Arbitrum Nitro: A Second-Generation Optimistic Rollup

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

Offchain Labs, Inc.

Abstract

We present Arbitrum Nitro, a second-generation Layer 2 blockchain protocol. Nitro provides higher throughput, faster finality, and more efficient dispute resolution than previous rollups. Arbitrum achieves these properties through several design principles: separating sequencing of transactions from deterministic execution; combining existing Ethereum emulation software with extensions to enable cross-chain functionalities; compiling separately for execution versus proving, so that execution is fast and proving is structured and machine-independent; and settling transaction results to the underlying Layer 1 chain using an optimistic rollup protocol based on interactive fraud proofs.

1 Introduction

In previous work, we described Arbitrum [9], a system and protocol for improving the performance and scalability of smart contracts. This paper describes Arbitrum Nitro, a significantly improved design that offers advantages over the original, including greater efficiency, reduced latency, stronger liveness guarantees, and better incentive compatibility.

1.1 Properties of Arbitrum Nitro

Nitro supports execution of smart contracts. The system is implemented as a "Layer 2" on top of Ethereum [14], although in principle it could be implemented on any blockchain system that supports at least basic smart contract functionality. Nitro provides an Ethereum-compatible chain: Nitro runs smart contract applications deployed in Ethereum Virtual Machine (EVM) code, and Nitro nodes support the same API as common Ethereum nodes.

The Nitro protocol guarantees both safety and progress for the Layer 2 chain, assuming that (1) the underlying Ethereum chain is safe and live, and (2) at least one participant in the Nitro protocol is behaving honestly. The protocol is termed "optimistic" because execution is more *efficient* when parties behave according to their incentives.

A variant of Nitro, called AnyTrust, provides lower cost in exchange for an additional trust assumption. The main body of this paper describes regular Nitro, and the differences in AnyTrust are described in Section 7.

Nitro's source code is available at `https://github.com/offchainlabs/nitro` and its submodules.

1.2 Design Approach

Nitro's design has four distinctive features, which we will use to organize the presentation.

- *Sequencing, then deterministic execution*: Nitro handles submitted transactions in two stages. First, it puts transactions into the sequence in which they will be processed, and publishes that sequence. Second, it applies a deterministic state transition function to each transaction, in sequence.
- *Geth at the core*: The core execution and state maintenance functions in Nitro are handled by code from the open source go-ethereum ("geth") package, which is the most popular Ethereum node software. By compiling in that geth code as a library, Nitro ensures that its execution and state are highly compatible with Ethereum's.
- *Separate execution from proving*: Nitro compiles the code of its state transition function for two targets. The code is compiled for native execution when used in ordinary operation in a Nitro node. It is compiled to portable web assembly ("wasm")[13] code for use in the fraud proof protocol if needed. This dual-target approach assures that execution is fast, while proving is based on structured, machine-independent code.

- *Optimistic rollup with interactive fraud proofs*: Building on the original Arbitrum [9] design, Nitro uses an improved optimistic rollup protocol based on an optimized dissection-based interactive fraud proof protocol.

1.3 Structure of the paper

The remainder of the paper is structured as follows. Section 2 introduces the sequencer and the deterministic state transition function. Section 3 describes the structure of the Nitro software, and the affordances it provides to support a Layer 2 chain. Section 4 describes the structure and derivation of the code used for proving execution results. section 5 presents the protocol used to assert execution results, and Section 6 describes the challenge sub-protocol, which resolves any disputes about those results. AnyTrust, an extension of Nitro using an external data availability committee, is described in Section 7. Section 8 concludes and suggests future directions.

2 Sequencing, Then Deterministic Execution

Processing of submitted transactions in Nitro occurs in two phases. First, a component called the *Sequencer* puts the transactions into an ordering. Second, the transactions are consumed, in sequence, by the deterministic *State Transition Function*. The process is illustrated in Figure 1.

Submitted transactions may or may not be valid. For example, they may lack a valid signature, or they may be garbage data. An honest Sequencer will make its best effort to discard submitted transactions that are invalid, but the protocol makes no assumptions about whether transactions in the Sequencer's output are valid. Executing the State Transition Function on an invalid transaction will simply discard that transaction.

2.1 The Sequencer

The Sequencer is trusted only to order incoming transactions honestly, according to a first-come, first-served policy.[1] At present the Sequencer is a centralized component operated by Offchain Labs, but in the future we intend to transition to a committee-based sequencer using a fair distributed sequencing protocol [11, 10].

The Sequencer publishes its transaction ordering in two ways. First, it publishes a real-time feed of the sequenced transactions, which any party can subscribe to. The feed represents the Sequencer's promise to record transactions finally in a particular order. The Sequencer has the power to keep its promises, so any deviation from the promised sequence would be due to malfunction or malice by the Sequencer.

Second, the Sequencer posts its transaction sequence as Ethereum calldata. The Sequencer collects a batch of consecutive transactions, compresses it using a general-purpose compression algorithm (currently brotli [1]), and passes the result to the Nitro chain's Inbox contract, which runs on L1 Ethereum. These batches represent the final and authoritative transaction ordering, so that once the Sequencer's transaction to the Inbox has finality on Ethereum, the Nitro chain's transaction sequence is final.

The Delayed Inbox Although most user transactions will be submitted directly to the Sequencer and included in one of the Sequencer's batches, there is another way to submit transactions, through the *Delayed Inbox*. This has two purposes. First, it allows transactions to be submitted by L1 Ethereum contracts, which cannot generate the digital signatures needed to submit a transaction through the Sequencer. Second, it provides a backup method for anyone to submit a transaction in case the Sequencer starts censoring valid transactions.

A transaction is added to the Delayed Inbox by calling a method on the Nitro chain's inbox contracts. The contracts keep a queue of timestamped transactions. The Sequencer can include in its sequence the first message in the delayed inbox queue. An honest Sequencer will do this after a brief delay, which is long enough to ensure that the arrival of that message in the delayed inbox will not be wiped out by a reorganization of the L1 chain– typically a 10-minute delay.

However, if a message has been in the delayed inbox for at least a threshold time period (currently 24 hours), anyone can force that message to be included next in the chain's inbox, thereby guaranteeing its execution. This forced inclusion step prevents censorship by the Sequencer, but would only be needed due to the Sequencer being malicious or having a long downtime.

Section 3.2.3 presents more details about the role of the Delayed Inbox.

2.2 Deterministic execution

After incoming transactions have been sequenced, there are processed by the Nitro chain, by using the chain's State Transition Function (STF). The STF takes as input a state (which is an Ethereum state tree [14]), and an incoming message, which is usually a single transaction. The STF's output is an updated state and a new

[1] In principle the Sequencer could implement any transaction ordering policy. The first-come, first-served policy is easy to implement and minimizes latency.

