked party.

This last attack achieves a delay of roughly N challenge periods, at a cost to the attacker of N stakes.

The cost to honest parties, against this attack, is linear in the number of honest parties, because every honest party would need to stake before the staking deadline.

3 CONSTANT-DELAY CHALLENGE PROTOCOL

An important property of the new Arbitrum challenge protocol we present here is that it can resolve disputes among multiple parties efficiently in a single procedure, rather than relying on the one-versus-one challenges of previous versions. To our knowledge, this is the first practical challenge protocol that supports efficient all-versus-all disputes.

In this section, we will describe the protocol for a single all-versus-all challenge. Later sections will describe how the complete protocol is constructed, using this challenge mechanism as a sub-protocol.

3.1 Design approach

To achieve a tight delay bound, we need to change some fundamental design principles underlying the challenge protocol. Whereas prior designs involved (1) a one-versus-one challenge (2) contested by stakers (3) to eliminate an incorrect branch, the new design uses (1) an all-versus-all challenge (2) contested by claims about execution (3) to confirm the correct branch.

The fact that the "contestants" in a challenge are claims about execution, rather than protocol participants, has significant implications for the design. In particular, this requires that all parties who support a specific claim must be able to "fight as a single team"—even if some of them are malicious—which in turn requires that any protocol action that can be taken "on behalf of the team" must be one that any honest team member would necessarily support.

As an example of that principle, execution claims in the protocol include not only a claimed end state, but also a Merkle root of the entire history being claimed. When such a claim is bisected, the midpoint state provided as part of the bisect action must include a Merkle root of the first-half history. The protocol requires that the bisecting party prove the consistency of these two Merkle roots, or intuitively that the history claimed at the midpoint is a prefix of the history claimed at the endpoint. Assuming that hash collisions are hard to find, an alternate history of the required length that is also a prefix of the endpoint history cannot be found; it follows that the midpoint claim introduced by the bisect action must be a claim that any supporter of the endpoint claim would also support. It follows that it is safe for the protocol to allow any any party to perform a bisection (without worrying about whether the party is malicious). This structure is much more efficient as it allows a party to silently rely on someone else representing its position without having to be concerned that that party could intentionally lose the challenge.

3.2 Execution and Histories

A challenge resolves disagreements about the result of executing a specified state transition function F which, given a state, returns the next state. F represents one step of computation and will be executed repeatedly to progress the computation. (Different implementations could use different notions of "step." A step might be the execution of one virtual machine instruction; or it might be the creation of one block in a blockchain. Section 4 discusses this further.)

For every possible result x, a special state R_x denotes a terminal state where the computation has completed and produced the result x. For convenience we require that for all x, $F(R_x) = R_x$. This allows a computation history that has terminated to be padded with post-termination $F(R_x) \rightarrow R_x$ steps at the end, which allows us to require the initial assertion to claim a fixed number of steps which is a convenient power of two.

We assume the parties agree on the starting state, and that there is a well-known algorithm for hashing any state to a fixed-length string. The correct execution invokes the state transition function $2^{k_{\max}}$ times. We assume that the final state will be a terminated state R_x for some x, and x will be the result of the computation.[4]

3.2.1 Histories and History Commitments.

A *history* is a sequence of states $(S_0, S_1, \ldots, S_n)$ such that S_0 is the agreed-upon initial state. The *height* of the history is $n \leq 2^{k_{\max}}$, the number of steps in the history.

A *correct history* is a history such that for all $i \in \{0, 1, \ldots, n-1\}$, $S_{i+1} = F(S_i)$. Because F is deterministic, there is exactly one correct history at each height.

A *complete history* is a history of height $2^{k_{\max}}$. There is exactly one correct complete history, and if $0 \leq i \leq 2^{k_{\max}}$ the unique correct history of height i is a prefix of the correct complete history.

A *history commitment* is a deterministic cryptographic commitment to a claim about the history. Specifically, it is a pair (n, M), with $0 \leq n \leq 2^{k_{\max}}$, representing a claim that a party knows a history $(S_0, S_1, \ldots, S_n)$ such that the Merkle tree whose leaves are $S_0, S_1, \ldots, S_n$ has a root hash of M. This claim might not be true, that is, a dishonest party might not know a set of leaves that induces the claimed root hash.

Some protocol operations require a proof of consistency between two history commitments. To accomplish this, the prover decomposes the Merkle tree for the lower-height commitment into the minimal set of complete subtrees, and supplies the root hash for each of those subtrees; and the verifier checks that combining those subtrees leads to a matching root hash. The prover also supplies the root hash of a subtree that can be appended to the lower-height commitment's decomposed tree to yield a tree of the other commitment's height[5], and the verifier checks that appending a subtree with the supplied hash leads to the matching root hash for the larger-height commitment. A proof of consistency between two history commitments of length n and $m > n$ requires $O(\log m)$ space and can be checked in $O(\log m)$ time.

We note that a dishonest party might be able to prove two history commitments consistent despite not knowing a sequence of leaves that can be hashed to create either of the commitments.

Lemma 3.1. *If it is infeasible to find a collision in the underlying hash function, then given a history commitment $C = (n, M)$ and a height $k < n$, it is infeasible to find two history commitments $C' = (k, M')$ and $C'' = (k, M'')$ with $M' \neq M''$ such that C' and C'' can both be proven consistent with C.*

3.3 Timing Assumptions

We assume that time proceeds in discrete ticks. Before each tick, the adversary can decide whether or not to censor the tick, with the only constraint being that the adversary cannot censor more than $C_{\max}$ ticks in total.

We define $N(t)$ to be the number of non-censored ticks that have occurred as of time t.

Any action by a party requires at most δ non-censored ticks to complete. We assume the following axioms:

Axiom 1. *An action submitted by an honest party at time t will be completed by time t' if $N(t') \geq N(t) + \delta$.*

Typically the most difficult cases to prove will be those in which malicious actions complete immediately but honest party actions require the maximum latency to complete, although of course the proofs must be correct for all cases.

We define the confirmation time of the challenge to be $T = C_{\max} + (k_{\max} + 2)\delta$.

3.4 Edges

The main data structures in the challenge protocol are *edges*. An edge is identified by a pair of history commitments $((n_0, M_0), (n_1, M_1))$ where $n_0 < n_1$. The *length* of the edge is $n_1 - n_0$. The length will always be a power of two. In fact, it will always be the case that there are non-negative integers i and j such that $n_0 = i \cdot 2^j$ and $n_1 = (i+1) \cdot 2^j$. We say that an edge is at level k if its length is $2^{k_{\max} - k}$, so the possible levels range from level zero for a maximum-size edge, to level $k_{\max}$ for an edge of the minimum length of 1.

The goal of the protocol is to confirm edges that correspond to correct computation, and to prevent the confirmation of any non-correct "top-level" (i.e., level zero) edge.

It is useful to think of the protocol as a contest between edges, which aims to choose correct edges as winners, rather than a contest between the participating parties. Although parties are sometimes required to deposit stakes to deter misbehavior, for most purposes

[4]An implementation may enforce the requirement that the final state has the form R_x by augmenting the state transition function to count the steps executed and force a return of R_{error} after $2^{k_{\max}}$ steps if the state would otherwise be non-terminal.

[5]In general, a Merkle consistency proof might require supplying a sequence of complete subtree hashes to be appended, but in our protocol the commitment heights will always be aligned so that only one subtree hash will be necessary.

the protocol tracks the status of edges but does not identify an edge with any particular party and is indifferent as to which party takes any particular action.

3.4.1 Terminology: types of edges. It is useful to classify edges in the protocol based on their relationship to the correct execution of the underlying iterated state transition function F. The protocol does not "know" which category an edge is in, but honest participants can determine the category of each edge, and the categories are useful for our proofs.

