BoLD: Fast and Cheap Dispute Resolution

Mario M. Alvarez
Offchain Labs, Inc., Clifton, NJ, USA

Ben Berger
Offchain Labs, Inc., Clifton, NJ, USA

Chris Buckland
Offchain Labs, Inc., Clifton, NJ, USA

Edward W. Felten
Offchain Labs, Inc., Clifton, NJ, USA

Raul Jordan
Offchain Labs, Inc., Clifton, NJ, USA

Akaki Mamageishvili
Offchain Labs, Inc., Clifton, NJ, USA

Aman Sanghi
Offchain Labs, Inc., Clifton, NJ, USA

Terence Tsao
Offchain Labs, Inc., Clifton, NJ, USA

Henry Arneson
Offchain Labs, Inc., Clifton, NJ, USA

Lee Bousfield
Offchain Labs, Inc., Clifton, NJ, USA

Yafah Edelman
Offchain Labs, Inc., Clifton, NJ, USA

Daniel Goldman
Offchain Labs, Inc., Clifton, NJ, USA

Mahimna Kelkar
Offchain Labs, Inc., Clifton, NJ, USA

Harry Ng
Offchain Labs, Inc., Clifton, NJ, USA

Victor Shoup
Offchain Labs, Inc., Clifton, NJ, USA

Abstract

BoLD is a new dispute resolution protocol that is designed to replace the originally deployed Arbitrum dispute resolution protocol. Unlike that protocol, BoLD is resistant to delay attacks. It achieves this resistance without a significant increase in onchain computation costs and with reduced staking costs.

2012 ACM Subject Classification Security and privacy → Distributed systems security

Keywords and phrases Optimistic rollups, fraud proofs

Digital Object Identifier 10.4230/LIPIcs.AFT.2024.2

Related Version *Full Version*: `https://arxiv.org/abs/2404.10491` [1]

1 Introduction

In this paper, we introduce BoLD, a *dispute resolution protocol*: it allows conflicting assertions about the result of a computation to be resolved. It is designed for use in a Layer 2 (L2) blockchain protocol, relying on a Layer 1 (L1) blockchain protocol for its security (more generally, it may be used with any parent chain in place of L1 and any child chain in place of L2).

In an optimistic rollup protocol such as Arbitrum, a dispute resolution protocol like BoLD operates as a component of a broader "rollup" protocol, in which validators post claims about the correct outcome of executing an agreed-upon sequence of transactions. These claims are backed by stakes posted by the claimants. If multiple competing claims are posted, the protocol must choose one of them to treat as correct. The goal of BoLD and comparable protocols is to determine, among a set of competing claims about the correct outcome of execution, which of the claims is correct.

Let us state more precisely the problem to be solved. We begin with a starting state, S_0, on which all parties agree. We assume that a commitment to S_0 has been posted to L1. (In this paper, all commitments are non-hiding, deterministic commitments that will typically be

6th Conference on Advances in Financial Technologies (AFT 2024).
Editors: Rainer Böhme and Lucianna Kiffer; Article No. 2; pp. 2:1–2:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

implemented using some sort of Merkle tree.) There is also an agreed-upon *state transition function* F, which maps state S_{i-1} to $S_i = F(S_{i-1})$ for $i = 1, \ldots, n$. In practice, the function F may be determined by the state transition function of a specific virtual machine, together with a specific sequence of transactions that has also been posted to L1; however, these details are not important here.

Any party may then compute $S_1, \ldots, S_n$ and post an assertion to L1 that consists of a commitment to S_n. Of course, such an assertion may be incorrect, and the purpose of a dispute resolution protocol is to allow several parties to post conflicting assertions and identify the correct one. Such a dispute resolution protocol is an interactive protocol that makes use of L1:

- each "move" made by a party in the protocol is posted as an input to a smart contract on L1;
- this smart contract will process each move and eventually declare a "winner", that is, it will identify which one of the assertions made about the commitment to S_n is correct.

Participation in the protocol requires resources:

- *staking:* tokens required for "staking", as specified by the dispute resolution protocol;
- *gas:* L1 tokens required to pay for "L1 gas costs", that is, the cost associated with posting assertions and subsequent moves to L1;
- *computation:* offchain compute costs incurred by the parties who participate in the dispute resolution protocol.

As for staking, the dispute resolution protocol specifies exactly how much and when stakes must be made. When the smart contract declares a winner, some stakes will be confiscated ("slashed") and some will be returned to the staking parties: corrupt parties may have some or all of their stake confiscated, while honest parties should get all of their stake returned to them. The staking requirement serves to discourage malicious behavior. In addition, confiscated stakes may be redistributed to honest parties to cover their L1 gas costs and offchain compute costs, or simply as a reward for participating in the protocol. These stakes will be held in escrow by the smart contract.[1]

We make the following assumptions:

- L1 provides both *liveness* and *safety*, that is, every transaction submitted to it is *eventually processed* and is *processed correctly*;
- at least one honest party participates in the dispute resolution protocol.

We wish to model the following types of attacks.

Censorship attacks. While we assume L1 provides liveness and safety, we assume that it may be subject to *intermittent censorship attacks*. That is, an adversary may be able to *temporarily* censor transactions submitted by honest parties to L1. During periods of censorship, we assume the adversary may still submit its own transactions to L1.

Ordering attacks. Even if the adversary is not actively censoring, we assume that the adversary may determine the order of transactions posted to L1 (for example, placing its own transactions ahead of honest parties' transactions in a given L1 block).

Resource exhaustion attacks. Even though a protocol might ensure that all honest parties are "made whole" after the protocol succeeds, the adversary may try to simply exhaust the resources of the honest parties (the staking, gas, and computation resources mentioned above) so the honest parties can no longer afford to participate in the protocol.

[1] In a (typical) run of the protocol in which there are no challenges to a correct assertion, there will be no confiscated stakes available, and so some other source of funds must be used to compensate honest parties for their (minimal) costs in participating in the protocol.

- **Delay attacks.** The adversary may try to delay the dispute from being resolved within a reasonable amount of time. In such a delay attack, the adversary might try to keep the protocol running for a very large number of moves without resolution – such an attack may become a resource exhaustion attack as well.

To mitigate against a resource exhaustion attack, a well-designed dispute resolution protocol should force the adversary to marshal many more resources than required by the honest parties. Similarly, to mitigate against a delay attack, a well-designed dispute resolution protocol should force an adversary who attempts to delay the protocol to expend a huge amount of resources.

The BoLD protocol is designed as a replacement of the originally deployed Arbitrum dispute resolution protocol. It makes more efficient use of resources than the original Arbitrum protocol while providing a much stronger defense against delay attacks.

The rest of the paper. Section 2 briefly reviews the original Arbitrum dispute resolution protocol, sketches the main ideas of BoLD, and discusses how BoLD improves on the original Arbitrum protocol. Section 3 describes a formal attack model for dispute resolution. Section 4 describes BoLD in its simplest form, which we call ***single-level BoLD***. Section 5 describes a version of BoLD, which we call ***multi-level BoLD***, that reduces some of the *offchain* computational costs of the honest parties. This is the version of BoLD that will replace the originally deployed Arbitrum dispute resolution protocol. Section 6 discusses details regarding gas, staking, and reimbursement.

2 Overview and Comparison to Prior Work

We provide a brief overview of BoLD and a comparison to prior approaches in this section. More details can be found in the full version of the paper [1].