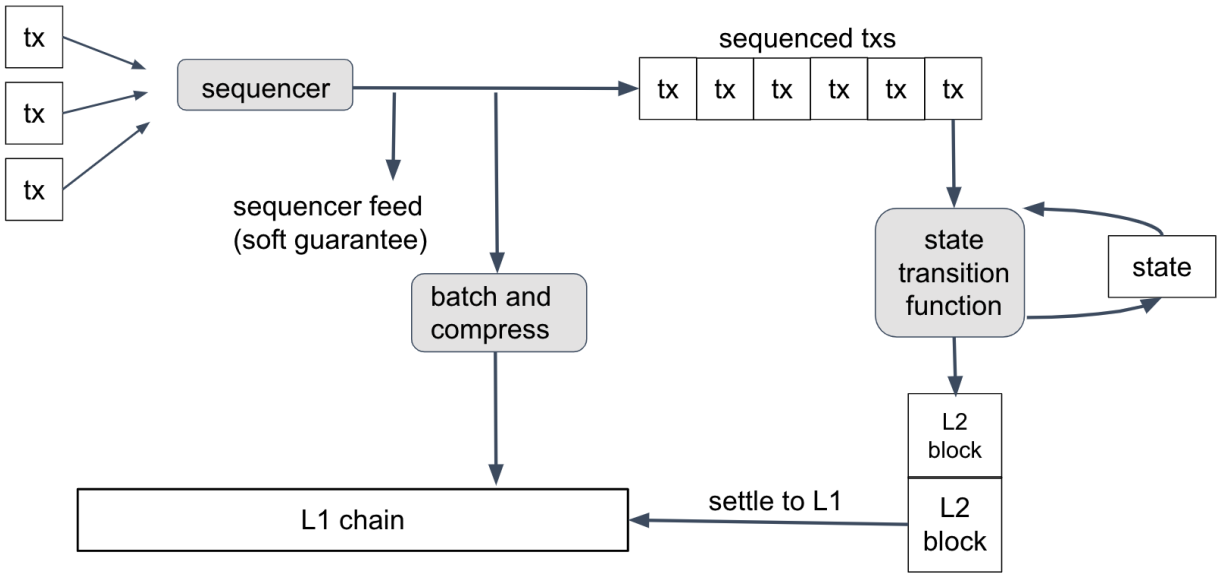

Figure 1: Processing of transactions in Nitro. The sequencer establishes an ordering on transactions, and publishes the order as a real-time feed and as compressed data batches on the L1 chain. Sequenced transactions are processed one at a time by a deterministic state transition function, which updates the chain state and produces L2 blocks. These blocks are later settled to the L1 chain.

Ethereum-compatible block header to be appended to the Nitro chain.

The STF is fully deterministic, so that the outcome of executing the STF on a transaction depends only on the transaction's data and the state before the transaction. Because of this, the outcome of a transaction T depends only on the genesis state of the Nitro chain, the sequence of transactions preceding T, and T itself.

Because of this determinism, an honest party can determine the full state and history of the chain, given only the transaction sequence. Nodes need not communicate, and no consensus is necessary among them, in order to agree on the correct state and history, because this depends only on the transaction sequence which is visible to all.

Nitro does have a rollup sub-protocol (discussed in Section [**?**]) to confirm the transaction results to the L1 Ethereum chain. That protocol does not *decide* the result of transactions but only *confirms* and *records* the result that was already known to honest protocol participants.

3 Software Architecture: Geth at the Core

The second key design idea in Nitro is "geth at the core." Here "geth" refers to `go-ethereum`, the most common node software for Ethereum. As its name would suggest, go-ethereum is written in Go [7], as is almost all of Nitro.

The software that makes up a Nitro node can be thought of as built in three main layers, which are shown in Figure 2.

- The base layer is the core of geth–the parts of geth that emulate the execution of EVM contracts and maintain the data structures that make up the Ethereum state. Nitro compiles in this code as a library, with a few minor modifications to add necessary hooks.

- The middle layer, which we call ArbOS, is custom software that provides additional functions associated with Layer 2 functionality, such as decompressing and parsing the Sequencer's data batches, accounting for Layer 1 gas costs and collecting fees to reimburse for them, and supporting cross-chain bridge functionalities such as deposits of Ether and tokens from L1 and withdrawals of the same back to L1.

- The top layer consists of node software, mostly drawn from geth. This handles connections and incoming RPC requests from clients and provides the other top-level functionality required to operate an Ethereu-compatible blockchain node.

Because the top and bottom layers rely heavily on code from geth, this structure has been dubbed a "geth sandwich."[2]

[2]Strictly speaking, geth plays the role of the bread in the sandwich, and ArbOS is the filling, but this sandwich is named for the bread.

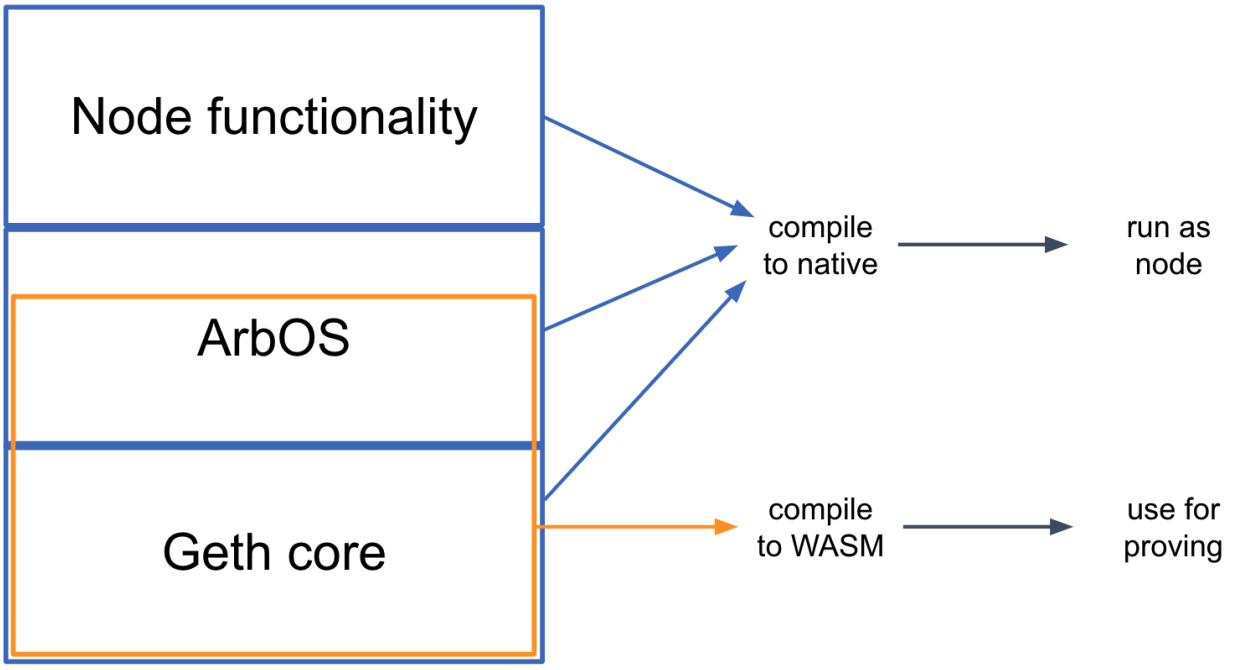

Figure 2: High-level structure of the Nitro code, showing major components. The boundary of the State Transition Function is shown in orange.

The State Transition Function consists of the bottom geth layer, and a portion of the middle ArbOS layer. In particular, the STF is a designated function in the source code, and implicitly includes all of the code called by that function. The STF takes as input the bytes of a transaction received in the inbox, and has access to a modifiable copy of the Ethereum state tree. Executing the STF may modify the state, and at the end will emit the header of a new block (in Ethereum's block header format) which will be appended to the Nitro chain.

3.1 ArbOS

ArbOS is a software layer that implements functionality that is necessary and convenient for managing a Layer 2 chain. This includes bookkeeping functions, cross-chain communication, and L2-specific fee tracking and collection. Portions of ArbOS are included in the State Transition Function.

3.1.1 State Representation

All state of a Layer 2 Nitro chain is stored in Ethereum's Merkle Patricia state trie data structure. This includes the state of ArbOS, which is modified as part of the State Transition Function.

ArbOS encodes its state in the storage slots of a special Ethereum account whose private key is unknown. The specific slots used are chosen to satisfy the following goals

- keep all ArbOS state in the storage of a single Ethereum account,
- allow sub-components of ArbOS to manage their state separately, without collisions,
- maintain reasonable locality within the same sub-component, and
- avoid constraining future additions to the state.