Each edge has a start history commitment and an end history commitment. An edge is a *justifiable edge* if both its start and end history commitments are correct. (A correct history commitment is one that commits to a correct execution history.) It is a *deviating edge* if its start history commitment is correct but its end history commitment is incorrect. Finally, it is an *irrelevant* edge if its start history commitment is incorrect (regardless of its end history commitment).

We prove the protocol correct by proving two main results. First, we prove the safety theorem, which says that no deviating edge can be confirmed. Because any dishonest top-level edge must be deviating, this implies that top-level dishonest edges cannot be confirmed, so no dishonest edge can win the challenge. Second, we prove the completion-time theorem, which says that an honest top-level edge can be confirmed by some deadline.

As the name suggests, the protocol "doesn't care" what happens to irrelevant edges. Indeed, they are never mentioned in the lemmas and proofs. Some irrelevant edges may be confirmed, but this won't matter, because (intuitively) any path of edges from the starting history commitment that includes an irrelevant edge must also contain a deviating edge, which will never be confirmed, and this renders the fate of the irrelevant edge moot.

3.4.2 Rival edges. If two distinct edges have the same starting history commitment and the same length, the two edges will necessarily have different ending history commitments, and we say the two edges are *rivals*.[6]

The following lemmas are proven in the full paper.

Lemma 3.2. *If two edges E and F are rivals, then*

(a) *if P_E is a possible parent of E, and P_F is a possible parent of F, then P_E and P_F are rivals (see section 3.5.2 for the relevant definitions); and*
(b) *if both E and F have been bisected, then E has a child C_E and F has a child C_F, such that C_E and C_F are rivals; and*
(c) *at most one of E and F is justifiable.*

Lemma 3.3. *If P_1, P_2 are both possible parents of E, and $P_1 \neq P_2$, then P_1 and P_2 are rivals.*

Lemma 3.4. *If E_l, E'_l are both edges at level l and are possible ancestors of E_k and E'_k, respectively, and E_k and E'_k are rivals, then E_l and E'_l are rivals. Note that E_k and E'_k being rivals guarantees that they are at the same level, which we call k; assume $k > l$.*

[6]Implementation note: The protocol needs to be able to detect whether an edge has any rivals, but does not need to keep track of who those rivals are. So it is sufficient to maintain a table with an entry for each (starting history commitment, length) pair that matches any edge, with the table entry recording whether there is more than one matching edge.

Lemma 3.5. *If P_l, P'_l are two edges at level l and are both possible ancestors of E_k (where E_k is at level $k > l$), then P_l and P'_l are rivals (or equal).*

3.4.3 Protocol state. The protocol state consists of a set of edges. In the beginning, this set is empty.

3.4.4 Creating top-level edges. From this starting point, any party can create a level-zero edge, which represents a claim about the complete history of the computation. The level-zero edge will have the form $((0, M_0), (2^{k_{\max}}, M))$ where M_0 is the Merkle hash of the sequence (S_0). The party must provide a Merkle proof that M is consistent with the agreed-upon S_0 and with $S_{2^{k_{\max}}} = R_x$ for some arbitrary x. We call x the (claimed) *result* of the computation.

We define the start time of the challenge, notionally $t = 0$, to be the time when the first level-zero edge is created.

Any party may create a new level-zero edge, at any time $t < T$.

3.4.5 Staking. A party who creates a level-zero edge must submit a stake S which they should expect to lose if their edge is incorrect. If there are L level-zero edges, the total stakes will be LS.

When a level-zero edge is confirmed as the challenge winner, the party who created that level-zero edge receives a refund of their stake. Any other stakes are confiscated and added to a public goods fund. Parties who staked on other level-zero edges get nothing.

An extension to the protocol, described later, can track the gas costs expended by parties in the protocol, and reimburse the gas costs spent defending the winning level-zero edge and its descendants. (Gas costs spent defending other edges are not reimbursed.) The costs of this can be covered by the funds captured into the public goods fund.

3.5 Actions in the protocol

3.5.1 Bisection. If an edge has length greater than one, and has not been confirmed, and has any rivals, and has not been bisected, then any party can do a *bisection* action on the edge. If the edge is $(H_0, H_1) = ((n_0, M_0), (n_1, M_1))$, the party must provide a "midpoint" history commitment $H_m = (\frac{n_0+n_1}{2}, M_m)$, and the party must prove that that midpoint history commitment is consistent with the endpoint history commitment (n_1, M_1).

Because the midpoint must be proven consistent with the endpoint, Lemma 3.1 implies that the bisecting party has no discretion to choose the midpoint history commitment.

If there is not already an edge (H_0, H_m), such an edge is created. Additionally, the edge (H_m, H_1), which will not already exist, is created.

The protocol records the fact that the original edge now has two children (H_0, H_m) and (H_m, H_1). The original edge still exists.

3.5.2 Parents and Children. In a bisection we say that (H_0, H_m) and (H_m, H_1) are the *children* of (H_0, H_1). If A is a child of B, we say that B is a *parent* of A. (Note that an edge might have more than one parent. Every edge not at level 0 has at least one parent.)

If a bisection of edge B could create edge A, we call B a *possible parent* of A, and call A a *possible child* of B (regardless of whether B has actually be bisected). Generally, we will use *parent* or *child* to talk about an edge that exists in the protocol at some point in timer, and use *possible parent* or *possible child* to talk about such edges regardless of whether and when they are created. We use *(strict) ancestor* to refer to the (reflexive)-transitive closure of the parent relation, and *(strict) possible ancestor* to refer to the (reflexive)-transitive closure of the possible-parent relation. Finally, we say that edge A is *(strict) (possible) descendant* of edge B is B is a (strict) (possible) ancestor of A.

3.5.3 Presumptive edges. An edge is *presumptive* if it has no rivals. Intuitively, a presumptive edge will eventually be confirmed if nothing more happens in the protocol.

3.5.4 Presumptive Timers. The protocol maintains timers for each edge with the purpose, intuitively, of tracking for how long the edge and its ancestors have been presumptive.

The protocol tracks the creation time $t_{\text{creation}}(E)$ of each edge E, as well as $t_{\text{rival}}(E)$, the earliest creation time of any rival of E (or $\perp$ if E never had a rival).

The *local presumptive timer*, $\lambda_E(t)$, of edge E at time t is defined to be:

- 0, if $t < t_{\text{creation}}(E)$ (or E is never created),
- $t - t_{\text{creation}}(E)$, if $t_{\text{creation}}(E) \leq t$ and ($t < t_{\text{rival}}(E)$ or $t_{\text{rival}}(E) = \perp$),
- $\max(t_{\text{rival}}(E) - t_{\text{creation}}(E), 0)$, otherwise.

The *path timer*, $\tau_E(t)$, of edge E at time t is defined as follows:

$$\tau_E(t) = \lambda_E(t) + \max_{P \in \text{parents(E)}} \tau_P(t),$$

where the max is taken to be zero if E has no parents (which will be the case iff E is at level zero)[7].

The protocol does not track the path timer, but a party can prove a lower bound on the path timer of an edge by providing a sequence of ancestors (a parent, a grandparent, etc.) of the edge.

Notional timers useful for proving properties of the protocol: Some further timer types are not used by the protocol, but are useful for proving properties of the protocol.

The *potential timer*, $\pi_E(t)$ of edge E at time t is defined as

$$\pi_E(t) = \lambda_E(t) + \max_{P \in \text{possible ancestors(E)}} \pi_P(t),$$

The *honest timer* $\gamma_E(t)$ of edge E at time t is defined as

$$\gamma_E(t) = \lambda_E(t) + \max_{P \in \text{parents(E)} \wedge P \text{ is justifiable}} \gamma_P(t),$$

Lemma 3.6. *For all edges E and times t, $\gamma_E(t) \leq \tau_E(t) \leq \pi_E(t)$.*

This follows from the fact that the set of justifiable parents for E is a subset of the set of parents for E, which in turn is a subset of the set of possible ancestors or E.