2.1 Arbitrum Classic

By Arbitrum Classic, we refer to the protocol deployed on Arbitrum in 2020. Note that this version differs in some ways from the original 2018 paper [2].

Arbitrum classic allows parties to post assertions to L1 accompanied by some stake (up until a designated ***staking deadline***). Parties with conflicting assertions may then challenge each other. In each such two-party ***challenge subprotocol*** instance, one party defends their assertion against another party who challenges their assertion. When this subprotocol instance finishes, one of the two parties will win and the other will lose. The dispute resolution protocol allows many such subprotocol instances to proceed, even concurrently. The protocol ends when all remaining parties are staked on the same assertion – which is declared to be the "winner". The challenge subprotocol guarantees that any honest party will win in any instance of the subprotocol in which it participates. This ensures that the protocol will eventually terminate and declare the correct assertion to be the winner.

In the two-party challenge subprotocol, also called the "bisection game", one party Daria, defends her assertion, against a challenger Charlie. Daria's assertion is of the form $(0, n, S_0, S_n)$; this denotes the assertion that executing the state transition function F n times on state S_0 leads to state S_n (in practice, the assertion will include commitments to the state rather than the state itself). WLG, we let n be a power of two. When Charlie challenges this assertion, the subprotocol game requires Daria to "bisect" her assertion by posting two smaller assertions $(0, n/2, S_0, S_{n/2})$ and $(n/2, n, S_{n/2}, S_n)$ of half the size each. Following this, Charlie is now required to pick one of these smaller assertions to challenge. The game continues in a similar fashion until Charlie's challenge is one a "one-step" assertion – i.e., of the form $(i, i+1, S_i, S_{i+1})$. At this point, Daria must submit a proof that this one-step assertion is correct which can be efficiently checked by the Layer 1 chain. If this proof is valid, Daria wins the two-party challenge subprotocol and Charlie loses; otherwise, Charlie wins and Daria loses.

The challenge subprotocol guarantees that any honest party will win whenever in which it participates. *However, a corrupt party may stake on the correct intial assertion, but still intentionally lose a challenge.* Because of this, and because honest parties cannot reliably identify other honest parties, each honest party must stake on the correct assertion.

To maintain liveness, the protocol will enforce that the total time taken for each party's moves is smaller than some amount called the "challenge period." This can be done using a "chess-clock" approach where a party's clock runs when it is her turn to make a move. Note that each subprotocol will end within a maximum of 2 challenge periods. The challenge period will be set to be large enough to accommodate any censorship.

There are a number of ways the dispute resolution protocol could orchestrate the challenge subprotocols, depending on the amount of concurrency allowed. In the full concurrency option, there is no limit on the concurrent execution of challenge subprotocol instances. While this guarantees fast resolution, it is susceptible to a resource exhaustion attack.

The deployed version of Arbitrum Classic implements a less concurrent orchestration method, by which each party is allowed to engage in at most one challenge subprotocol instance at a time, although many such subprotocol instances may run concurrently. This method reduces the risk of a resource exhaustion attack significantly, but is instead susceptible to a delay attack. For this reason the deployed Arbitrum Classic has limited participation in the protocol to a permissioned set of parties.

A complete description and discussion of Arbitrum Classic can be found in the full version of the paper [1].

2.2 From Arbitrum Classic to BoLD

A primary motivation that separates BoLD was from Arbitrum Classic is to able to give the following guarantees:

- resolves disputes within a bounded amount of time (independent of the number of parties or staked assertions),[2] unlike Arbitrum Classic and
- has gas and staking costs that scale better than Arbitrum Classic.

BoLD achieves these goals using the following ideas.

2.2.1 Trustless cooperation

Instead of just committing to the final state S_n, parties must commit to the entire execution history $S_1, \ldots, S_n$. This can be done compactly using a Merkle tree whose leaves are the individual commitments $Com(S_i)$ for $i = 1, \ldots, n$. With this approach, a similar type of bisection game as in Arbitrum Classic can be designed with the property that there is only one (feasibly computable) justifiable bisection move that can be made at any step. In other words, if a party submits a correct initial assertion, all smaller assertions obtained by

[2] "BoLD" is an acronym that stands for *Bounded Liquidity Delay*, emphasizing the fact that it is resistant to delay attacks, unlike partially-concurrent Arbitrum Classic. The term "liquidity" refers to the fact that once the protocol terminates, funds sent from L2 to L1 become available on L1.

bisection must also be correct. This means that the correct assertion need only be staked once, and that the honest parties can build on the work of any *apparently honest* parties that make correct assertions, *even if those parties turn out not to be honest parties after all.* In this sense, all honest parties can work together as a team, although they do not have to trust each other or explicitly coordinate with one another.

With just this one idea, one could fairly easily modify Arbitrum Classic to get a protocol with the following properties. The honest parties in aggregate need to make just one staked assertion. If the corrupt parties together make N_{A} staked assertions, then within a bounded amount of time (independent of N_{A}) the honest parties can disqualify all incorrect staked assertions using L1 gas proportional to N_{A}. Indeed, this protocol will terminate within two challenge periods.

2.2.2 A streamlined protocol

BoLD takes the above idea and turns it into a more elegant and streamlined protocol. We design a new execution pattern in which all assertions (both original assertions and "smaller" assertions obtained from bisection) are organized as nodes in a dynamically growing graph. The edges in the graph represent parent/child relationships corresponding to bisection. In this approach, there are no explicit one-on-one challenges nor associated "chess clocks". Instead of "chess clocks", each node in the graph has a "local" timer that ticks so long as the corresponding assertion remains unchallenged by a competing assertion – so higher local timer values indicate that the corresponding assertion is in some sense more likely to be correct (or incorrect but irrelevant to the protocol's outcome). The values of these local timers are aggregated in a careful way to ultimately determine which of the original assertions (which are roots in this graph) was correct. This idea yields the a version of BoLD that we call ***single-level BoLD***, which maintains the same fast termination time of two challenge periods.

2.2.3 Multi-level refinement

The downside of the above approach to trustless cooperation is that the offchain compute cost needed to compute the commitments $Com(S_i)$ for $i = 1, \ldots, n$ and build a Merkle commitment from them may be unacceptably high in practice when n is large. To address this, we introduce a *multi-level refinement strategy* – the resulting protocol is called ***multi-level BoLD***. For example, suppose $n = 2^{55}$. Very roughly speaking, in two-level BoLD, we might execute single-level BoLD using the "coarse" iterated state transition function $F' = F^{2^{25}}$, which only requires 2^{30} state hashes, and narrow the disagreement with the adversary to one iteration of F' which is equivalent to 2^{25} iterations of F. A recursive invocation of single-level BoLD over those iterations of F would then "refine" the disagreement down to a single invocation of F which could then be proven using a one-step proof. Each such recursive invocation would require just 2^{25} state hashes. A naive realization of this rough idea would potentially double the amount of time it would take to run the dispute resolution protocol to completion – and even worse, for L-level BoLD, the time would get multiplied by a factor of L. Most of this time is due to the built-in safety margins that mitigate against censorship attacks. However, by carefully generalizing the logic of single-level BoLD, and in particular the logic around how timer data on nodes is aggregated, we obtain a protocol that enjoys the same fast termination time as single-level BoLD, namely, two challenge periods. Based on our experience in implementing BoLD, we find that setting $L = 3$ reduces the offchain compute costs to a reasonable level.