The state is organized as a hierarchy of nested "spaces" where each space is a mapping from 256-bit index to 256-bit value, with all values implicitly initialized to zero. This structure is mapped onto a single flat 256-bit to 256-bit key-value store, which is the storage of the special Ethereum contract.

Each space is associated with a key. The key of the root space is zero, and the key of a subspace named n within a space with key k is $H(k||n)$ where H is Ethereum's standard Keccak256 hashfunction. This scheme ensures that spaces' keys do not collide.

Within the space with key k, the item with index i is stored at location $H'(k||i)$ in the underlying flat storage, where H' is a locality-preserving hashfunction. The function $H'(x)$ hashes all but the last 8 bits of x, truncates the result to 248 bits, then appends the last 8 bits

of x. This ensures that contiguous groups of 256 indices are kept contiguous by the hashfunction, while ensuring that the function is collision-resistant.[3] The use of this locality-preserving hashfunction will reduce the cost of state accesses when Ethereum switches to a state representation that rewards contiguity.

3.2 Cross-chain Interaction

One of the roles of ArbOS is to support secure cross-chain calls in both directions between Nitro and Layer 1 Ethereum. An account on one layer can send a transaction to the other chain, and that transaction will be executed asynchronously. In this section we describe the Outbox, which supports calls from a Nitro chain to Ethereum, and two mechanisms, the Inbox and Retryable Tickets, which support calls from Ethereum to a Nitro chain.

3.2.1 Address Aliasing

When a Layer 1 Ethereum contract submits a transaction to a Nitro chain, the question arises of what sender address should be attached to the transaction when it runs on Nitro. It is tempting to simply use the L1 address of the sending contract, but there could be a contract at the same address on the Nitro chain, and if so the two contracts would be indistinguishable to a call recipient on Nitro, which would allow either one to impersonate the other on the Nitro chain. This is potentially dangerous.

To avoid this, the address of an L1 sender at address A on Layer 1 is presented on the Nitro chain as $f(A) = (A + C) \mod 2^{160}$, where C is a specified odd constant. Because all Ethereum addresses, and all other Nitro addresses, are generated by hashing some data (the exact data depending on how the address originated), it is infeasible to generate a collision between an aliased address and another Nitro address. Nitro software translates addresses in either direction as needed, when interacting with Ethereum.

3.2.2 The Outbox

Nitro's Outbox system allows for arbitrary L2 to L1 contract calls; i.e., messages initiated on L2 which are eventually executed on L1. Given the security properties inherent to optimistic rollups, an outgoing message's L1 execution can only take place after its message's dispute period passes and its Rollup Block is confirmed (as described in 5).

Logically, an L2 to L1 message is like a "ticket" that is created on L2 and can later be "redeemed" at L1 to cause a specified transaction call to occur at L1. The recipient of that transaction call can verify that it is an authorized L2 to L1 message call, and can confirm the L2 sender and data of the call. This functionality is sufficient to support secure transfers of ETH, tokens, or other forms of value from L2 to L1. The asynchronous ticket model is needed for safety. Messages must be asynchronous because they cannot be redeemed until an RBlock that includes them has been confirmed; and redemption is done per-ticket and not in strict order because redemption of a particular ticket could require executing arbitrary code which may be very expensive in L1 gas or not even possible.

L2 to L1 messages are initiated via an L2 transaction that calls a special *ArbSys* precompile that is part of ArbOS. ArbOS serializes the L2 sender's address, amount of ETH provided with the call, L1 destination address, and calldata, and the result becomes an L2 to L1 message.

Part of the state asserted by an RBlock is the root hash of a Merkle tree of all L2 to L1 messages in the chain's history. When an RBlock is confirmed, this root hash is updated in the chain's Outbox contract on L1; at this point, a user can call the Outbox contract with a Merkle proof of inclusion of a message to redeem it. The L1 Outbox contract tracks which messages have been successfully redeemed, so that each message can be redeemed at most once.

ArbOS uses an efficient representation to support incremental computation of the Merkle tree root while requiring only logarithmic storage. Any Merkle tree can be decomposed into a minimal set of completely full binary subtrees, of decreasing size. ArbOS remembers only the size of the overall Merkle tree, along with the root hashes of these full subtrees. The addition of a new leaf to the tree will result in a state with exactly one completely full subtree that did not previously exist. ArbOS emits an L2 EVM event containing the hash of the subtree. Every inclusion proof in a Merkle tree version consists of a set of these subtree hashes, and a client who wants to create a proof can use standard event searches to find the L2 events containing the hashes needed for their proof. (The Nitro node includes support for constructing these proofs automatically.)

3.2.3 The Inbox

The Inbox, which is managed by a set of Layer 1 Ethereum contracts, is responsible for recording messages (typically transactions) sent to a Nitro chain. The *Delayed Inbox* receives messages that are submitted on Layer 1, while the main Inbox receives messages sent by the Sequencer as well as merging in messages from the Delayed Inbox.

The Delayed Inbox is a set of Layer 1 Ethereum con-

[3] H' loses a few bits of collision resistance compared to H, requiring a factor of 2^4 less effort to find a collision by brute force search, but this reduction is acceptable.

tracts that accept messages to be delivered to the Nitro chain. This is an alternative to submitting via the Sequencer. The Delayed Inbox is the only way for Layer 1 contracts to submit messages, because Layer 1 contracts cannot sign messages or submit them to the Sequencer. It also provides a way for any user to submit a message without relying on the Sequencer, in case the Sequencer is unavailable or misbehaving.

The Delayed Inbox is logically a queue. It tracks the number of messages submitted to it and a hash-chain commitment to the contents of those messages. These messages will eventually be copied into the main Inbox as described below.

The Sequencer submits its data batches directly to the Inbox contracts. Each batch contains a compressed sequence of transactions, along with a directive to include a specified number of messages from the head of the Delayed Inbox. The main input loop of the ArbOS will consume these Sequencer batches in order.

A well-behaved Sequencer will include Delayed Inbox messages after a delay promptly, but after a long enough delay to minimize the risk that a reorganization of the Layer 1 chain will cause an included message to disappear or change. The current Sequencer implementation includes Delayed Inbox messages after ten minutes. If the Sequencer fails to include a Delayed Inbox message within a fixed interval[4], any party can call the Inbox to force inclusion of the message, which occurs by forcing a Sequencer batch including the message(s) into the Inbox. The ability to submit a message to the Delayed Inbox and force its inclusion without relying on the Sequencer supports Nitro's guarantee of trustless liveness.

3.2.4 Retryable Tickets

Layer 1 contracts can submit transactions to a Nitro chain, but those transactions necessarily run asynchronously on the Nitro chain, so the submitting Layer 1 transaction cannot see whether they succeed. This poses problems for applications such as token bridging which require a Layer 1 contract to ensure that a deposit transaction runs at Layer 2. If the deposit transaction fails, for example due to changes in gas prices, the Layer 1 bridge contract cannot know this until much later, and user funds could be lost or stranded in the meantime.

To support this and other use cases, Nitro includes a retryable ticket system, which allows a transaction submitted from Layer 1 to be designated as retryable, meaning that if the transaction fails, ArbOS creates a retryable ticket for the transaction. If the transaction had ETH callvalue attached, ArbOS escrows that callvalue, associated with the ticket. A later transaction can redeem the ticket by providing funds for its gas. The retry will run with the original sender, callvalue, and data, with the only difference being the gas parameters and who is paying for the gas.

If a retry fails, the ticket remains in the retry buffer, and can be retried again. (If the retry succeeds, the ticket is removed.) After a fixed interval, currently one week, an unredeemed ticket expires and will be automatically deleted by ArbOS. If the deleted ticket had callvalue escrowed by ArbOS, that callvalue is burned.