Each of these timers has a different purpose. The path timer corresponds to the most natural notion of timer inheritance. We use the potential timer when proving safety, since it provides an upper bound on the path timer. Finally, the honest timer represents a simplification of the protocol implementation: it requires less bookkeeping to maintain than the path timer, and we show (in the full paper) that the key properties of the protocol still hold if the honest timer is used.

[7]This definition might seem at first to be ill-formed, but it is not, because the τ on the left-hand side refers to an edge at level $k+1$ whereas the τ on the right-hand side of the definition refers to edges at level k. This can be formalized by including a level number in the definition of τ so there is a separate τ function for each level, which renders the definition well-formed (because the number of levels is bounded). Readers who prefer such a formulation can imagine that the definition has been specified with levels, and the levels are elided in the text where obvious from context.

3.6 Confirmation

An edge can be *confirmed* if its path timer is at least T; or if it has children and both children have been confirmed; or if it has been one-step proven.

3.6.1 One-step proofs. If an edge has a length of one, it corresponds to a single step of computation. At this point a special *one-step proof* procedure can be used to prove the edge correct. The nature of that procedure will differ depending on the use case, but the procedure must be able to verify a proof that a particular length-one edge is correct.

When an edge is one-step proven, it becomes confirmed.

3.7 How the challenge protocol ends

If any level-zero edge is confirmed, the challenge ends and that level-zero edge is declared the winner of the challenge.

Typically this will happen because some party notices that all of the childless descendants of some level-zero edge are confirmed or have path timers of at least T. The party will confirm the tree of descendants bottom-up, moving upward through the family tree. This will result in the level-zero edge being confirmed, ending the challenge. Note that separate branches of the descendant tree can be confirmed concurrently.

3.8 An Example

Figure 1 gives a high-level view of what an execution of the protocol could look like. Suppose the correct history commitment is (H_0, H_{1024}). At t_1, a deviating top-level edge (H_0, H'_{1024}) is created by an adversary. The honest parties need to act to prevent (H_0, H'_{1024}) from being confirmed. To do so, they begin by creating a justifiable top-level edge (H_0, H_{1024}) at t_2. They then bisect this edge, leading to the creation of two child edges, (H_0, H_{512}) and (H_{512}, H_{1024}) at t_3.

In this example, the adversary's history commitment agrees with the honest history until the next-to-last step, so the first child of (H_0, H_{1024}) and (H_0, H'_{1024}) will be the same edge (H_0, H_{512}). The adversary now needs to act to prevent confirmation of the honest edge (H_{512}, H_{1024}), and so bisects its top-level edge, yielding (H_{512}, H'_{1024}) at time t_4. Now the honest party needs to bisect the justifiable edge (H_{512}, H_{1024}), yielding two new edges (H_{512}, H_{768}) and (H_{768}, H_{1024}).

Again the adversary, seeking to prevent (H_{768}, H_{1024}) from accumulating enough time that it can be confirmed, bisects (H_{512}, H'_{1024}). This process continues until the honest party has created an edge of length 1, (H_{1022}, H_{1023}). This edge corresponds to the single step of computation on which the honest party and the adversary disagree. The adversary continues "chasing" the honest party as before, eventually creating the edge (H_{1022}, H'_{1023}).

Since these edges are of length 1, and only the honest party's edge is correct, the honest party can then confirm (H_{1022}, H_{1023}) through a one-step proof. If the adversary takes no other action, all the other first-child edges created so far will never obtain rivals, so their timers will continue to increase until they can be confirmed by timer. The other second-child nodes will all be confirmable because of the confirmability of their children: their first child will be confirmable (as just discussed), while the second child will be confirmable either by its children having been confirmed (or by one-step proof or presumptive timer, in the case of (H_{1023}, H_{1024})).

3.9 Correctness and Completion

We need to prove that if our assumptions hold, there is a "winning" strategy available to any honest party which will ensure that:

- no incorrect level-zero edge can ever be confirmed, and
- a correct level-zero edge can be confirmed within some bounded time.

The winning strategy operates by first creating a correct (and justifiable) level-zero edge (if one does not already exist), and then by simultaneously defending all justifiable edges. In doing so, the strategy may sometimes create new justifiable edges, which it will also defend.

To defend an edge E means:

- If E has been confirmed, no further action is necessary.
- Otherwise, if E can be confirmed, do so.
- Otherwise, if E is presumptive or E has children, no other action is needed at present.
- Otherwise, E has at least one rival and has no children. If E has length one, do a one-step proof of E. Otherwise bisect E.

A proof of the following safety lemma appears in the full paper.

Lemma 3.7. *If any party is following the winning strategy, then no deviating edge will be confirmed.*

A proof of the following completion-time lemma appears in the full paper.

Lemma 3.8. *If any party is following the winning strategy, then the correct level-zero edge will be confirmed at time t such that $t \leq C_{\max} + (3k_{\max} + 4)\delta$.*

3.10 Worst-case cost for honest parties

We now turn to the question of how much L1 gas the honest parties must expend to win a challenge. The following lemmas are proven in the full paper.

Lemma 3.9. *If exactly one deviating edge is made at level zero during a challenge, the honest parties can win the challenge while making at most $3k_{\max} + 2$ transactions.*

Lemma 3.10. *If L stakes are deposited for a challenge, the honest parties can win the challenge while making at most $(3k_{\max}+2)(L-1)$ transactions.*

Because the number of transactions by honest parties scales linearly in the number of stakes that will be confiscated, it will be possible to use confiscated funds to reimburse the costs of all honest transactions, provided the stake S is large enough to pay for $3k_{\max} + 2$ transactions.

3.10.1 Protocol extension: Reimbursing honest costs. We can extend the protocol to do the bookkeeping needed to support reimbursement of honest transaction costs. To do this we add to each edge a data structure counting the number of transactions that each party has done on behalf of that edge and its descendants.

Credits are computed as follows:

- When a level zero edge is created, initialize the new edge's credit counter by crediting the party whose transaction created the edge.
- When an edge is bisected, the party whose transaction did the bisection gets a credit on the bisected edge.
- When an edge is confirmed, the party whose transaction did the confirmation gets a credit on the newly confirmed edge. If the edge has children, the credits on both children are added to the credits on the newly confirmed edge.

The protocol ends when a level zero edge is confirmed. Each credit on the confirmed level zero edge (i.e. the edge that "won" the challenge) is then reimbursed. Other edges get no reimbursement, since such an edge either was rejected by the protocol, or had its credits accumulated up into the level zero edge that was reimbursed.

4 MULTI-LEVEL CHALLENGES

In practice, the protocol cannot use a single-level challenge as described in the previous section, because the cost of doing so would be too large, due to the need to stop and recompute a hash of the virtual machine state after every instruction of emulated execution.

Instead, the protocol uses a multi-level structure of challenges and sub-challenges, where a one-step fork in a challenge is resolved by carrying out a sub-challenge at the next level down. Within each challenge or sub-challenge, the basic protocol is executed as described in the previous section. It is only at the boundary between levels—when starting a challenge or when using resolving a one-step fork in a challenge—that the multi-level structure becomes relevant.

4.1 General scheme for multi-level challenges

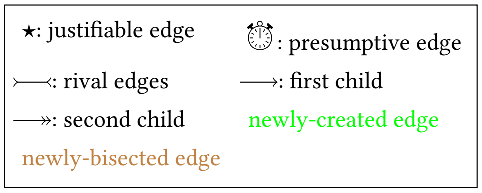
Before diving into the details of multi-level challenges in the Arbitrum protocol, we will first describe a simple two-level challenge scheme, to illustrate what is possible. The two-level scheme will generalize naturally to more than two levels.

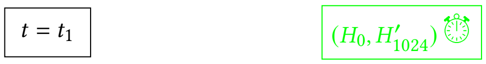
We start by specifying a lower transition function F_l, defined over a lower state space, with a maximum number of steps 2^{k_l}. The upper transition function F_u is defined as $F_u(x) = F_l^{2^{k_l}}(x)$, that is, the upper transition function is the result of applying the lower transition function 2^{k_l} times.