3 Formal attack model

In the real world, there may be many parties that participate in the protocol, but we formally describe the attack model in terms of just two parties: the ***honest party*** and the ***adversary***.

- The *honest party* in our formal model represents the actions taken in aggregate in the real world by honest individuals who are correctly following the protocol. While one might consider protocols that require communication and coordination among honest individuals, our protocol does not require this. That said, in our protocol, some amount of coordination between honest individuals may be useful in terms of efficiently distributing the resources necessary to carry out the protocol.
- The *adversary* in our formal model represents the actions taken in aggregate in the real world by corrupt individuals who may well be coordinating their actions with one another, and may also be influencing the behavior of the L1 itself, at least in terms of L1 censorship and ordering attacks.

At the beginning of the challenge protocol, we assume that both the honest party and adversary are initialized with the initial state S_0 and a description of the state transition function F. We assume that a commitment to S_0 and a description of F are also recorded on L1. The challenge protocol proceeds in rounds. While time plays a central role in our attack model and our protocol, we shall simply measure time in terms of the number of elapsed rounds. (As an example, if the L1 is Ethereum, a round might be an Ethereum block.)

In each round $t = 1, 2, \ldots,$ the honest party submits a *set* Submit_t of moves to L1. After seeing the set Submit_t, the adversary specifies the precise *sequence* Exec_t of moves to be executed on L1 in round t. The sequence Exec_t may contain moves submitted by the honest party in this or any previous round, as well as arbitrary moves chosen by the adversary. The sequence Exec_t is also given to the honest party, so that its value is available to the honest party in its computation of Submit_{t+1}. We do not place any limit on how many moves may be submitted to or executed on L1 in a round. We assume that the appropriate party (either the honest party or the adversary) is charged for the gas required to execute each move on L1.

We introduce a ***nominal delay parameter*** δ that models the maximum delay between the submission of a move and its execution under normal circumstances, that is, without censorship. An adversary may choose to ***censor*** any given round t. To model censorship, we define the following rules that the adversary must follow. At the beginning of the attack, we initialize Pool, a set of (move, round-number) pairs, to the empty set. In each round t:

- For each move in Submit_t, we set its ***due date*** to $t + \delta$ and add the move, paired with its due date, to the set Pool.
- If t is designated a ***censored round*** by the adversary, then we increment the due date of every move in Pool (including those just added).
- The adversary chooses some moves in pool to include in Exec_t, which are then removed from Pool, subject to the rule:

  any move in Pool whose due date is equal to t must be included in Exec_t*.*

We introduce another parameter, $C_{\max}$, which we call the ***censorship budget***. We require that the adversary censors at most $C_{\max}$ rounds during the attack game. This is our way of modeling the assumption that censorship attacks cannot be carried out indefinitely.

4 The BoLD protocol: single-level version

In this section, we describe the BoLD protocol in its purest form, which we call ***single-level BoLD***.

4.1 Preliminaries

To simplify things, we assume that the parties involved seek to prove a computation of $n := 2^{k_{\max}}$ steps, for $k_{\max} \geq 0$. We assume that a commitment to S_0 is already stored on L1, and we denote this by H_0.

The goal is to have a commitment to S_n posted to L1 in such a way that the dispute resolution protocol ensures that this commitment is correct (under well defined assumptions).

4.2 The protocol graph

As the protocol proceeds, a data structure will be built on L1 that represents a directed acyclic graph $\mathcal{G}$. The structure of $\mathcal{G}$ will be described below. Initially, the protocol graph $\mathcal{G}$ is empty, and grows over time. Participants in the challenge protocol will make moves that lead to the creation of new nodes and edges – the details of these moves are described below in Section 4.3.

4.2.1 The syntax of a node

We begin by defining the syntax of a node. A node specifies a ***base commitment*** and a ***span commitment***. The base commitment is supposed to be (but may not be) a commitment to an initial sequence of states, while the span commitment is supposed to be (but may not be) a commitment to an adjacent sequence of states. A node also specifies the length of these two sequences.

More precisely, a node is a tuple

$$(\mathit{nodeType},\ \mathit{lbase},\ \mathit{lspan},\ \mathit{base},\ \mathit{span}), \tag{1}$$

where *base* and *span* are the base and span commitments of the node, while *lbase* and *lspan* specify the corresponding commitment lengths. As will become evident, the value *lbase* will always be an integer in the range $0, \dots, n-1$, while the value *lspan* will always be a power of two dividing n; moreover, it will always hold that *lbase* is a multiple of *lspan* and $\mathit{lbase} + \mathit{lspan} \leq n$. The value *nodeType* is a flag, equal to either

- `regular`, in which case we say the node is a ***regular node***, or
- `proof`, in which case we say the node is a ***proof node***.

The role of this flag will be described below.

Correct construction. Suppose the correct sequence of states is $S_0, S_1, \dots, S_n$. We say the node (1) is ***correctly constructed*** if the base commitment *base* is the root of a Merkle tree whose leaves are commitments to

$$S_0, S_1, \dots, S_{\mathit{lbase}} \tag{2}$$

and the span commitment *span* is the root of a Merkle tree whose leaves are commitments to

$$S_{\mathit{lbase}+1}, \dots, S_{\mathit{lbase}+\mathit{lspan}}, \tag{3}$$

where

- the Merkle tree rooted at *span* is a *perfect* binary tree (which is possible because *lspan* is always a power of two), and
- the shape of the Merkle tree rooted at *base* is determined by the rules governing parent/child relationships, given below.

Intuition. Intuitively, a party who makes a move that leads to the creation of a regular node is implicitly claiming that this node is correctly constructed. The nodes created in response to moves made by the honest party will always be correctly constructed. However, the adversary may make moves that result in the creation of incorrectly constructed nodes. The smart contract on L1 cannot distinguish between correctly and incorrrectly constructed nodes (however, the honest party, or any entity with access to S_0, certainly can).

4.2.2 Root nodes

Since $\mathcal{G}$ is directed-acyclic, it will have some number of *roots*, i.e., nodes with in-degree zero. Recall that H_0 is the commitment to S_0. A ***root*** in $\mathcal{G}$ is a regular node of the form

$$r = (\texttt{regular}, 0, n, H_0, \mathit{span}). \tag{4}$$

Correct construction. By definition, r is correctly constructed if the span commitment *span* is a commitment to $S_1, \ldots, S_n$.

Intuition. The party that makes a move that creates a root is claiming that *span* is a commitment to $S_1, \ldots, S_n$.

4.2.3 Nonterminal nodes

We call a regular node in $\mathcal{G}$ of the form

$$v = (\texttt{regular},\ \mathit{lbase},\ \mathit{lspan},\ \mathit{base},\ \mathit{span}), \tag{5}$$

where $\mathit{lspan} > 1$, a ***nonterminal node***. If this node has any children, it will have exactly two children. These children are of the form

$$v_{\mathrm{L}} = (\texttt{regular},\ \mathit{lbase},\ \mathit{lspan}/2,\ \mathit{base},\ \mathit{span}_{\mathrm{L}}) \tag{6}$$

and

$$v_{\mathrm{R}} = (\texttt{regular},\ \mathit{lbase} + \mathit{lspan}/2,\ \mathit{lspan}/2,\ H(\mathit{base}, \mathit{span}_{\mathrm{L}}),\ \mathit{span}_{\mathrm{R}}), \tag{7}$$

for some $\mathit{span}_{\mathrm{L}}, \mathit{span}_{\mathrm{R}}$ with

$$\mathit{span} = H(\mathit{span}_{\mathrm{L}}, \mathit{span}_{\mathrm{R}}). \tag{8}$$

Here, H is the hash function used to form the internal nodes of the Merkle trees. We call v_{L} the ***left child of*** v and v_{R} ***right child of*** v.