The submitter of a retryable transaction must pay a submission fee, which will be refunded to the submitter if the initial execution of the transaction succeeds, or paid to ArbOS if the initial execution fails and a retryable ticket is made. The submission fee is meant to cover the cost of keeping the ticket in ArbOS's storage until the ticket's expiration time. The submission fee depends on the size of the transaction and is determined, for each submission, by a Layer 1 contract, to ensure that Layer 1 submitters know exactly what the fee will be.

3.2.5 Token Bridge

Nitro's cross-chain messaging affordances can be used to create a Token Bridge, an application that allows for the effective transfer of assets between the Ethereum and Nitro chains. The Offchain Labs team has implemented and released a Token Bridge informally referred to as "canonical", though the Nitro core protocol grants it no special recognition or affordances; it is effectively an application like any other (note that, similarly, Nitro has no natively recognized notion of tokens nor of any particular token standard, much like Ethereum).

At its core, the Token Bridge offers the ability to deposit (transfer from Ethereum to Nitro) and withdraw (transfer from Nitro to Ethereum) fungible tokens. To deposit n tokens, a transaction is sent to Ethereum which carries out two operations: it sends n tokens to an L1 contract (known as a Token Gateway), and creates a retryable transaction (Section 3.2.4) that mints n tokens of an L2-counterpart contract. The two token contracts are counterparts, due to the guarantee that a holder of L2 token can carry out a withdrawal: a withdrawal of m tokens is initiated via an L2 transaction which burns m tokens on L2 and creates a L2 to L1 message (Section 3.2.2) directing that the L1 Token Gateway release m tokens on L1. Upon confirmation, the message can be executed in the Outbox, which releases the m tokens from escrow.

By default, tokens are bridged via the "Standard Gateway" contracts. When going through the Standard Gate-

[4]Setting this parameter reflects a tradeoff between the desire for prompt inclusion, and the desire to avoid unexpected behavior if the Sequencer experiences downtime. Currently it is set to 24 hours, but we expect the value to be reduced as the perceived risk of Sequencer downtime diminishes.

way, a token gets its L2 counterpart deployed on L2 at a deterministically generated address (via the CREATE2 EVM opcode). The token contract deployed on L2 is a StandardArbERC20 — an OpenZeppelin [12] ERC20 contract with additional affordances to mint/burn from the bridge contracts, along with callback hook affordances. Alternatively, to use a different contract as its L2 counterpart, an L1 token contract can register itself to any other "custom" gateway. Gateway Router contracts are responsible for tracking the mapping of L1 tokens to their Gateways (which in turn map them to their L2 counterpart tokens).

Many additional token bridge features are theoretically possible, including non-fungible token bridging, atomic swaps for fast L2 to L1 withdrawals, and bridging tokens natively deployed on L2 back to L1. Several independent services offer enhanced bridging functionalities, typically building on the canonical bridge.

3.3 Gas and Fees

Like many blockchains, Arbitrum collects fees from each transaction, to cover the costs of operating the chain, align incentives, and ration resources when demand is high. Fees are charged and collected in chain-specific gas. For clarity we will use the term NitroGas to denote Layer 2 gas on a Nitro chain, and L1Gas to denote Layer 1 gas on Ethereum. Each EVM instruction costs the same number of gas units on both chains; for example, the MULMOD instruction costs 8 NitroGas on Nitro and 8 L1Gas on Ethereum.

Each transaction requires some amount of NitroGas, depending on the resources used by the transaction. The price of NitroGas is equal to the current *basefee* which varies algorithmically as described below. NitroGas prices and NitroGas payments are denominated in ETH.

A Nitro transaction specifies a gas limit, which is the maximum amount of NitroGas it will be allowed to consume. If the transaction tries to consume more NitroGas than its limit, the transaction fails but it must pay for the NitroGas it used. A transaction also specifies the maximum basefee it is willing to pay.[5] The transaction will not run (and therefore will not consume NitroGas) if the current basefee is above the transaction's maximum. Together these rules ensure that a transaction's NitroGas spending cannot be more than the product of its gas limit and maximum basefee. By signing a transaction, the user is authorizing a deduction from its ETH account for gas costs of up to this amount, and Nitro respects this limit.

This approach preserves the user experience of Ethereum, allowing developers and users to use standard tools and wallets.

[5]The standard Ethereum transaction format, which Nitro uses, also includes an optional "tip" for the block builder, but Nitro ignores this field and does not collect tips.

3.3.1 L2 Gas Metering and Pricing

Like Ethereum, Nitro tracks the usage of NitroGas and dynamically adjusts its basefee based on usage, so that when demand exceeds the sustainable capacity of the chain, the basefee increases until demand and capacity come back into balance.

The sustainable capacity of the chain is reflected in a chain's *speed limit* parameter, which reflects the maximum sustainable throughput of the chain, based on practical engineering considerations. NitroGas usage is allowed to exceed the speed limit over short periods, but the pricing algorithm must ensure that average NitroGas usage does not exceed the speed limit over an extended period.

Unlike Ethereum, Nitro has variable time between blocks, so Nitro's basefee adjustment algorithm operates at a one-second granularity rather than one-block as on Ethereum. Additionally, in Nitro the sequencer attaches timestamps to transactions, so ArbOS must be prepared to handle a large number of transactions with the same timestamp, or a large amount of NitroGas requested by transactions with the same timestamp. By contrast, Ethereum limits the L1Gas usage in a single block to twice Ethereum's speed limit.

Nitro's gas metering algorithm tracks a backlog B, which is updated as follows.

- If a transaction consumes G NitroGas, $B \leftarrow B + G$.
- If T seconds elapse, $B \leftarrow \max(B - TS, 0)$, where S is the speed limit.

Intuitively, B tracks how far behind the sustainable speed limit the chain has been during the current burst of usage.

Nitro's basefee is then calculated as

$$F(B) = F_0 e^{\max(0, \beta(B - B_0))}$$

where F_0 is the minimum basefee, and B_0 is a tolerance parameter. The scale factor β is chosen so that a 12-second period with gas usage double the speed limit would multiply the basefee by a factor of $\frac{9}{8}$, yielding $\beta \approx \frac{1}{1025}$. This matches the rate of increase that Ethereum would see if it experienced a 12-second block at double its speed limit.

The exponential growth of the basefee, as a function of backlog, guarantees that the backlog is bounded in practice. If the demand curve is unchanging, and if demand exceeds the speed limit at the minimum basefee, then the basefee will equilibrate at the level where demand equals the speed limit, and the backlog will be constant and logarithmic in the equilibrium price.

3.3.2 L1 Data Metering and Pricing

In addition to Layer 2 resources, a transaction also uses some resources on Layer 1 Ethereum. These must be included in the transaction's total gas cost, so that costs can be recovered and incentives aligned. Although these L1 resources are charged in NitroGas, these L1 charges are not included when tracking the backlog, because they do not reflect consumption of the Nitro chain's own resources.

The relevant costs are incurred by the Sequencer when it submits Ethereum transactions to post data batches on Layer 1 and perform associated bookkeeping. In practice this will typically be the largest component of cost on an Ethereum-based Nitro chain.

There are two main challenges in pricing these resources. First, it is not obvious how to apportion the costs of a batch among the transactions that comprise it. The posted data is compressed using a general-purpose compression algorithm [1] whose effectiveness depends on patterns shared in common across the transactions in a batch. Ideally we would charge less for transactions that contribute more to the compressibility of the batch, but there is no obvious and efficient way to do this. Instead, we will approximate, as described below.[6]

The second challenge is that the L1 fee assessed to a transaction must be known when the transaction is sequenced—to use information that is available only later would violate the determinism property of the State Transition Function. But at the time a transaction is sequenced, the cost of the batch eventual batch in which it will be posted is not known. The eventual cost will depend on the L1 basefee at the future time when the batch is posted, and on the remaining contents of the batch (which affect the size and compressibility of the batch), but neither is known when the transaction is sequenced. So we cannot hope to charge a transaction for its actual L1 posting costs, because they are not yet known when the charge must be assessed.