The upper challenge is a challenge over 2^{k_u} steps of the upper transition function F_u. If there is a one-step fork in the upper-level challenge, that fork represents a disagreement over the correct value of $F_u(x)$ for some x. This is resolved by starting an equivalent lower challenge over 2^{k_l} steps of the lower transition function F_l. The winner of the lower challenge will determine which edge of the upper one-step fork is the winner, and that winning edge will be confirmed, just as if it had been one-step proven in the single-level protocol.

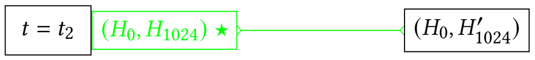
If there are multiple one-step forks in the upper challenge, a separate lower challenge is started to resolve each one. These lower challenges are separate and independent of each other, and are carried out concurrently.

Any one-step forks in a lower challenge are resolved by one-step proofs, as in the single-level case.

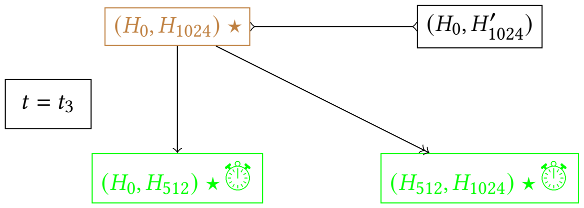




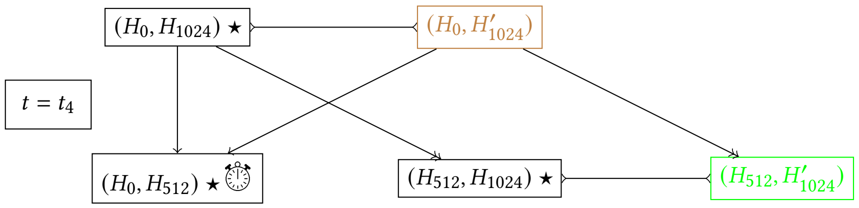
(a) Adversary creates a dishonest toplevel edge, (H_0, H'_{1024}). H'_{1024}'s next-to-last state is incorrect. It is presumptive as no other toplevel edges exist.



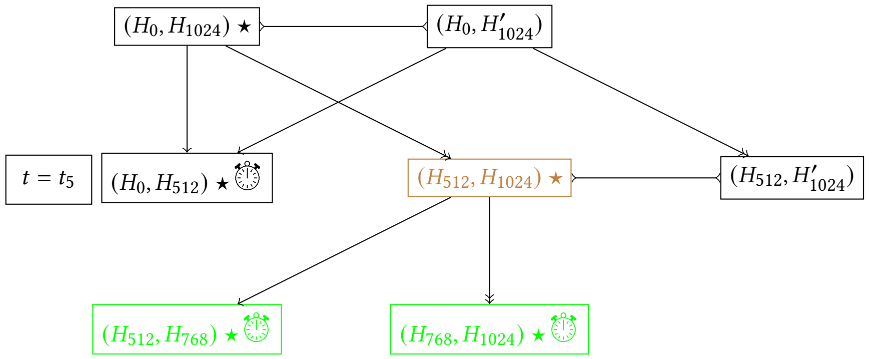
(b) Honest party responds by creating an honest (justifiable) toplevel edge, (H_0, H_{1024}). H_{1024} is the correct history.



(c) Honest party bisects (H_0, H_{1024}) to get two honest edges (H_0, H_{512}) and (H_{512}, H_{1024}) corresponding to the two halves of the execution. Both become presumptive as there are no other "competing" edges of the same size.



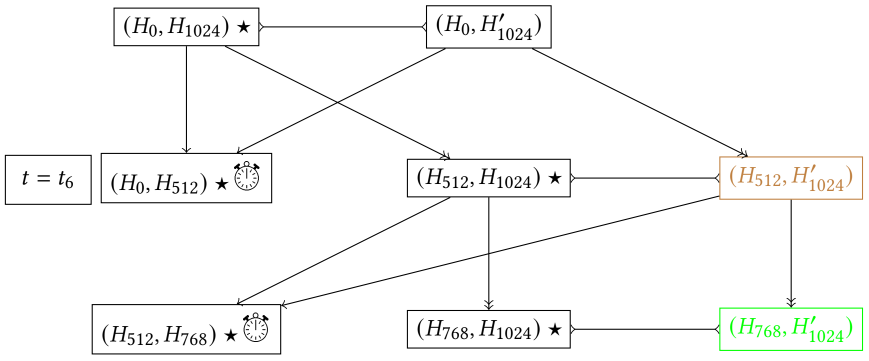
(d) Adversary responds by bisecting their toplevel edge, (H_0, H_{1024}). In this example, the adversary is correct about the first half of execution, so its first child is also (H_0, H_{512}). The second half, (H_{512}, H'_{1024}), disagrees with the honest party's edge (H_{512}, H_{1024}), making them rivals. Neither is presumptive now.



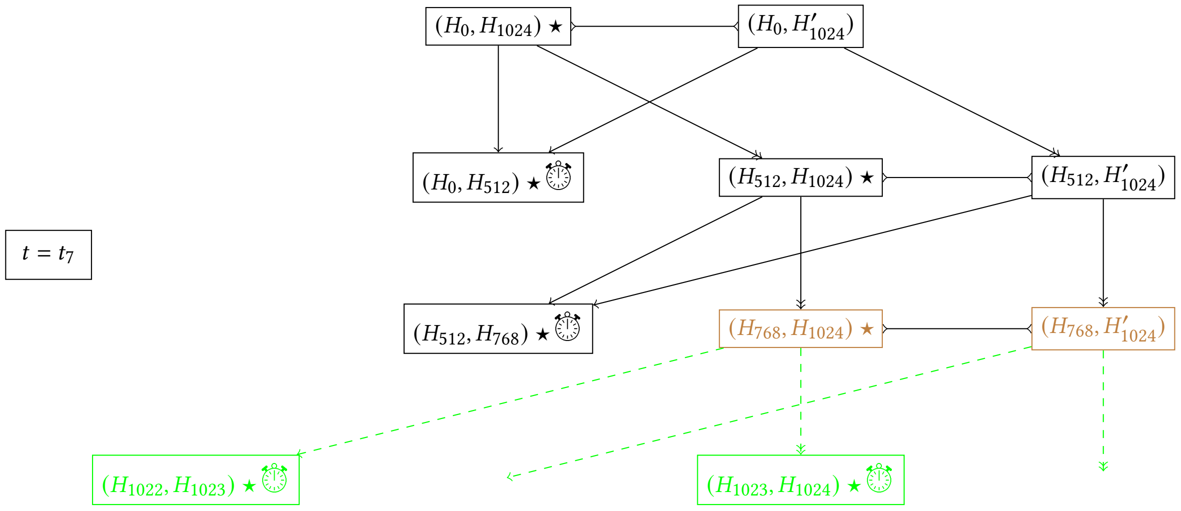
(e) Honest party responds again, bisecting its edge (H_{512}, H_{1024}), with which the adversary disagrees. Again, the result is two presumptive edges of half the size.