Correct construction. Recall that v is correctly constructed if its base commitment *base* is a commitment to (2) and its span commitment *span* is a commitment to (3). One sees that v_{L} is correctly constructed if its base commitment is also a commitment to (2) and its span commitment is a commitment to

$$S_{lbase+1}, \ldots, S_{lbase+lspan/2}. \tag{9}$$

Similarly, v_{R} is correctly constructed if its base commitment is a commitment to

$$S_0, \ldots, S_{lbase+lspan/2} \tag{10}$$

and its span commitment is a commitment to

$$S_{lbase+lspan/2+1}, \ldots, S_{lbase+lspan}. \tag{11}$$

This definition also tells us the precise shape of the Merkle tree for the base commitment of a correctly constructed node.

It is easy to see that if v_{L} and v_{R} are correctly constructed, then so is v. Conversely, assuming that H is collision resistant, if v is correctly constructed, then so are v_{L} and v_{R}.

Intuition. The first implication (v_{L} and v_{R} correctly constructed implies v correctly constructed) says the following: to prove the claim corresponding to the parent, it suffices to prove the claims corresponding to both children. The second implication (v correctly constructed implies v_{L} and v_{R} correctly constructed) says the following: to disprove the claim corresponding to the parent, it suffices to disprove the claim corresponding to one of the children.

4.2.4 Terminal nodes and proof nodes

We call a node of the form

$$v = (\texttt{regular},\ lbase,\ 1,\ base,\ span) \tag{12}$$

a ***terminal node***.

If v has any children, it must have exactly one child, and that child must be

$$v_{\mathrm{P}} = (\texttt{proof},\ lbase,\ 1,\ base,\ span). \tag{13}$$

Correct construction. Clearly, if v is correctly constructed, then so is v_{P}.

Intuition. A terminal node v corresponds to a claim that *base* is a commitment to (2) and that *span* is a commitment to $S_{lbase+1}$. Assuming that the claim regarding *base* is true, the presence of the child v_{P} in the graph indicates that the one-step state transition from S_{lbase} to $S_{lbase+1}$ has been proven to be correct, that is, $F(S_{lbase}) = S_{lbase+1}$, which means the claim corresponding to v is also true.

4.2.5 Position, context, and rivals

For a given regular node ($\texttt{regular}$, *lbase*, *lspan*, *base*, *span*), we define its ***position*** to be (*lbase*, *lspan*), and we define its ***context*** to be (*lbase*, *lspan*, *base*). We say two distinct regular nodes are ***rivals*** if their contexts are equal. A node that has no rivals is called ***unrivaled***. Note that, by definition, proof nodes are ***unrivaled***.

Intuition. A rivalry between two nodes corresponds to a particular type of dispute between the corresponding claims. If two nodes v and v' are rivals, then their corresponding claims agree with respect to the commitment *base* to the initial sequence (2), but disagree with respect to the commitment *span* to the following sequence (3). If v and v' are nonterminal nodes with children, and v has children v_{L} and v_{R}, and v' has children v'_{L} and v'_{R}, then either v_{L} and v'_{L} are rivals or v_{R} and v'_{R} are rivals, but not both (assuming collision resistance) – see full version of the paper for further details [1]. Thus, the dispute between the claims corresponding to v and v' can be resolved by resolving the dispute between the claims corresponding to either their left children or their right children.

4.2.6 Some general observations

A given node v in the protocol graph $\mathcal{G}$ may have several parents. However, the distance between v and any root is the same, which we call the ***depth*** of v.

We also observe that any two nodes that are children of nonterminal nodes and that have the same position are either both left children or both right children of their respective parents. More generally, the position of any regular node implicitly encodes the complete sequence of left/right steps along any path from the root to that node.

4.3 Types of Protocol Moves

There are three types of protocol moves. Each such move will supply some data, and when the L1 protocol processes this data, it will add zero, one, or two nodes to the protocol graph $\mathcal{G}$. Whenever a new node or edge is added to $\mathcal{G}$, the L1 protocol also records the round number in which it was added.

4.3.1 Root creation

The first type of move in the protocol is ***root creation***. Such a move supplies a commitment *span*. The L1 protocol adds to $\mathcal{G}$ the root node r as in (4), unless this node already exists in $\mathcal{G}$. We say this move ***creates the root*** r.

4.3.2 Bisection

The second type of protocol move is ***bisection***. Such a move supplies a nonterminal node v as in (5) in Section 4.2.3, together with commitments $\mathit{span}_{\mathrm{L}}$ and $\mathit{span}_{\mathrm{R}}$. The L1 protocol checks that

(a) v is already in $\mathcal{G}$ and rivaled, **(b)** v has no children, and **(c)** (8) holds,

and if so, adds to $\mathcal{G}$

- the node v_{L} as in (6), unless it is already in $\mathcal{G}$
- the node v_{R} as in (7), unless it is already in $\mathcal{G}$, and
- the edges $v \to v_{\mathrm{L}}$ and $v \to v_{\mathrm{R}}$.

We say this move ***bisects the node*** v. Note that the precondition that v has no children means that v has not been previously bisected. Also note that, in principle, a node may be bisected in the same round in which it was added to $\mathcal{G}$, so long as the preconditions hold at the moment the bisection move is executed (as we will see, although the adversary is free to do this, the honest party will not).

4.3.3 One-step proof

The third and final type of protocol move is ***one-step proof***. Such a move supplies a terminal node v as in (12) and a proof π. The L1 protocol checks that **(a)** v is already in $\mathcal{G}$ and rivaled, **(b)** v has no children, and **(c)** π is a valid proof (see details below), and if so, adds to $\mathcal{G}$ the node v_{P} as in (13) and the edge $v \to v_{\mathrm{P}}$.

We say this move ***proves the node*** v. Note that, in principle, a node may be proved in the same round in which it was added to $\mathcal{G}$ (as we will see, although the adversary is free to do this, the honest party will not).

Some details on the proof system. We assume a proof system that is comprised of a commitment scheme *Com*, a proof generator *Prove*, and a proof verifier *Verify*. The proof generator should take as input a state S and output a proof p. The proof verifier takes as input (h, h', p) and outputs `accept` or `reject`, and should always output `accept` on inputs of the form (h, h', p) where $h = \mathit{Com}(S)$, $h' = \mathit{Com}(F(S))$, and $p = \mathit{Prove}(S)$. We may state the required ***soundness property*** for the proof system as follows:

> *It should be infeasible for an adversary to construct a state S along with a triple $(h, \hat{h}', \hat{p})$, such that $h = \mathit{Com}(S)$, $\hat{h}' \neq \mathit{Com}(F(S))$, and $\mathit{Verify}(h, \hat{h}', \hat{p}) =$ `accept`.*

Note that the proof π supplied in a proof move must actually include a proof p as above, as well the commitment $h = \mathit{Com}(S_{\mathit{lbase}})$ and a right-most Merkle path *mp* for this commitment relative to the root *base*. To check the proof π, the L1 protocol validates *mp* (relative to *base* and h) and verifies that $\mathit{Verify}(h, \mathit{span}, p) =$ `accept` – note that for a correctly constructed node, we will have $\mathit{span} = \mathit{Com}(S_{\mathit{lbase}+1})$.