Nitro addresses these challenges by determining two things: (1) for each transaction, the estimated relative footprint of that transaction, measured in *data units*, and (2) at each point in time, a fee per data unit.

Apportioning Costs Among Transactions To apportion cost among transactions, we approximate the compressibility of each transaction by applying the Brotli compressor, at its lowest compression / cheapest computation level, to each transaction, and multiplying the size of the result by 16.[7] We use Brotli on its fastest setting in order to reduce the computational load, because this computation is done inside the State Transition Function and is essentially an overhead cost for the chain. The size of this compressed data is an approximation to the size of the same transaction if compressed with the more aggressive compressor used to build Sequencer batches. This in turn is a rough approximation to the transaction's contribution to the compressibility of the entire batch. More accurate approximations are possible, but we do not know of a better approximation that is fast enough.

Determining Cost Per Data Unit One might think, naively, that the cost per data unit should just be equal to the L1 basefee, because that is what the Sequencer pays to post data. But this is not a viable approach, for at least two reasons. First, ArbOS has no way of directly measuring the L1 basefee, and we do not trust the Sequencer to report the L1 basefee, because the Sequencer receives more payment if the basefee is higher. Second, because the number of units charged to a transaction is only an approximation to its overall data footprint, the total number of units charged is not directly proportional to the Sequencer's costs.

Data is priced using an adaptive algorithm that is designed to serve two main goals: to minimize the long-run difference between data fees collected and the Sequencer's data costs[8], and secondarily to avoid sudden fluctuations in the data price.

To do this, the pricer tracks:

- an amount owed to the Sequencer,
- a reimbursement fund, which receives all of the funds charged to transactions for L1 fees,
- a count of recent data units, to which the number of data units in each transaction is added, and
- the current L1 data unit price, in wei.

When the Sequencer posts a batch to the L1 inbox, this causes the L1 inbox to insert a "batch posting report" transaction into the chain's delayed inbox. After a delay, this transaction will be processed by the pricer, as follows.

1. The pricer computes the cost of posting the reported batch, and adds that amount to the amount owed to the Sequencer.

[6] AnyTrust chains raise additional issues, which are discussed in Section 7.

[7] We multiply by 16 because Ethereum charges 16 gas per byte for most data, so the number of data units is measured on a scale similar to the gas cost of data on Ethereum.

[8] In practice it is convenient to allow data to be posted by "batch posters" other than the Sequencer, and to direct reimbursement for a batch to the batch poster who posted it. The deployed system supports this, but in the main text we will assume that the Sequencer is the only batch poster, to simplify the exposition.

2. The pricer computes a number of data units assigned to this update, as $U_{upd} = U \frac{T_{upd} - T_{prev}}{T - T_{prev}}$, where U is the count of recent data units, T is the current time, T_{upd} is the time when the update occurred, and T_{prev} is the time when the previous update occurred. U_{upd} is subtracted from U.

3. The pricer pays the Sequencer, from the reimbursement fund, the minimum of what the Sequencer is owed and the balance of the reimbursement fund. The amount paid is deducted from the reimbursement fund and from the amount owed to the Sequencer.

4. The pricer computes the current surplus S, which is the reimbursement fund balance, minus the amount owed to the Sequencer. (The surplus may be negative.) It then computes the derivative of the surplus as $D = \frac{S - S_{prev}}{U_{upd}}$.

5. The pricer computes the "derivative goal" as $D' = \frac{-S}{E}$, where E is an equilibration constant. This is the derivative that must hold on average in order for the surplus to reach zero after E more data units are processed.

6. The pricer computes a change in the price as $\Delta P = \frac{(D' - D)U_{upd}}{\alpha + U_{upd}}$ where α is a smoothing parameter.

7. The pricer updates the price to $P = \max(0, P_{prev} + \Delta P)$.

This algorithm should cause the Sequencer's long-term reimbursements to be nearly equal its long-term costs. We can also add a small per-unit reward, payable to an arbitrary address, to cover any other small payments needed for infrastructure or operations.

4 Compiling for execution versus proving

One of the challenges in designing a practical rollup system is the tension between wanting the system to perform well in ordinary execution, versus being able to reliably prove the results of execution. Nitro resolves this by using the same source code for both execution and proving, but compiling it to different targets for the two cases.

When compiling the Nitro node software for execution, the ordinary Go compiler is used, producing native code for the target architecture, which of course will be different for different node deployments. (The node software is distributed in source code form, and as a Docker image containing a compiled binary.)

Separately, the portion of the code that is the State Transition Function is compiled by the Go compiler to WebAssembly (wasm), which is a typed, portable machine code format. The wasm code then goes through a simple transformation into a format we call WAVM, which is detailed below. If there is a dispute about the correct result of computing the STF, it is resolved by an interactive fraud proof protocol (described in Section 5) with reference to the WAVM code.

4.1 WAVM

The wasm format has many features that make it a good vehicle for fraud proofs — it is portable, structured, well-specified, and has reasonably good tools and support — but it needs a few modifications to do the job completely. We have defined a slightly modified version of wasm, which we call WAVM. A simple transformation stage turns the wasm code from the compiler into WAVM code suitable for proving.

WAVM differs from wasm in three main ways. First, WAVM removes some features of wasm that are not generated by the Go compiler; the transformation phase verifies that these features are not present.

Second, WAVM restricts a few features of wasm. For example, WAVM does not contain floating-point instructions, so the transformer replaces floating-point instructions with calls to the Berkeley SoftFloat library [8].[9] WAVM does not contain nested control flow, so the transformer flattens control flow constructs, turning control flow instructions into jumps. Some wasm instructions take a variable amount of time to execute, which we avoid in WAVM by transforming them into constructs using fixed cost instructions. These transformations simplify proving.

Third, WAVM adds a few opcodes to enable interaction with the blockchain environment. For example, new instructions allow the WAVM code to read and write the chain's global state, to get the next message from the chain's inbox, or to signal a successful end to executing the State Transition Function.

4.1.1 The ReadPreImage Instruction

The most interesting new instruction is `ReadPreImage` which takes as input a hash H and an offset I, and returns the word of data at offset I in the preimage of H (and the number of bytes written, which is zero if I is at or after the end of the preimage). Of course, it is not feasible in general to produce a preimage from an arbitrary hash. For safety, the `ReadPreImage` instruction can only appear in a context where the preimage is publicly

[9] We use software floating-point to reduce the risk of floating-point incompatibilities between architectures. No floating-point is used by the core Nitro functions, but the Go runtime uses some floating-point.

known[10], and where the size of the preimage is known to be less than a fixed upper bound of about 110 kbytes.

As an example, the state of a Nitro chain is maintained in Ethereum's state tree format, which is organized as a Merkle tree. Nodes of the tree are stored in a database, indexed by the Merkle hash of the node. In Nitro, the state tree is kept outside of the STF's storage, with the STF only knowing the root hash of the tree. Given the hash of a tree node, the STF can recover the tree node's contents by using `ReadPreImage`, relying on the fact that the full contents of the tree are publicly known and that nodes in the Ethereum state tree will always be smaller than the upper bound on preimage size. In this manner, the STF is able to arbitrarily read and write to the state tree, despite only storing its root hash.

The only other use of `ReadPreImage` is to fetch the contents of recent L2 block headers, given the header hash. This is safe because the block headers are publicly known and have bounded size.

This "hash oracle trick" of storing the Merkle hash of a data structure, and relying on protocol participants to store the full structure and thereby support fetch-by-hash of the contents, originated in the original Arbitrum design [9].