Figure 1: A Schematic Example of the Challenge Protocol, t=1-5

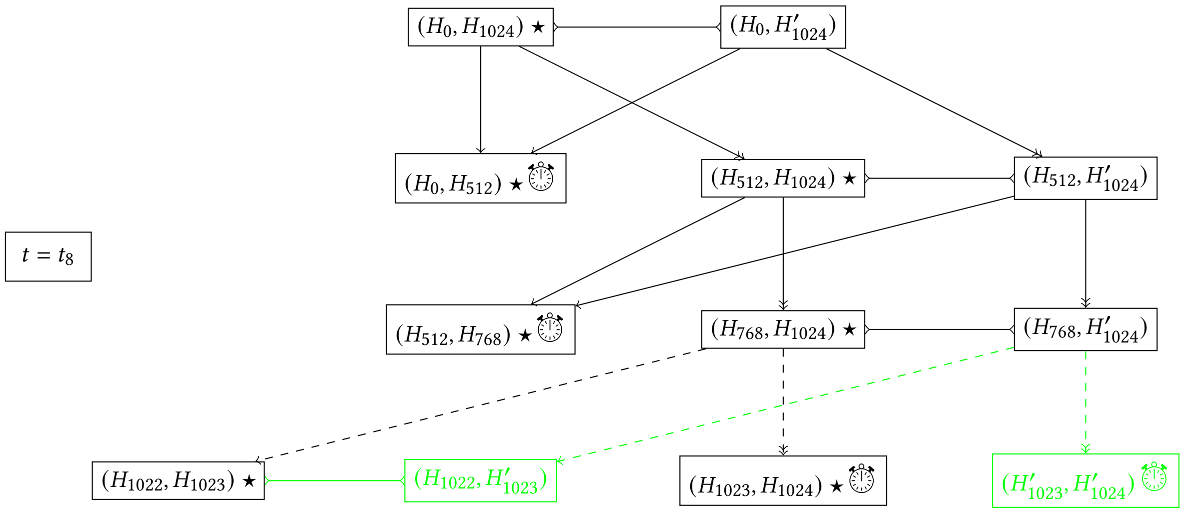
Each box represents an edge. In this example, we have a dishonest party that disagrees with the correct execution only at the next-to-last step of execution. Edges marked with a star are justifiable (i.e., honest and correct). The correct edge created at t_7 can be one-step proven and thereby confirmed. Eventually the correct edges created at t_3 and t_5 will accumulate enough time on their presumptive timers to be confirmed. At that point, the remaining correct edges can be confirmed in bottom-up treewise fashion, confirming each correct edge because its two children are confirmed. The protocol will finish when the correct edge at the top left is confirmed.



(f) The adversary bisects its edge (H_{512}, H_{1024}). It disagrees with the fourth quarter of the overall computation (the second half of the second half), so its first child equals the honest edge (H_{512}, H_{768}), but its second child (H_{768}, H'_{1024}) is a rival to the honest edge (H_{768}, H_{1024}).

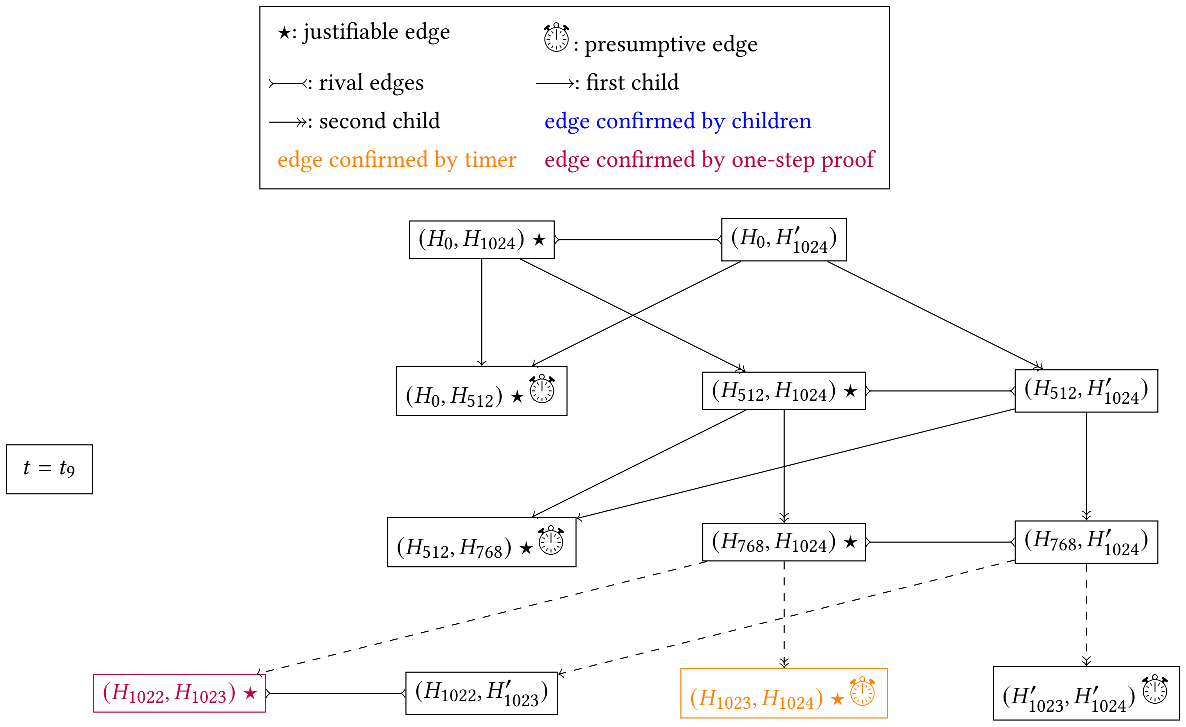

$(H_0, H_{1024}) \star$
(H_0, H'_{1024})
$(H_0, H_{512}) \star$
$(H_{512}, H_{1024}) \star$
(H_{512}, H'_{1024})
$t = t_7$
$(H_{512}, H_{768}) \star$
$(H_{768}, H_{1024}) \star$
(H_{768}, H'_{1024})
$(H_{1022}, H_{1023}) \star$
$(H_{1023}, H_{1024}) \star$

(g) The honest party and adversary continue bisecting until the honest party creates edges (H_{1022}, H_{1023}) and (H_{1023}, H_{1024}). These are one-step edges and cannot be bisected any further.

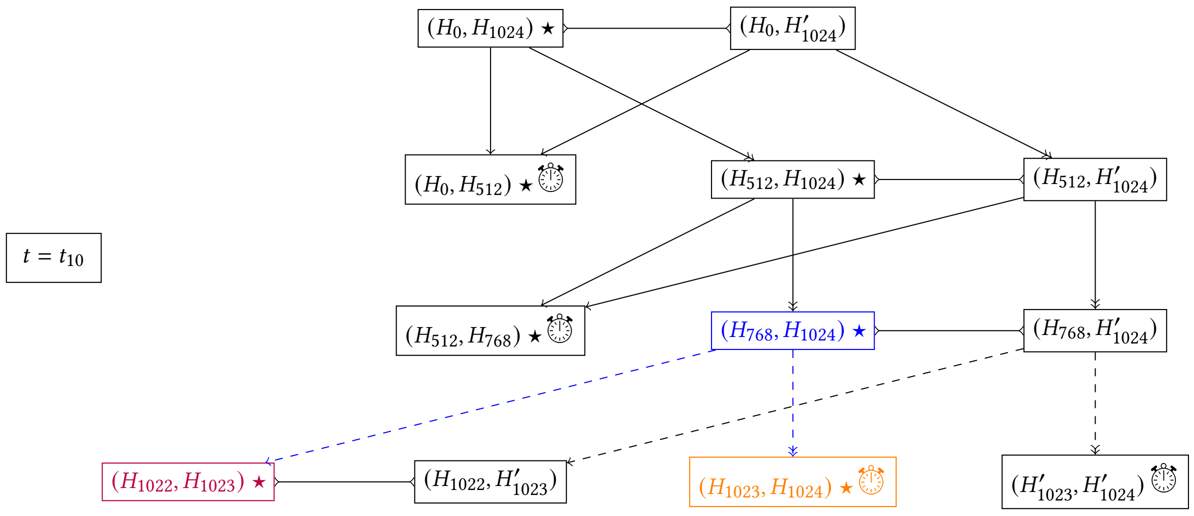


(h) The adversary finishes bisecting, producing (H_{1022}, H'_{1023}) and (H'_{1023}, H'_{1024}). The dispute is now narrowed to a single computation step (the next-to-last step, in this example).