4.4 Timers

We are not quite done describing our dispute resolution protocol. However, before going further, some intuition is in order. The ultimate goal of the protocol is to allow both the honest party and the adversary to create root nodes and to make other moves in such a way that the L1 protocol can determine which root node is correctly constructed. Now, one trivial way to do this would be to have the honest party bisect the correctly constructed root, bisect its children, bisect all of their children, and so on, creating n terminal nodes, and then proving each of these terminal nodes. However, this trivial approach is extremely expensive. Instead, we adopt the following approach. Whenever a node is created and remains unrivaled for a period of time, the L1 protocol will take that as evidence that the claim corresponding to that node cannot (or need not) be proven false – the more time that elapses, the stronger the evidence. The L1 protocol has to then analyze all of this evidence and declare a "winner", that is, the root nodes that is most likely the correctly constructed one. The honest party will then adopt a "lazy" strategy, and only defend claims (i.e., bisect nodes) that are disputed (i.e., rivaled), and indeed the protocol is designed so that the honest party must defend all disputed honest claims in order to guarantee victory. However, if its claim corresponding to the correctly constructed root remains undisputed for a sufficiently long period of time, no further moves need to be made.

> *Throughout the remainder of Section 4.4, we consider a fixed run of the protocol for some number, say N, of rounds and let $\mathcal{G}$ be the resulting protocol graph.*

4.4.1 Creation and rival time

Recall that the L1 protocol keeps track of the round in which a given node v was created, that is, added to the graph $\mathcal{G}$. Let us call this the ***creation time of*** v, denoted $\mathsf{ct}(v)$, defining $\mathsf{ct}(v) \coloneqq \infty$ if v was never created throughout the protocol execution.

Let us call the round in which a regular node v becomes rivaled its ***rival time***, denoted $\mathsf{rt}(v)$. More precisely, $\mathsf{rt}(v)$ is defined to be the first round in which both v and a rival of v appear in $\mathcal{G}$, defining $\mathsf{rt}(v) \coloneqq \infty$ if v was never created or was created but never rivaled throughout the protocol execution. Clearly, $\mathsf{rt}(v) \geq \mathsf{ct}(v)$.

4.4.2 Local timers

For any regular node v and round number $t = 1, \ldots, N$, we define the ***local timer of*** v ***as of round*** t, denoted $\lambda_v(t)$, to be the number of rounds in which v, as of round t, has remained unrivaled since its creation. Formally, we define

$$\lambda_v(t) \coloneqq \max\left(\ \min(t, \mathsf{rt}(v)) - \mathsf{ct}(v),\ \ 0\ \right),$$

where the usual rules governing infinity arithmetic are used.

The following is an equivalent and perhaps more intuitive characterization of $\lambda_v(t)$:

- if v was created in round t or later, then $\lambda_v(t) = 0$;
- otherwise, if v was unrivaled as of round $t-1$, then $\lambda_v(t) = 1 + \lambda_v(t-1)$ and we may say "v's local timer ticks in round t";
- otherwise, $\lambda_v(t) = \lambda_v(t-1)$ and we may say "v's local timer does not tick in round t".

We also define the local timer for a proof node v as follows:

$$\lambda_v(t) \coloneqq \begin{cases} 0 & \text{if } \mathsf{ct}(v) > t, \\ \infty & \text{otherwise.} \end{cases}$$

Note that throughout the paper, whenever we say something happens "as of round t", we mean "after executing all moves in round t".

4.4.3 Bottom-up timers

Recall that the L1 protocol keeps track of the round in which any given node is added to the graph $\mathcal{G}$. For any node v and round number $t = 1, \ldots, N$, let us define $\mathsf{Child}_v(t)$ as the set of children of v as of round t. We define the ***bottom-up timer of*** v ***as of round*** t, denoted $\beta_v(t)$, recursively as follows:

$$\beta_v(t) \coloneqq \lambda_v(t) + \begin{cases} \min\left(\{\beta_w(t) : w \in \mathsf{Child}_v(t)\}\right), & \text{if } \mathsf{Child}_v(t) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$

In other words:

- if v was created later than round t, then $\beta_v(t) = 0$;
- otherwise, if v has no children as of round t, then $\beta_v(t) = \lambda_v(t)$;
- otherwise, $\beta_v(t)$ is the sum of $\lambda_v(t)$ and $\min_w \beta_w(t)$, where the minimum is taken over all children w of v as of round t.

4.4.4 Winners

We are finally in a position to define the condition under which the L1 protocol declares a winner. To this end, we introduce a parameter T, which we call the ***confirmation threshold***. We say a root node r in the protocol graph is ***confirmed in round*** t if $\beta_r(t) \geq T$. Suppose that t^* is the first round in which any root node is confirmed. If there is a unique root node r^* confirmed in round t^* then r^* ***is declared the winner***; otherwise, ***"none" is declared the winner***.

4.4.5 Paths in the protocol graph

To better understand the role of bottom-up timers in the protocol, and the rule for confirming root nodes, it is helpful to introduce some additional notions (which will also be useful in the analysis of the protocol).

A ***path*** P is a sequence of nodes $(v_0, \ldots, v_{q-1})$ in $\mathcal{G}$ such that $\mathcal{G}$ includes the edges $v_0 \to v_1 \to \cdots \to v_{q-1}$. We define the ***length of*** P to be q. We define the ***weight of*** P to be $\omega_P := \sum_{i=0}^{q-1} \lambda_{v_i}(N)$. We say P is a ***complete path*** if it is nonempty and and v_{q-1} has no children in $\mathcal{G}$.

Note that in defining paths, path weights, and complete paths, we are looking at the state of affairs as of round N, the last round of execution that led to the creation of the protocol graph $\mathcal{G}$. In particular, path weights are defined in terms of local timers as of round N.

We can now characterize bottom-up timers as of round N in terms of path weights. Specifically, for any node v in $\mathcal{G}$, we have

$$\beta_v(N) = \min_P \omega_P, \tag{14}$$

where the minimum is taken of all complete paths P starting at v. It follows that for any given nonnegative integer W, we have:

$\beta_v(N) \geq W$ *if and only if every complete path starting at v has weight at least W.* (15)

4.5 The honest strategy

So far, we have described the logic of the L1 smart contract that acts as a "referee" to ensure that all moves are legal and to declare a winner. However, we have yet to describe the (offchain) logic of the honest party. We do that here.

4.5.1 The honest party's initial move

We assume that the honest party has the states $S_0, S_1, \ldots, S_n$ and begins by computing the Merkle tree whose leaves are the commitments to $S_1, \ldots, S_n$. Let *span* be the root of this Merkle tree. The honest party submits a root creation move in round 1 using this value *span*. This is the only move that the honest party submits in round 1. This move, when executed, will add the honest root $r^\dagger$ (which we defined in Section 4.4.5 as the correctly constructed root node) to the protocol graph.