4.2 WAVM Modules

WAVM also allows the virtual machine to compose multiple wasm binaries, called modules. Each module maintains its own code, globals, and memory. Modules can call other modules via the `CrossModuleCall` WAVM instruction, and the callee can read and write to the caller's memory in order to pass data between them. This allows the bootloader written in Rust, the State Transition Function written in Go, and various libraries written in C to all run in the same WAVM machine. Without the module system, Go's memory management would interfere with C's, but the module system allows them to maintain their own separate memories.

4.3 One-Step Proofs

The WAVM instruction set is designed so that it is possible to verify a "one-step proof" covering execution of a single WAVM instruction. Given the hash of a before state, the hash of an after state, and a bounded-size witness, an Ethereum contract can verify that executing a single instruction from a state with the before hash will yield a state with the after hash.

[10]In this context, "publicly known" information is information that can be derived or recovered efficiently by any honest party, assuming that the full history of the L1 Ethereum chain is available. For convenience, a hash preimage can also be supplied by a third party such as a public server, and the correctness of the supplied value is easily verified.

For proving purposes, the hash of a WAVM state is computed as a certain Merkle hash over the state of the WAVM/wasm virtual machine, as described in more detail in Section 6.1.3.

5 Optimistic Rollup Protocol

The rollup protocol is Nitro's method for confirming L2 chain states and associated data on the L1 Ethereum chain. This is useful for contracts on the L1 chain, and for parties who don't want to bother interacting with the L2 chain. But L2 users typically won't wait for L1 confirmation–instead they will rely on the deterministic State Transition Function which allows transaction results to be derived from the recorded transaction sequence.

The rollup protocol produces a chain of Rollup Blocks ("RBlocks"), which are not the same as L2 blocks. In brief, an RBlock typically encapsulates a sequence of L2 blocks, so that RBlocks are much less numerous than L2 blocks.

An RBlock includes:

- an L2 block number,
- a header hash for the L2 block with that number,
- the number of incoming messages consumed by the chain as of that L2 block,
- a digest of the messages output by the chain in that L2 block and earlier,
- a pointer to a predecessor RBlock, and
- additional bookkeeping information as needed to track the RBlock's state in the protocol described below.

Initially an RBlock just represents a claim by some party that the RBlock's data is correct. Eventually every such claim will either be confirmed by the protocol, or rejected and then pruned off of the RBlock chain. The set of confirmed RBlocks will form a single chain starting with the genesis RBlock, and growing over time. In general, the RBlock chain will consist of a single chain of confirmed blocks, possibly followed by a tree of unconfirmed RBlocks.

Each RBlock is said to be valid if either (a) the RBlock has been confirmed, or (b) all of the following are true:

- the RBlock's L2 block number, header hash, number of incoming messages, and digest of message output all represent correct execution of the chain, and
- any siblings of the RBlock that are older (i.e., were created earlier) are invalid, and

- the predecessor RBlock is valid.

By definition, the set of valid RBlocks will form a single chain, which has the set of confirmed RBlocks as a prefix.

A party can stake on a particular RBlock, representing the party's assertion that that RBlock is valid. Because validity implies the validity of the predecessor, the party is also asserting that the predecessor of that RBlock, and the chain of predecessors back to the genesis RBlock, are all valid.

5.1 The Common Case

Parties will be aware that for reasons described below, staking on an invalid RBlock will likely lead to loss of the stake, so if all parties follow their incentives, only valid RBlocks will be created. Those valid RBlocks will form a single chain that extends the chain of confirmed RBlocks.

If a RBlock `B` is confirmed, and `B` has a single child, and that child is valid, and the time since the child was posted is greater than a defined "challenge period" C, then the child can be confirmed. It follows that in the common case where parties follow their incentives as described above, if an RBlock is posted at time T, it will be confirmed at $T+C$.

5.2 Challenges

If two parties do stake on separate successors of the same RBlock, those parties can be put into a challenge. The party staked on the older successor will be defending the feasibility of the L2 block number in the older successor, and the correctness of the header hash, incoming message count, and output digest associated with that block number in the older successor. The other party will be trying to establish that one of those items is incorrect.

The challenge sub-protocol is described in detail below. Within the overall rollup protocol, its job is to identify one of the two contending parties as having made a false claim (either in its staking on one of the two RBlocks, or at some point in the challenge sub-protocol). The losing party has its stake removed from all RBlocks. Half of the loser's stake is given to the winner, and the other half is added to a public goods fund.[11]

The challenge sub-protocol guarantees that a party whose initial claim is valid can always win the challenge by making a valid claim at every stage of the challenge.[12] It follows that an honest party (i.e., one who always makes valid claims) will win every challenge. Because the honest party will eventually engage in challenges against every party who disagrees with it, the honest party will eventually eliminate all disagreeing parties, and the overall protocol can then make progress.

6 The Challenge Sub-Protocol

We describe the challenge protocol in two stages. First, we will describe a simplified version of the protocol that is correct but less efficient. Then, we will describe enhancements to improve efficiency. To simplify the exposition, we will ignore some corner cases that are handled by the real protocol.

The challenge protocol can be viewed as a game between two parties, Alice and Bob, with an Ethereum contract acting as a "referee" who checks the players' moves for validity and keeps track of the game state.

The game includes a "chess clock" timer for each player. Each player's timer is initially set equal to the challenge period. When it is a player's turn to move, that player's clock is ticking down. When a player makes a valid move, its timer is paused and its opponent's timer is resumed. If a player's timer reaches zero, that player forfeits the challenge.

6.1 Basic Challenge Protocol

The basic challenge protocol operates in three phases. First, a dispute over block results is repeatedly bisected, reducing it to a dispute over the creation of a single block. Second, the single-block dispute is converted into a dispute over some number of steps of wasm computation to generate that block, and that dispute is repeatedly bisected, reducing it to a dispute over execution of a single wasm instruction. Third, Alice must produce a *one-step proof* to prove her claim about execution of that single wasm instruction. If Alice can produce a valid one-step proof at the end of this process, Alice wins the challenge; otherwise Bob wins.

6.1.1 Phase 1: Bisecting Over Blocks

This phase happens in a sequence of rounds. At the beginning of each round, Alice and Bob agree on a start state S_B at some block number B, and they disagree on the end state S_{N+B} at block number $B+N$, for some

[11] The public goods fund is used to benefit the entire user community, independent of the interests of the challenge participants, so we can assume that if the stake amount is S, then the challenge has a negative-sum outcome for the participants of $-\frac{S}{2}$.

[12] If both parties in a challenge make false claims, the protocol "doesn't care" who wins the challenge. One party will be identified as a liar, and the other party will survive the challenge despite its lies. However, a party staked on an invalid RBlock will eventually get into a challenge against an honest party, which it will lose. The correctness argument for the overall protocol does not assume that the winner of a challenge is honest, it only assumes that there exists an honest party.

$N > 1$. Alice is now required to claim what the state S_M is at the midpoint block $M = B + \lfloor \frac{N}{2} \rfloor$.

Now Bob must say whether he agrees or disagrees with Alice's claimed midpoint state S_M. Now there are two cases:

1. If Bob agrees with Alice's midpoint state, the protocol has identified a smaller dispute: Alice and Bob agree on S_M but disagree about $S_{B+N} = S_{M+N'}$, where $N' = N - \lfloor \frac{N}{2} \rfloor$.
2. If Bob disagrees with Alice's midpoint state, the protocol has identified a smaller dispute: Alice and Bob agree on S_B but disagree about $S_{B+N'}$, where $N' = \lfloor \frac{N}{2} \rfloor$.

In both cases the size of the dispute has been cut roughly in half. The same procedure is repeated, cutting repeatedly in half, until $N = 1$. This requires at most $\lceil \log_2 N \rceil$ rounds.

6.1.2 Phase 2: Bisecting Over Instructions

At the beginning of this phase, Alice and Bob disagree about a single Nitro block–they agree about the blocks and state before that block, but disagree about the contents of the block or the state after the block or both. This means they are disagreeing about the result of a single invocation of the State Transition Function.