Figure 1: A Schematic Example of the Challenge Protocol, t=6-8

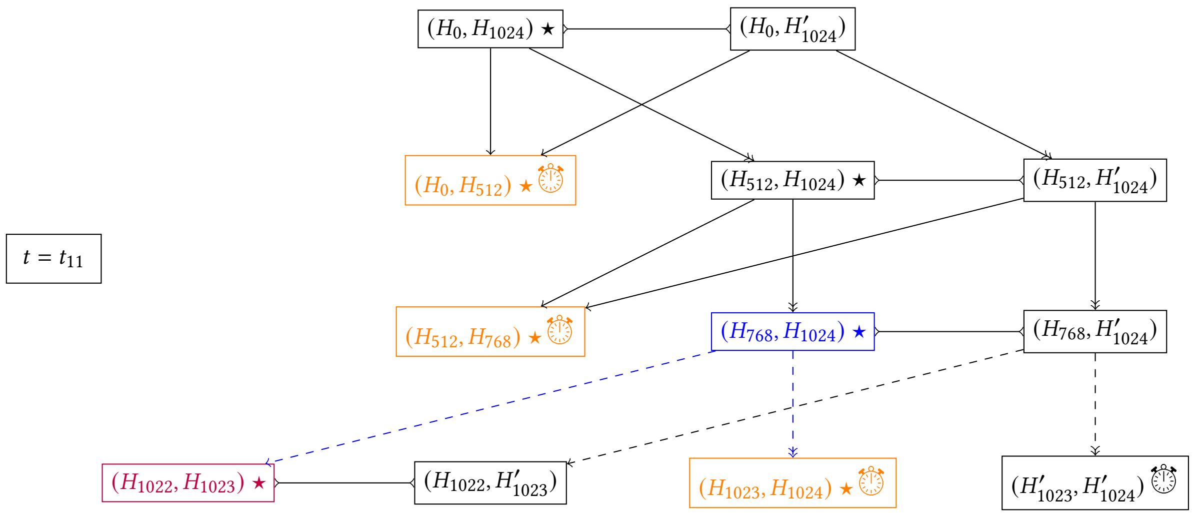


(i) Because (H_{1022}, H_{1023}) and (H_{1023}, H_{1024}) are one-step edges, they can be confirmed with one-step proofs. (H_{1023}, H_{1024}) has no rivals ((H'_{1023}, H'_{1024}) has a different starting history, so it is not a rival). Thus it can also be confirmed once its presumptive timer grows large enough (as is the case in this example). The honest party will take actions to ensure both are confirmed. The attentive reader will note that (H'_{1023}, H'_{1024}) can also be confirmed once its presumptive timer gets large enough. The honest party can prevent this with more bisections, but does not need to do so: (H_{768}, H'_{1024}) will never be confirmable as it is not presumptive, not a one-step edge, and has a descendant (H_{1022}, H'_{1023}) that can never be confirmed.

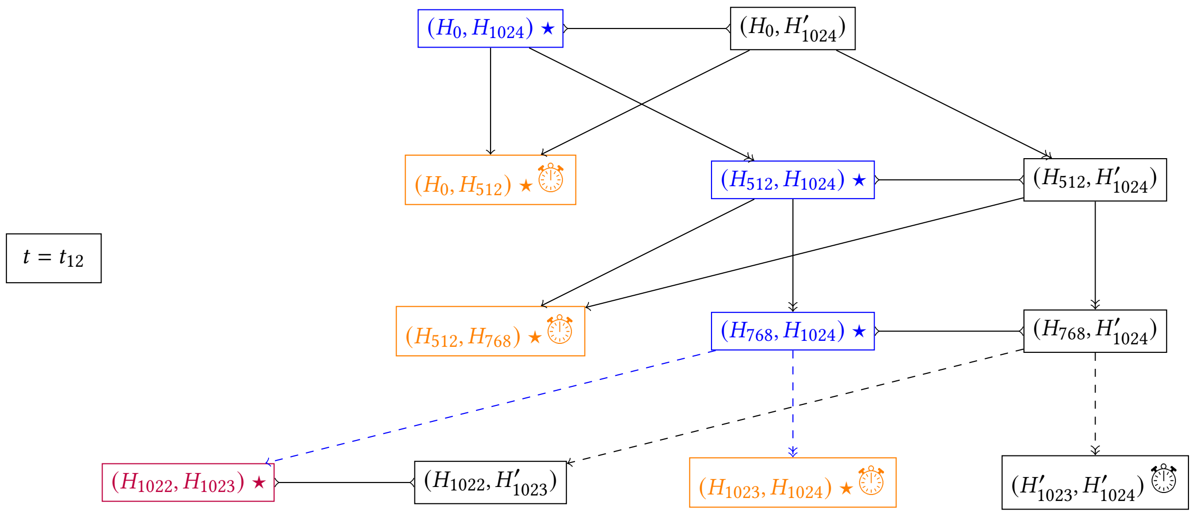


(j) The immediate honest parent of (H_{1022}, H_{1023}) and (H_{1023}, H_{1024}) (not shown) can now be confirmed, as both its children are confirmed. This process repeats until eventually (H_{768}, H_{1024}) can be confirmed due to its children being confirmed. The first child of (H_{768}, H_{1024}) will be confirmed once its timer grows large enough; the second child will be confirmed due to its children being confirmed.

Figure 1: A Schematic Example of the Challenge Protocol, t=9-10



(k) Eventually, (H_{512}, H_{768}) and (H_0, H_{512}) will have their presumptive timers grow large enough that they can be confirmed.



(l) Finally, (H_{512}, H_{1024}) can be confirmed due to all its children being confirmed. After this, (H_0, H_{1024}) can also be confirmed as its children are all confirmed. The honest party has now caused (H_0, H_{1024}) - the correct top-level edge corresponding to the entire correct execution history - to be confirmed. The challenge protocol ends.

Figure 1: A Schematic Example of the Challenge Protocol, t=11-12

Importantly, the timers in the two challenges are linked. When a lower challenge is started, any party can create a level-zero edge in the lower challenge. That level-zero edge must be proven consistent with the ending history state of one of the edges of the upper-level one-step fork, and the new lower level-zero edge's timer will be initialized to equal the timer of that upper-level edge.

In effect, the multi-level protocol is treating the creation of the level-zero edge in a lower challenge as "just another bisection" from the standpoint of timekeeping, and in the basic logic of the protocol.

The proofs of the single-level protocol can be adapted to the two-level case, if we replace $k_{\max}$ by $k_u + k_l$. With more than two levels, the proofs can be adapted with $k_{\max}$ replaced by the sum of k_i over the levels i.

The main advantage of a multi-level challenge is that it reduces the amount of hashing that is necessary. A single-level challenge requires $O(2^{k_{\max}})$ hashes, to compute the state hash of each state and then to compute a Merkle root over the states. By contrast, in a two-level challenge with $k_u = k_l = \frac{k_{\max}}{2}$, and m lower challenges, the total hashing required scales as $O((m+1)2^{\frac{k_{\max}}{2}})$.

4.2 Multi-level challenges in the Arbitrum protocol

There are three levels of challenge in the updated Arbitrum challenge protocol. At the top level are *block challenges*, where each step is the creation of one new block of the Arbitrum chain. Any one-step fork in a block challenge is resolved by starting a *big-step challenge*, where a step represents the execution of 2^{20} instructions in the Arbitrum block creation function. Any one-step fork in a big-step challenge is resolved by starting a *single-step challenge*, where a step represents the execution of one instruction. Any one-step fork in a single-step challenge is resolved by a one-step proof covering a single instruction of execution.