4.5.2 The honest party's subsequent moves

Now suppose the protocol has executed rounds $1, \ldots, t$ and no winner has been declared as of round t. Consider the protocol graph $\mathcal{G}$ as of round t (which the honest party can compute for itself). Recall the confirmation threshold parameter T introduced in Section 4.4.4 and the notions of paths and path weights introduced in Section 4.4.5. The honest party submits moves in round $t + 1$ as follows:

For each complete path P in $\mathcal{G}$ which starts at $r^\dagger$ and has weight less than T:

- *if P ends in a node v that is rivaled, then*
  - *if v is a nonterminal node, the honest party will submit a move to bisect v (if it has not already done so),*
  - *otherwise, v must be a terminal node, and the honest party will submit a move to prove v (again, if it has not already done so).*

4.6 Execution Example

In this section we go over an example execution of single-level BoLD (see Fig. 1). Here, we have $n = 4$. We assume in this example that the adversary's base and span commitments are valid commitments to sequences of states, but these sequences of states may be incorrect. For brevity, we write a node as "$[\cdots][\cdots]$", where first set of brackets encloses the sequence of states committed to in the base commitment, while the second set of brackets encloses the sequence of states committed to in the span commitment.

Initially, the honest party moves to create the honest root. As indicated in part (a) of the figure, this move gets executed in round 5 of the protocol execution. We will annotate each node with a superscipt indicating the current value of its local timer, adding a "+" to that value if its timer is still ticking (meaning the node is unrivaled).

We see in part (b) that the adversary executes moves in round 12 to create a rival root node *and* to bisect that node as well. We indicate the rivalry relation between nodes using dashed lines. While the honest root's local timer accumulated 7 rounds, it is now stopped because it is rivaled. Seeing that the honest root is rivaled, the honest party submits a move to bisect it. (The reader should note that in each part, new elements in the protocol graph, both nodes and edges, are highlighted in red.)

We see in part (c) that the honest party's bisection is executed in round 15. Note that the left child of the honest root is actually identical to a node created by the adversary in round 12. This illustrates how nodes can come to have multiple parents. While the left child of the honest root is unrivaled, its right child is rivaled. So the honest party submits a move to bisect that right child. (The reader should note that the edges of the honest tree are highlighted with thicker arrows.)

We see in part (d) that in round 20, the honest party's submitted bisection move has not yet been executed. Instead, the adversary is able to execute moves to create another rival root *and* to bisect that node as well. This bisection creates a node that rivals the left child of the honest root. Seeing that this node is rivaled, the honest party submits a move to bisect it. So now there are two bisection moves submitted by the honest party that are "in flight".

We see in part (e) that in round 24 both these bisection moves are executed.

We see in part (f) that in round 29, the adversary bisects a node that creates a rival of one of the honest node's created in round 24. That node, denoted "$[S_0S_1S_2][S_3]$" in the figure, is a terminal node. As such, the honest party submits a move to prove that node.

We see in part (g) that in round 35, the proof move submitted by the honest party is executed. This created a proof node, which we also write as "$[S_0S_1S_2][S_3]$", but one sees that its local timer is ∞.

At this point, if no other moves are made by the adversary, the timers on all of the leaves in the honest tree that are regular nodes will continue to tick. At the same time, for both adversarial roots, there is at least one path along which all timers are stopped (while the the local timer on the node denoted "$[S_0S_1^*S_2^*][S_3^*S_4^*]$" will continue ticking and remain the largest local timer in the graph, it will not help the adversary confirm an adversarial root). Thus, so long as the confirmation threshold is high enough, the honest root will eventually be confirmed while the adversarial roots will not be.

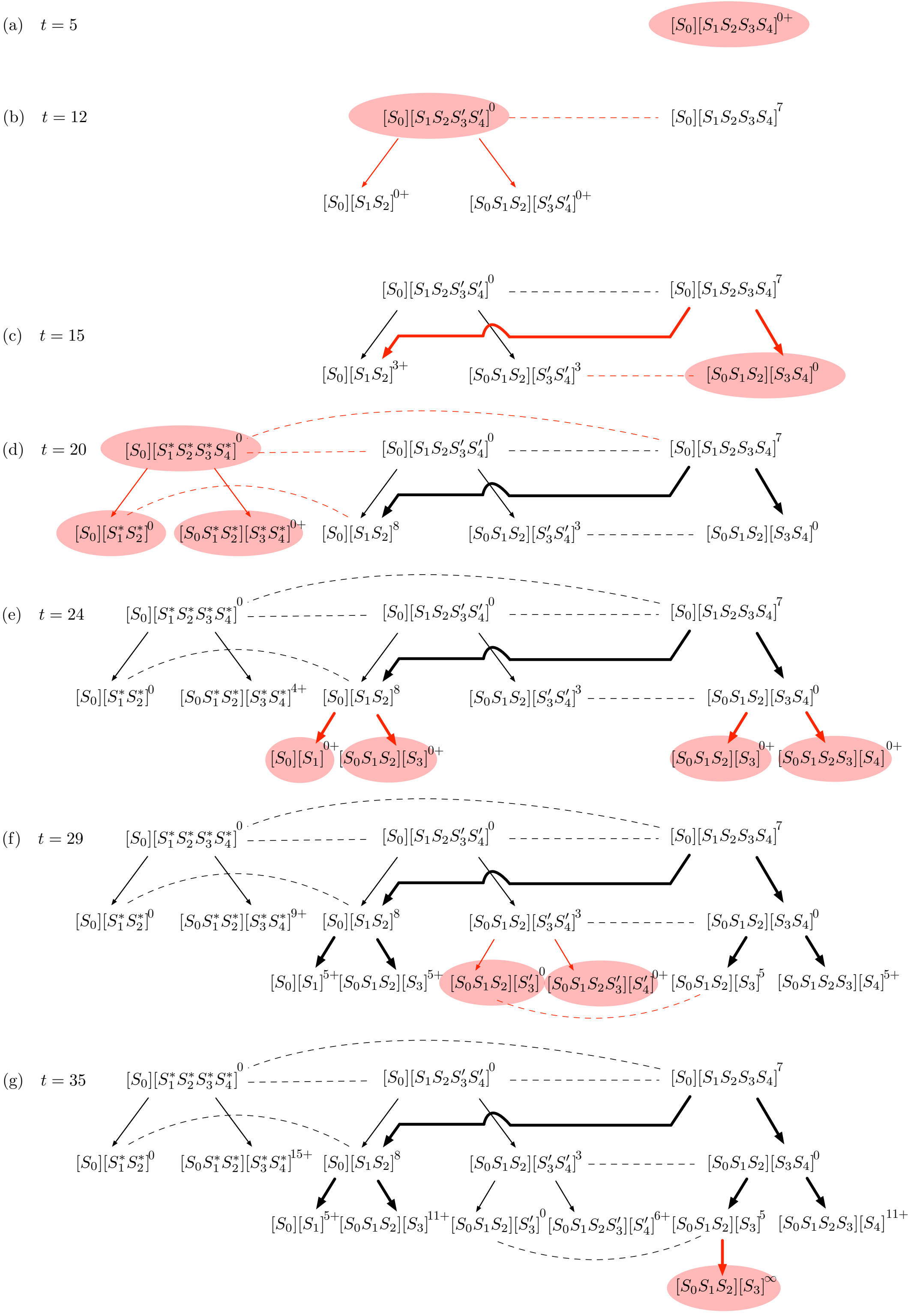
(a) $t = 5$
$[S_0][S_1S_2S_3S_4]^{0+}$
(b) $t = 12$
$[S_0][S_1S_2S'_3S'_4]^{0}$
$[S_0][S_1S_2S_3S_4]^{7}$
$[S_0][S_1S_2]^{0+}$
$[S_0S_1S_2][S'_3S'_4]^{0+}$
(c) $t = 15$
$[S_0][S_1S_2S'_3S'_4]^{0}$
$[S_0][S_1S_2S_3S_4]^{7}$
$[S_0][S_1S_2]^{3+}$
$[S_0S_1S_2][S'_3S'_4]^{3}$
$[S_0S_1S_2][S_3S_4]^{0}$
(d) $t = 20$
$[S_0][S^*_1S^*_2S^*_3S^*_4]^{0}$
$[S_0][S_1S_2S'_3S'_4]^{0}$
$[S_0][S_1S_2S_3S_4]^{7}$
$[S_0][S^*_1S^*_2]^{0}$
$[S_0S^*_1S^*_2][S^*_3S^*_4]^{0+}$
$[S_0][S_1S_2]^{8}$
$[S_0S_1S_2][S'_3S'_4]^{3}$
$[S_0S_1S_2][S_3S_4]^{0}$
(e) $t = 24$
$[S_0][S^*_1S^*_2S^*_3S^*_4]^{0}$
$[S_0][S_1S_2S'_3S'_4]^{0}$
$[S_0][S_1S_2S_3S_4]^{7}$
$[S_0][S^*_1S^*_2]^{0}$
$[S_0S^*_1S^*_2][S^*_3S^*_4]^{4+}$
$[S_0][S_1S_2]^{8}$
$[S_0S_1S_2][S'_3S'_4]^{3}$
$[S_0S_1S_2][S_3S_4]^{0}$
$[S_0][S_1]^{0+}$
$[S_0S_1S_2][S_3]^{0+}$
$[S_0S_1S_2][S_3]^{0+}$
$[S_0S_1S_2S_3][S_4]^{0+}$
(f) $t = 29$
$[S_0][S^*_1S^*_2S^*_3S^*_4]^{0}$
$[S_0][S_1S_2S'_3S'_4]^{0}$
$[S_0][S_1S_2S_3S_4]^{7}$
$[S_0][S^*_1S^*_2]^{0}$
$[S_0S^*_1S^*_2][S^*_3S^*_4]^{9+}$
$[S_0][S_1S_2]^{8}$
$[S_0S_1S_2][S'_3S'_4]^{3}$
$[S_0S_1S_2][S_3S_4]^{0}$
$[S_0][S_1]^{5+}$
$[S_0S_1S_2][S_3]^{5+}$
$[S_0S_1S_2][S'_3]^{0}$
$[S_0S_1S_2S'_3][S'_4]^{0+}$
$[S_0S_1S_2][S_3]^{5}$
$[S_0S_1S_2S_3][S_4]^{5+}$
(g) $t = 35$
$[S_0][S^*_1S^*_2S^*_3S^*_4]^{0}$
$[S_0][S_1S_2S'_3S'_4]^{0}$
$[S_0][S_1S_2S_3S_4]^{7}$
$[S_0][S^*_1S^*_2]^{0}$
$[S_0S^*_1S^*_2][S^*_3S^*_4]^{15+}$
$[S_0][S_1S_2]^{8}$
$[S_0S_1S_2][S'_3S'_4]^{3}$
$[S_0S_1S_2][S_3S_4]^{0}$
$[S_0][S_1]^{5+}$
$[S_0S_1S_2][S_3]^{11+}$
$[S_0S_1S_2][S'_3]^{0}$
$[S_0S_1S_2S'_3][S'_4]^{6+}$
$[S_0S_1S_2][S_3]^{5}$
$[S_0S_1S_2S_3][S_4]^{11+}$
$[S_0S_1S_2][S_3]^{\infty}$