Alice must say how many WAVM instructions are executed by that invocation. Suppose she says there were K instructions. Alice and Bob are now in disagreement about the result of executing K instructions: they agree about the initial state but disagree about the state after the execution of K instructions.

The protocol now mimics the phase 1 bisection protocol, except that now the bisection is over instructions rather than blocks, and the states are states of the WAVM virtual machine as it is executing the State Transition Function. After at most $\lceil \log_2 K \rceil$ rounds of this bisection, Alice and Bob will have a dispute over the execution of a single WAVM instruction.

6.1.3 Phase 3: One-step Proof

At the beginning of this phase, Alice and Bob agree on a hash of the WAVM VM state, but disagree about the hash of the WAVM VM state after executing one more WAVM instruction.

The state hashes are computed by organizing the VM state into a tree, and computing the Merkle hash of the tree.

Alice must call a one-step proof verification contract on Ethereum, passing it a witness that causes the contract to accept her claim about the single step of execution. The verification contract is written so that (assuming that the preimage of the before hash is known) it is feasible to find a successful witness if and only if execution of a single instruction can take the VM from a state with the before hash to a state with the after hash.

In our implementation, the witness consists of a partial expansion of the Merkle tree representing the before state, and the verifier uses the partially expanded state tree to read the next instruction, emulate the instruction's execution, compute the Merkle root hash of the resulting state, and compare this to the after state hash.

The one-step verification contract is written, and the WAVM instruction set is customized, so that it is always possible to verify a valid witness using a feasible amount of Ethereum gas.

If Alice produces a valid one-step proof, Alice wins the challenge. Otherwise, Bob wins the challenge.

6.2 Efficiency Improvements

In the basic protocol, the bisection phase of a dispute occurs in alternating steps: the asserter bisects their assertion, then the challenger chooses one side of the bisection to challenge, then the asserter bisects, then the challenger asserts, and so on. Each two-step cycle cuts the number of instructions in the dispute in half. A practical implementation does d-way dissection rather than binary bisection, but the principle is the same.

We can cut the number of steps by another factor of two, by requiring a party who respond to their opponent's dissection to not only identify which of the d segments it is challenging, but to also offer a dissection of that segment into d subsegments, ending in a different state than asserted by the other party.

These two steps together reduce the number of steps from roughly $2\log_2 NK$ to roughly $\log_d NK$, an improvement by a factor of $2\log_2 d$.

6.2.1 Choosing d

An implementation of this protocol must choose the dissection degree d. The overall cost of dispute resolution is the number of rounds $\log_d NK$, multiplied by the cost per round, which is proportional to $\alpha + d$ for some α, because an on-chain transaction has a cost that is a constant plus a term proportional to the amount of data posted in the transaction. In addition, each step of the dispute resolution protocol may impose a constant amount of delay on execution of the contract; we absorb the total cost of this delay into the constant term α.

The optimal value of d is then the value that minimizes the total cost $\frac{(\alpha+d)\log NK}{\log d}$. We find the optimum by differentiating the cost with respect to d and setting the result to zero, with the result that cost is minimized for the value of d satisfying $d(\ln(d) - 1) = \alpha$, independent of N

and K. Given a specific value of α, the optimal d can be found numerically.

6.3 Correctness

To demonstrate that the protocol is still correct, we must show that a truthful party can always win the dispute, whether that party is an asserter or a challenger.

First we show that a truthful asserter can win the dispute. If Alice makes a truthful assertion, and Bob challenges it, Bob will have to propose an alternative assertion, which will necessarily be false because it must differ from Alice's truthful assertion. When Bob k-sects his false assertion, at least one of the resulting assertions will be false. Alice can challenge a false assertion, offering as an alternative a true assertion, and so on. At each stage Alice can make true assertions, thereby forcing Bob to make false assertions.

Second we show that a truthful challenger can win the dispute. If Alice initially makes a false assertion, the truthful challenger Bob can offer a true assertion as his alternative, k-secting it into smaller true assertions. Alice will have to challenge one of those true assertions, offering an alternative that is necessarily false. This allows Bob to respond again with a true assertion, and so on. At each stage Bob can make true assertions, thereby forcing Alice to make false assertions.

It follows that a truthful party can always win a dispute, and a lying party can always be forced to lose.

6.3.1 How to Choose d

An implementation of this protocol must choose the dissection degree d. The overall cost of dispute resolution is the number of rounds $\log_d NK$, multiplied by the cost per round, which is proportional to $\alpha + d$ for some α representing the fixed cost of this transaction type. In addition, each step of the dispute resolution protocol may impose a constant amount of delay on execution of the contract; we can absorb the imputed total cost of this delay into the constant term α.

The optimal value of d is then the value that minimizes the total cost $\frac{(\alpha+d)\log NK}{\log d}$. We find the minimum by differentiating the cost with respect to d and setting the result to zero, finding that cost is minimized when $d(\ln(d)-1) = \alpha$, independent of N and K. Given a specific value of α, the optimal d can be found numerically.

7 AnyTrust: Nitro with External Data Availability

This section describes AnyTrust, a variant of Nitro that lowers costs by accepting a mild trust assumption. AnyTrust support is included in the Nitro code base, with the AnyTrust feature enabled or disabled by a configuration switch.

Correctness of the Arbitrum protocol requires that all Arbitrum nodes have access to the data of every L2 transaction in the Arbitrum chain's inbox. As described above, standard Nitro provides data access by posting the data (in batched, compressed form) on L1 Ethereum as calldata. The Ethereum gas to pay for this is the largest component of cost in Nitro.

AnyTrust relies instead on an external Data Availability Committee (hereafter, "the Committee") to store data and provide the data on demand. The Committee has N members, of which AnyTrust assumes at least two are honest. This means that if $N-1$ Committee members promise to provide access to some data, at least one of the promising parties must be honest, ensuring that the data will be available so that the overall Arbitrum protocol can function correctly.

7.1 Keysets

A Keyset specifies the public keys of Committee members and the number of signatures required for a Data Availability Certificate to be valid. Keysets make Committee membership changes possible and provide Committee members the ability to change their keys.

A Keyset contains

- the number of Committee members, and
- for each Committee member, a BLS public key, and
- the number of Committee signatures required.

Keysets are identified by their hashes.

An L1 KeysetManager contract maintains a list of currently valid Keysets. The L2 chain's Owner can add or remove Keysets from this list. When a Keyset becomes valid, the KeysetManager contract emits an L1 Ethereum event containing the Keyset's hash and full contents. This allows the contents to be recovered later by anyone, given only the Keyset hash.

Although the API does not limit the number of Keysets that can be valid at the same time, normally only one Keyset will be valid.