This structure was chosen to reduce execution and hashing costs. The top-level block challenge allows parties to execute the block creation function in a locally efficient fashion, for example by just-in-time compiling the WASM [HRS+17] code of the block creation function. Below the block level are two levels (big-step and single-step) that pertain to instruction-by-instruction execution of WASM code. An Arbitrum block requires up to 2^{43} instructions to be executed [8]. The execution is divided into two levels to reduce hashing costs, so that no more than 2^{23} states need to be hashed in any one history commitment.

5 IMPLEMENTATION

A working implementation of the protocol is available at TODO.

As is common for protocols built on Ethereum-compatible chains, our implementation consists of a set of on-chain components, implemented in Solidity, that enforce the rules of the protocol; and a set of off-chain components that track the state of the protocol and execute the honest strategy.

5.1 Implementation Challenges

This section outlines some issues that arose in implementing the protocol.

5.1.1 Subchallenge Granularity. In the implementation, disputes are created at the level of blocks on the L2 blockchain, where each block represents a deterministic execution of the L2 chain's state transition function. The state transition function is represented as a sequence of WASM instructions in the current implementation. At the top-level challenge, validators disagree over a potentially large range of blocks. For example, there could be an assertion at block number 300 and two conflicting ones at number 400.

Because the goal of the protocol is to reach a "one-step proof", and each block in our implementation can encompass up to 2^{43} opcodes of WASM execution, creating history commitments over individual opcodes would be computationally infeasible. Instead, the implementation uses a three-level structure. First, validators narrow down disagreement to a single block, then, they execute a "large-step challenge" over ranges of 2^{20} opcodes, and finally, they execute a "small-step challenge" over a single range of 2^{20} individual opcodes to reach a one-step proof. This three-level structure allows for a more practical and performant implementation.

5.1.2 Ensuring Honest Party Liveness. Another challenge in the implementation has to do with ensuring liveness, assuming the chain cannot be censored for longer than the assumed $C_{\max}$. Although the onchain protocol guarantees liveness if actions are correctly taken by an honest party, our offchain client must also be free of bugs that would prevent it from taking honest actions dictated by the honest strategy, and from encountering an unrecoverable crash.

5.1.3 Using Efficient Data Representation. A third challenge has to do with propagating edge confirmations and efficiently confirming edges in the implementation. Confirmations must propagate all the way from the deepest subchallenge level to the top, level zero edge in order to finalize a challenge. Because of the compute-constrained environment that Ethereum provides, we must be efficient in how the onchain confirmation works. Recursively confirming the entire tree of honest edges as a one-shot transaction in pure Solidity will often be infeasible, as the caller could hit the gas limit. Instead, we split up confirmations into different approaches: (1) by one step proof, (2) by unrivaled timer, (3) by confirmed children, (4) by claim. As a general practice, we offload as much computation as possible to the offchain caller rather than calculating paths in the ancestor tree in Solidity. When a party wishes to confirm an edge by its unrivaled timer, it must provide a list of ancestors that contribute to that edge's path timer to complete the action.

5.1.4 Honest Validator Client. For the offchain component of the protocol, we built an honest validator client in Go which can interact with the Ethereum contracts to initiate and win challenges against malicious parties. The L2 blockchain that the validators settle has a state transition function written in Go, and this state is critically important to produce valid history commitments and proofs for making challenge moves. The Go programming language also provides useful concurrency primitives, making it easy for our implementation to support managing many challenges simultaneously. Because the honest party's strategy is to defend all honest edges, even those it did not originally create, the implementation logically resolves concurrent challenges.

5.2 High-Level Architecture

Our honest validator implementation has three high-level roles: posting claims about its local chain state to L1, initiating challenges on invalid assertions on L1, and defending honest edges in ongoing challenges. Figure 2 shows the architecture of the honest validator implementation.

The client is constantly scanning for newly posted assertions to Ethereum, and checks them against its local state. If the client disagrees, it will initiate a challenge by creating a new level zero edge via an Ethereum transaction. This will spawn an edge which can be tracked as a goroutine. Each edge goroutine then makes challenge moves according to its status until all of the tracked edges are finally confirmed.

5.2.1 Local Execution Backend. The validator software has access to a local execution backend, which provides it with history commitments and proofs of states at any level of granularity needed to make challenge moves. For example, it can produce a Merkle commitment of ranges of opcodes between blocks B and B+1, which are needed for creating level zero edges. The honest validator will always produce valid commitments. In the implementation, the state can be represented as a Merkle tree in-memory for fast access at each level of granularity. This is feasible because of known bounds: a Merkle tree over the worst case number of opcodes at the lowest level of granularity will take up approximately 32Mb of

[8]The Arbitrum protocol rejects any block that would require more than 2^{43} instructions to compute.

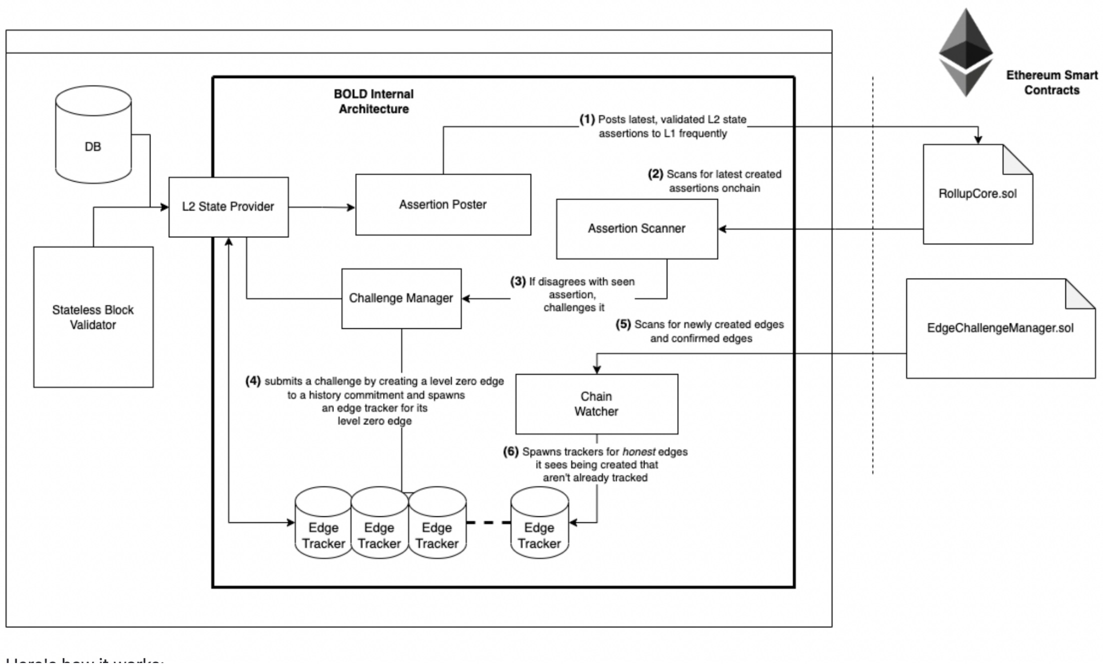

Ethereum Smart Contracts
BOLD Internal Architecture
DB
L2 State Provider
Assertion Poster
Assertion Scanner
Stateless Block Validator
Challenge Manager
Chain Watcher
Edge Tracker
Edge Tracker
Edge Tracker
Edge Tracker
RollupCore.sol
EdgeChallengeManager.sol
(1) Posts latest, validated L2 state assertions to L1 frequently
(2) Scans for latest created assertions onchain
(3) If disagrees with seen assertion, challenges it
(4) submits a challenge by creating a level zero edge to a history commitment and spawns an edge tracker for its level zero edge
(5) Scans for newly created edges and confirmed edges
(6) Spawns trackers for honest edges it sees being created that aren't already tracked

Figure 2: Block diagram of the honest validator design.

memory. At the lowest level, the backend allows for creating one step proofs of computation for the honest participant to win.