Figure 1 An example execution.

4.7 Main result

We first recall the various parameters introduced so far: the nominal delay bound δ (see Section 3), the censorship budget $C_{\max}$ (see Section 3), the length of a computation $n = 2^{k_{\max}}$ (see Section 4.1), and the confirmation threshold T (see Section 4.4.4).

Given the above, we define

$$N^* := T + C_{\max} + (\delta + 1)(k_{\max} + 2), \tag{16}$$

The main result of this section is:

▶ **Theorem 1.** *Assume the hash function H used to build Merkle trees is collision resistant. Consider an execution of the protocol using the honest strategy and an arbitrary (efficient) adversary. Assume that $C_{\max} + (\delta + 1)(k_{\max} + 2) < T$. Then (with overwhelming probability) the honest root will be declared the winner at or before round N^*.*

The full proof of Theorem 1 can be found in the full version of the paper [1]. Below we provide an informal proof overview.

Proof Overview. Recall that the protocol declares some root as a winner only when the weight of every complete path starting at that root (equal to the sum of local timers of the nodes along the path) reaches the confirmation threshold. Thus, to prove Theorem 1 it suffices to prove two main claims: the first asserts that by round N^* the weight of every complete path starting at the honest root will surpass the threshold. The second asserts that by that round every adversarial root will have some complete path starting from it with weight strictly less than the threshold.

The honest strategy is designed to achieve exactly this – all its moves aim to extend complete paths starting at the honest root to allow them to accumulate more weight. These honest moves serve the second goal as well, since the nodes created by them rival nodes on paths starting at adversarial roots – this in turn prevents these paths from accumulating weight. The honest party has advantage in this game in the long run, since it is always able to provide one-step proofs to regular terminal nodes that are descendants of the honest root.

A bit more formally, in order to prove the first claim we observe that at any round of the protocol, if a complete path starting at the honest root does not accumulate more weight, it must be the case that all its nodes are rivaled. But in this case, the honest party will submit a (legal) move to extend the path by either bisecting or proving the last node in the path. Of course, once that move is executed the adversary can immediately submit a counter-move to rival the new node in the path, preventing the local timer of the new node from ticking by doing so. At this point the honest party will submit another move to extend the path, followed by another adversary counter-move and so on. This process must end at some point since the maximum depth of a path is bounded by $k_{\max} + 1$, and once a proof node has been added to a path, its weight becomes ∞ by definition. The number of rounds N^* is chosen to account for all steps in this process and also to account for possible rounds in which the honest party's moves were delayed or censored.

To see why the second claim holds, consider any adversarial root r at round N^*. A first observation is that r rivals the honest root $r^\dagger$ (which must exist in the protocol graph by this round). Another observation is that for any two nonterminal rival nodes, both with left and right children nodes, either both left children are rivals or both right children are rivals. An iterative application of this observation shows that there must exist two complete paths P and $P^\dagger$ starting at r and $r^\dagger$ (respectively), such that every node in the shorter path of the two has a matching rival node in some prefix of the longer path. We then make the crucial observation that at any given round, only one node in the union of both paths can have its local timer tick. It follows that the sum of weights of both paths is bounded by the number of elapsed rounds since the beginning of the protocol. However, the first claim above gives a lower bound on the weight of $P^\dagger$. The combination of these gives us the desired upper bound on the weight of P, which proves the second claim.

4.8 A Practical Implementation

As a practical matter, our protocol as given requires too much from the L1 protocol. Specifically, we are asking the L1 protocol to continuously track all of the local and bottom-up timers round by round, which would be prohibitively expensive. A simple change to the protocol and the honest party's strategy can fix this. The idea is to use "lazy evaluation" that computes bottom-up timer estimates "on demand". These estimates will always be no more than the true value of the timer. A detailed description and analysis of the modified protocol is deferred to the full version of the paper [1].