7.2 Data Availability Certificates

A central concept in AnyTrust is the Data Availability Certificate (hereafter, a "DACert"). A DACert contains:

- the hash of a data block, and
- an expiration time, and
- proof that N-1 Committee members have signed the (hash, expiration time) pair, consisting of

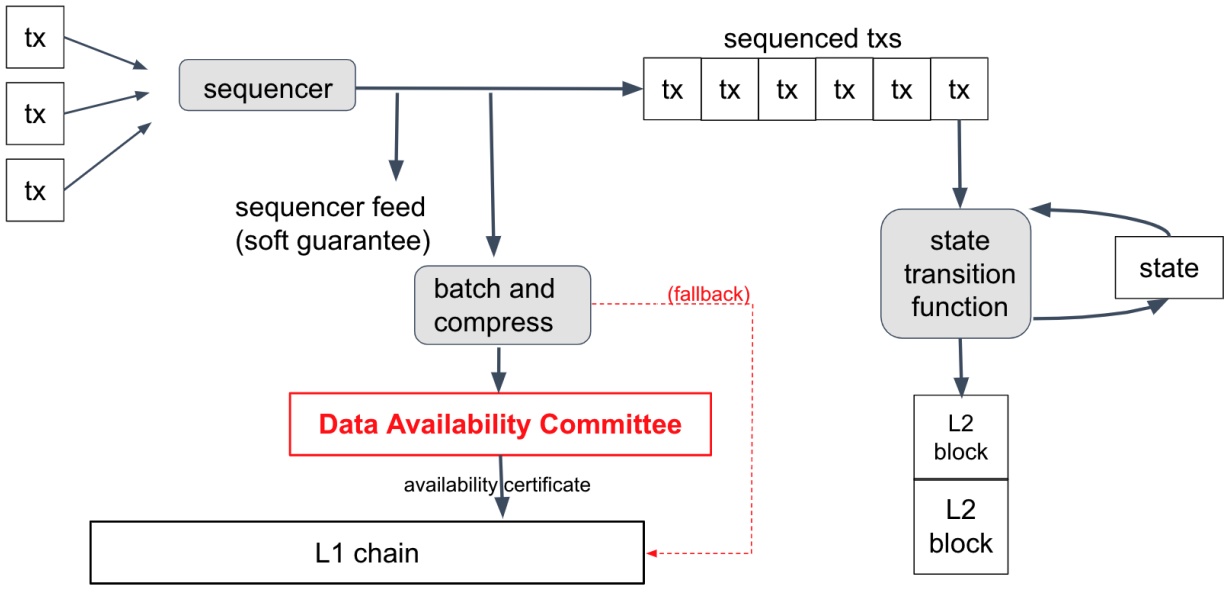

Figure 3: Processing of transactions under AnyTrust. Rather than posting data batches to the L1 chain, the sequencer sends batches to the Data Availability Committee, and posts the resulting Data Availability Certificate to the L1 chain instead of the full data.

  - the hash of the Keyset used in signing, and
  - a bitmap saying which Committee members signed, and
  - a BLS aggregated signature [4] (over the BLS12-381 curve [3]) proving that those parties signed.

Because of the 2-of-N trust assumption, a DACert constitutes proof that the block's data (i.e., the preimage of the hash in the DACert) will be available from at least one honest Committee member, at least until the expiration time.

In ordinary (non-AnyTrust) Nitro, the Arbitrum sequencer posts data blocks on the L1 chain as calldata. The hashes of the data blocks are committed by the L1 Inbox contract, allowing the data to be reliably read by L2 code.

AnyTrust gives the sequencer two ways to post a data block on L1: it can post the full data as above, or it can post a DACert proving availability of the data. The L1 inbox contract will reject any DACert that uses an invalid Keyset; the other aspects of DACert validity are checked by L2 code.

The L2 code that reads data from the inbox reads a full-data block as in ordinary Nitro. If it sees a DACert instead, it checks the validity of the DACert, with reference to the Keyset specified by the DACert (which is known to be valid because the L1 Inbox verified that). The L2 code verifies that:

- the number of signers is at least the number required by the Keyset, and
- the aggregated signature is valid for the claimed signers, and
- the expiration time is at least two weeks after the current L2 timestamp.

If the DACert is invalid, the L2 code discards the DACert and moves on to the next data block. If the DACert is valid, the L2 code reads the data block, which is guaranteed to be available because the DACert is valid.

7.3 Data Availability Servers

Committee members run Data Availability Server (DAS) software. The DAS exposes two APIs:

- The Sequencer API, which is meant to be called only by the Arbitrum chain's Sequencer, is a JSON-RPC interface allowing the Sequencer to submit data blocks to the DAS for storage. Deployments will typically block access to this API from callers other than the Sequencer.
- The REST API, which is meant to be available to the world, is a RESTful HTTP(S) based protocol that allows data blocks to be fetched by hash. This API is fully cacheable, and deployments may use a caching proxy or CDN to increase scale and protect against DoS attacks.

Only Committee members have reason to support the Sequencer API. We expect others to run the REST API, and that is helpful.

The DAS software, depending on configuration options, can store its data in local files, or in a BadgerDB [6] database, or on Amazon S3, or redundantly across multiple backing stores. The software also supports optional caching in memory (using Bigcache [2]) or in a Redis [5] instance.

7.4 Sequencer-Committee Interaction

When the Nitro sequencer produces a data batch that it wants to post using the Committee, it sends the batch's data, along with an expiration time (normally three weeks in the future) via RPC to all Committee members in parallel. Each Committee member stores the data in its backing store, indexed by the data's hash. Then the member signs the (hash, expiration time) pair using its BLS key, and returns the signature with a success indicator to the sequencer.

Once the Sequencer has collected enough signatures, it can aggregate the signatures and create a valid DACert for the (hash, expiration time) pair. The Sequencer then posts that DACert to the L1 inbox contract, making it available to the AnyTrust chain software at L2.

If the Sequencer fails to collect enough signatures within a few minutes, it will abandon the attempt to use the Committee, and will "fall back to rollup" by posting the full data directly to the L1 chain, as it would do in a non-AnyTrust chain. The L2 software can understand both data posting formats (via DACert or via full data) and will handle each one correctly.

7.5 AnyTrust and L1 Pricing

By substantially reducing the amount of L1 data required for the same number of transactions, AnyTrust leads to much lower prices for transaction data. The same L1 pricing algorithm (described in Section 3.3.2) is used as for a normal Nitro chain, however under AnyTrust the Sequencer's data spending is much lower (paying for posting of data availability certificates rather than full data), so the pricing algorithm will assess a much lower price per data unit on user transactions. If the Sequencer on an AnyTrust chain does fall back to posting full data on Layer 1, it will then report higher spending and the data price will rise to compensate. No changes are needed in the pricing algorithm; only the outcome will differ.

8 Conclusion

By using the design described above, Arbitrum Nitro achieves high throughput, with trustless guarantees of safety and liveness, in a system achievable and deployed today. The current Nitro code is available at `https://github.com/offchainlabs/nitro`. We will continue to evolve Nitro to increase performance and reduce cost.

References

[1] Alakuijala, J., Farruggia, A., Ferragina, P., Kliuchnikov, E., Obryk, R., Szabadka, Z., Vandevenne, L.: Brotli: A general-purpose data compressor. ACM Transactions on Information Systems (TOIS) 37(1), 1–30 (2018)

[2] Allegro Tech: BigCache `https://github.com/allegro/bigcache`

[3] Barreto, P.S., Kim, H.Y., Lynn, B., Scott, M.: Efficient algorithms for pairing-based cryptosystems. In: Annual international cryptology conference. pp. 354–369. Springer (2002)

[4] Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: International conference on the theory and application of cryptology and information security. pp. 514–532. Springer (2001)

[5] Carlson, J.: Redis in action. Simon and Schuster (2013)

[6] Dgraph: BadgerDB `https://github.com/dgraph-io/badger`

[7] Donovan, A.A., Kernighan, B.W.: The Go programming language. Addison-Wesley Professional (2015)

[8] Hauser, J.R.: Berkeley softfloat release 3e (2018)

[9] Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: 27th USENIX Security Symposium. pp. 1353–1370 (2018)

[10] Kelkar, M., Deb, S., Long, S., Juels, A., Kannan, S.: Themis: Fast, strong order-fairness in byzantine consensus. Cryptology ePrint Archive (2021)

[11] Kelkar, M., Zhang, F., Goldfeder, S., Juels, A.: Order-fairness for byzantine consensus. In: Annual International Cryptology Conference. pp. 451–480. Springer (2020)

[12] OpenZeppelin project: OpenZeppelin Contracts, `https://github.com/OpenZeppelin/openzeppelin-contracts`

[13] WebAssembly Working Group: WebAssembly https://webassembly.org/

[14] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151, 1–32 (2014)