5.2.2 Initiating Challenges. When it comes to challenging, the validator code polls the protocol's events on Ethereum to check for newly posted assertions to cross-check their claims against its local state. If there is a disagreement, the validator will submit the assertion it believes is correct if that assertion does not already exist. Then, it will submit a level zero edge by producing its history commitment against its local state. Once the edge is created, or if it already exists, the validator will track the edge in a thread which will manage its lifecycle and make challenge moves as needed.

The client is constantly scanning for newly posted assertions to Ethereum, and checks them against its local state. If the client disagrees, it will initiate a challenge by creating a new level zero edge via an Ethereum transaction. This will spawn an edge which can be tracked as a goroutine. Each edge goroutine then makes challenge moves according to its status until all of the tracked edges are finally confirmed.

5.2.3 Managing Multiple, Concurrent Challenges. As the core primitive of the protocol is a challenge edge, and given that each entity running a validator client could be participating in many concurrent challenges, we choose to track edges' lifetimes and perform actions on edges via goroutines, which are the concurrency primitive provided by Go, equivalent to green threads.

5.3 Tracking Edges

At any given time, T, entities in the protocol can take actions on edges, such as bisecting them, opening sub-challenge level zero edges, one step proving edges, or confirming edges via various means. The implementation tracks each edge in a green thread, known as a goroutine, and uses a finite state machine (FSM) to capture its series of states until it reaches confirmation. Whether or not an edge can take an action and the type of action it must take is captured nicely via a FSM, as it allows for retrying states in case of failure.

Each challenge move is highly asynchronous, prone to potential frontrunning, and is error prone as it requires a connection to an Ethereum chain backend. A transaction could fail, for example, due to low gas payments, or the chain client's RPC connection may temporarily be unavailable. In such a scenario, the edge will retry the same state until it can transition to the next.

In our finite state machine, an edge's terminal state is reached once the edge is confirmed, as that is the sole purpose of each edge tracker goroutine. Once a terminal state is reached, the goroutine can be cleaned up. As each edge is being tracked in its own goroutine, with its own FSM, eventually all of an honest validator's edges will get confirmed and a challenge will end.

5.4 Crash Recovery and Initial Startup

In case the validator process crashes, or the validator is joining late into an ongoing challenge, it will deterministically catch up to the state it needs to be in by querying all chain events since the latest confirmed assertion. By scanning for edges that have been created and spinning up edge tracker goroutines for them if the validator agrees, it will safely track all edges it needs to defend.

5.5 Edge Confirmations

One of the most complex parts of the implementation is to support confirmations of edges by their unrivaled path timers. Edges can be confirmed by time if they have a local timer that exceeds a challenge period. However, ancestors of edges' timers also contribute towards this calculation. This means that each edge tracker goroutine would need to keep track of the unrivaled timers of each edge and their ancestors by computing them on the fly. As this could be a large, recursive call, and because the validator client must participate in potentially many, concurrent challenges, this approach is untenable.

Instead, the implementation defines a "challenge watcher" singleton service, which reads all edge added events in the protocol and keeps track of honest branches of edges. This service frequently computes and refreshes the path timers of edges it tracks. Because we only need to care about honest edges for the purpose of confirmation, we can keep their recursive path timers updated, and an edge tracker can then perform a quick lookup to check if an edge can be confirmed by time.

In order to make a transaction for this confirmation, however, the edge tracker must also retrieve the list of ancestors for an edge. This can also be retrieved via a recursive call from the Challenge-Watcher service, as needed. Once all edges are confirmed by time, confirmation will propagate to the top-level and the challenge will end as an assertion then becomes confirmed.

5.6 Gas consumption

The challenge protocol functionality is encapsulated in the EdgeChallengeManager smart contract. Since it is only expected to be used in the rare dispute case it has not been optimised for gas consumption. Instead, the focus of the code is to be easy to read, audit, test and maintain.

Function	Gas
createLayerZeroEdge	193,000
bisectEdge	401,000
confirmByChildren	12,000
confirmByTime	180,000
confirmByOneStep	21,700 + one step specific costs
confirmByClaim	18,800

Table 1: Gas costs for EdgeChallengeManager functions. One step proofs have different costs depending on the actual step being proven, so here we only provide the overhead associated with all steps.

6 FORMAL VERIFICATION

To further build confidence in this protocol, we have begun work on formally verifying a model of the protocol using the Isabelle/HOL proof assistant [NPW02]. Our goal is to produce machine-checked proofs of the core safety and confirmation-time lemmas (lemmas 3.7 and 3.8, respectively). Currently, we have a complete formal model of the protocol, as well as machine-checked proofs of lemmas relating to the structure of and relationships between edges (lemmas 3.2, 3.3, 3.4, 3.5, and several others not explicitly listed in this paper).

6.1 Differences Between Model and Specification

We aim to minimize the differences between the Isabelle/HOL formal model and the specification given in this paper (Section 3), in order to maximize confidence that the results obtained there apply to the protocol as described here. However, the two are not identical. The most significant difference between them is the way in which histories are modeled. To avoid having to reason about the cryptographic commitments used by the protocol to express histories, the formal model abstracts histories to lists of states (from which such commitments could be produced).

We believe this is a reasonable simplification to make: as long as the assumption of the impracticality of finding collisions in the underlying hash function holds, the only risk to the soundness of the formal model (with respect to the specification given in this paper) is that honest parties might use the extra information to which they have access in the Isabelle/HOL model (i.e., the sequence of states corresponding to every commitment) that they would not otherwise have access to. (Allowing the adversary to make use of this extra information just leads to a stronger result than necessary). By ensuring that the Isabelle/HOL implementation of the honest strategy corresponds to the strategy described in this paper (in particular, that it does not make use of the extra information), we avoid this potential problem.

The formal model addresses only a single level of the challenge protocol (as opposed to the multi-level extension discussed in Section 4). It also does not address gas costs, staking, reimbursement, and associated bookkeeping (described in Section 3.10.1).

6.2 Differences Between Model and Implementation

The goal of the Isabelle/HOL formalization is to build confidence in the protocol, rather than the concrete implementation. Though Isabelle/HOL can be configured to generate code from formal specifications corresponding to executable programs, we have not designed the formal model with this in mind. The model could be seen as *partially-executable* implementation, but contains pieces from which code cannot be naturally generated.

For example, in order to make reasoning simpler, the formal model represents the protocol state using a mathematical set of edges (with associated timers and other metadata), rather than using a "real" data structure, and assumes honest parties can tractably perform certain operations (e.g. finding maximum timers for the purposes of computing path timers) over them. Additionally, we deliberately minimize the amount of state kept in our formal model of the protocol: rather than, for instance, caching and storing the time at which each edge obtained a rival, we model honest parties as recomputing this value each time they need it. The actual implementation (see Section 5) is designed to be more efficient, leading to more complicated state representation and logic that would make it harder to reason about directly.

7 CONCLUSION

The BoLD protocol, described in this paper, achieves the best known delay bound for confirming chain results for an optimistic challenge protocol, and limits the worst-case work required of honest parties to be linear in the number of adversary stakes confiscated. The protocol is practical, and we describe a complete implementation that is in security audit as of the paper submission date.

REFERENCES

[BBB$^+$] Lee Bousfield, Rachel Bousfield, Chris Buckland, Ben Burgess, Joshua Colvin, Edward W. Felten, Steven Goldfeder, Daniel Goldman, Braden Huddleston, Harry Kalodner, Frederico Arnaud Lacs, Harry Ng, Aman Sanghi, Tristan Wilson, Valeria Yermakova, and Tsahi Zidenberg. Arbitrum nitro: A second-generation optimistic rollup.

[CRR13] Ran Canetti, Ben Riva, and Guy N Rothblum. Refereed delegation of computation. *Information and Computation*, 226:16–36, 2013.

[HRS$^+$17] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[KGC$^+$18] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security '18)*, pages 1353–1370, 2018.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[NT22] Diego Nehab and Augusto Teixeira. Permissionless refereed tournaments. *CoRR*, abs/2212.12439, 2022.