5 Multi-level BoLD

Single-level BoLD has a number of desirable properties, but for some use cases the offchain compute cost of the honest party may be excessive, because of the amount of hashing required by the honest strategy. For example, if $n = 2^{55}$, which is plausible for a dispute in Arbitrum, the adversary will need to compute 2^{55} state commitments, each of them a Merkle hash of a virtual machine state, and then do about 2^{55} additional hashes to build the Merkle tree. This much hashing may be too time-consuming in practice.

An alternative is to use BoLD recursively. For example, we might execute BoLD using the iterated state transition function $F' = F^{2^{25}}$, thereby narrowing the disagreement with the adversary to one iteration of F' which is equivalent to 2^{25} iterations of F. A recursive invocation of BoLD over those iterations of F would then narrow the disagreement down to a single invocation of F which could then be proven using the underlying proof system.

The (realized) hope is that if $n = 2^{55}$ and there are N_{A} adversarial roots, then the honest party will have to do one iteration of BoLD with a "sequence length" of $n_2 = 2^{30}$ and a "stride" of $\Delta_2 = 2^{25}$, then at most N_{A} "sub-challenge" iterations of BoLD, each with a "sequence length" of $n_1 = 2^{25}$ and a "stride" of $\Delta_1 = 1$. For realistic values of N_{A}, this requires much less hashing than single-level BoLD.

The ***multi-level BoLD*** protocol generalizes this idea of applying single-level BoLD recursively, to support more than two levels. To do this, we extend the single-level protocol by introducing two new elements: refinement nodes and refinement moves. A refinement move takes a node representing a claim about a single step of the iterated transition function at a particular level – this would be a deepest node in that level, which we also call a *terminal node* – and creates a child node representing the same computation but according to the refined transition function, in a new, deeper level of the protocol. This new child node is called a refinement node, and is analogous to a root of this next level.

Refinement moves are different than the bisection and proof moves in the sense that the protocol cannot be directly convinced that the newly created refinement node is correctly constructed (assuming the correct construction of its parent) – in the same way that the single-level protocol cannot be directly convinced that some root is correctly constructed. To deal with this, the protocol does not limit the amount of refinement nodes created from a single parent terminal node, and these nodes compete with each other in a recursive execution of the protocol. Essentially, the goal of this recursive competition is to convince the protocol which of the refinement nodes is the correctly constructed one.

The transition to multiple levels requires a significant change to the timer scheme. Specifically, the bottom up timer of a node is defined exactly as in the single-level version – except for terminal nodes at any level, for which it is defined as the local timer of that terminal node plus the *maximum* of the bottom-up timers of all of its children refinement nodes. The reason for using maximum instead of minimum here is exactly the same as the reason that the winner root is the one with the highest value bottom-up timer (and not the lowest) – this definition ensures that the unique correctly constructed refinement node, presumably the one with the highest bottom-up timer, will be the one to contribute to the conviction that its parent is correctly constructed as well.

We then prove, as we did in the single-level version, that as long as the confirmation threshold is set high enough, the honest root will be declared winner before any adversarial root. Further details of the extended protocol are deferred to the full version of the paper [1].

6 Gas, staking, and reimbursement in BoLD

In this section, we summarize our analysis of the gas costs incurred by the honest party when executing the BoLD protocol. We introduce a staking requirement for participation and give an overview of how staking can be used to reimburse the honest party for its efforts, and to mitigate against resource exhaustion attacks. For further details, see full version of the paper [1].

6.1 Single-Level BoLD

In single-level BoLD, we require parties creating a root node to place a *stake* (locking up funds on the parent chain) when doing so. When a root node is declared a winner, the stake for the winner is reimbursed and the stake for the other root nodes is confiscated. In the event of a challenge, some stakes will be confiscated; these can be used to reimburse honest parties for their gas costs incurred while running the protocol.[3]. The value of the stake – S – should be set high enough to ensure that confiscated stakes are sufficient to reimburse the gas costs incurred by honest parties running the protocol. However, it should probably be set much higher than this, to discourage an adversary from making any challenge at all, which delays confirmation of the honest root, and to mitigate against resource exhaustion attacks.

In order to reimburse honest parties for their gas costs, it suffices to set S at least as large as a certain fixed gas cost, G, which we can calculate (this is essentially the cost of bisecting down a single path from the honest root to a terminal node and then making one proof move).

To reason about resource exhaustion, we define the notion of a *resource ratio* or *griefing ratio* (ρ). This is (as the name suggests) a ratio, whose numerator is the total staking and gas cost paid by the adversary to mount an attack on BoLD, and whose denominator is the total gas and staking cost paid by the honest party in the course of responding to that

[3] If there is no challenge, BoLD does not have a built-in mechanism to refund the gas cost of the party who creates the confirmed root; compensation for this party will need to come from some external source.

attack. The denominator does not include the staking and gas cost to create the honest root, since the honest party must bear this cost regardless of what the adversary does to attack the protocol (indeed, the honest party must pay this cost even if there is no challenge).

A tradeoff exists between the size of stake and the griefing ratio obtained from using that size stake. A larger stake means that the capital cost for an honest party to create an honest root is higher, but also leads to a larger resource ratio, meaning it is proportionally more expensive for the adversary to impose additional costs on the honest party.

To achieve a target resource ratio ρ, we show that it suffices to set S to, roughly, $\rho \cdot G'$, for a fixed amount of gas G'.

For further details, see full version of the paper [1].

6.2 Multi-Level BoLD

In multi-level BoLD, we require stakes on refinement edges as well as root edges. The amount of stake required can differ between levels, but at any given level, root or refinement edges at that level all require the same amount of stake. That is, for each level ℓ, we have a parameter S_l, giving the required stake to create a root or refinement node at that level. The most effective staking schemes involve these stakes being larger at higher levels; that is, $S_1 \leq ... \leq S_L$.

As in single-level BoLD, in order to reimburse honest parties for their gas costs, it suffices to set each S_l at least as large as a certain fixed gas cost, G_l, which we can calculate.

In order to obtain a target resource ratio of ρ, it turns out that we need exponentially escalating stakes. That is, for some fixed amount of gas G', we set S_1 to G', and then for $1 \leq \ell < N$, $S_{\ell+1}$ will be set to (roughly) $\rho \cdot S_l$. This is why it is important to limit the number of levels in multi-level BoLD: stakes need to increase exponentially in the number of levels in order to maintain the same griefing ratio.

We have also explored an alternate staking regime, which (roughly speaking) gives an "asymptotically unbounded" resource ratio. Intuitively, in this regime, rather than being bounded by a constant, the ratio of the adversary's cost to the honest party's cost tends toward infinity as the honest party's cost grows. However, as the number of levels grows, the rate at which this ratio tends toward infinity becomes slower, and does so very quickly. This approach is therefore only useful for single-level BoLD and 2-level BoLD (i.e., multi-level BoLD with $L = 1$ or $L = 2$).

For further details, see full version of the paper [1].

References

1 Mario M. Alvarez, Henry Arneson, Ben Berger, Lee Bousfield, Chris Buckland, Yafah Edelman, Edward W. Felten, Daniel Goldman, Raul Jordan, Mahimna Kelkar, Akaki Mamageishvili, Harry Ng, Aman Sanghi, Victor Shoup, and Terence Tsao. Bold: Fast and cheap dispute resolution. *CoRR*, 2024. arXiv:2404.10491.

2 Harry A. Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. Arbitrum: Scalable, private smart contracts. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1353–1370. USENIX Association, 2018. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner.