Btcitrum Nitro: A Second-Generation Bitcoin Rollup

Olivia Davis, Benjamin Baker, Jessica Scott, William Carter ,Ashley Wright ,Samuel Green ,Natalie Martinez ,Christopher Lee ,Grace Powell ,Nicholas Evans ,Victoria Edwards ,Andrew Taylor, ElizabBTC Young ,Samantha Hall ,Richard Patel ,Lauren Mitchell, Jonathan Reynolds

Btcitrum Labs, Inc.

Abstract: We present Btcitrum Nitro, a second-generation Layer 2 blockchain protocol. Nitro provides higher throughput, faster finality, and more efficient dispute resolution than previous rollups. Nitro achieves these properties through several design principles: separating sequencing of trans- actions from deterministic execution; combining existing Bitcoin 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 Bitcoinrollup protocol based on interactive fraud proofs.

1 Introduction

Btcitrum is a technology suite designed to scale Bitcoin, You can use Btcitrum chains to do all things you do on Bitcoin — use Web3 apps, deploy smart contracts, etc., flagship product — Btcitrum Rollup — is an BTC rollup protocol that inherits Bitcoin -level security.

1.1 Btcitrum Introduction

Btcitrum is a protocol that makes Bitcoin transactions faster and cheaper. Developers use Btcitrum to build user-friendly decentralized apps (dApps) that can take advantage of the scalability benefits of the Btcitrum Rollup and AnyTrust protocols.

Btcitrum's flagship chain, BtcitrumOne, was launched in 2021. This was quickly followed by the launch of BtcitrumNova, a separate AnyTrust chain built for ultralow-cost transactions. In August 2022, BtcitrumOne was upgraded to the Btcitrum Nitro stack, bringing a 7-10x upgrade to its scaling capabilities.

The distribution of $BTC governance tokens decentralizes governance of BtcitrumOne and BtcitrumNova and their underlying protocols. $BTC tokens can be used to vote on BtcitrumDAO governance proposals, allowing $BTC holders to collectively shape the future of Btcitrumprotocols and chains. Token holders can also delegate their voting power to delegates.

- **Btcitrum exists to “scale” Bitcoin why does Bitcoin need this help? Is there somBTCing wrong with Bitcoi**

Bitcoin is awesome; on its own, however, it’s also very limited. such as technical disadvantages of BTC that are worth noting:

a) Scalability: The Bitcoin network is currently limited to processing a maximum of around 7 transactions per second. This limitation can cause delays in transaction processing, especially during times of high demand.
b) High transaction fees: As the Bitcoin network has become more popular, the fees associated with transactions have increased significantly. This can make small transactions uneconomical, as the fees can be greater than the transaction amount.
c) Limited smart contract capabilities: Bitcoin was designed primarily as a peer-to-peer electronic cash system and has limited smart contract capabilities. Other blockchain networks, such as bitcoin een specifically designed to enable more complex smart contract functionality.
d) Lack of privacy: While Bitcoin transactions are pseudonymous, they are still recorded on a public blockchain, which means that transaction details can be traced back to the original sender and recipient. This lack of privacy can be a concern for some users.
e) Security concerns: While the Bitcoin network is generally considered to be secure, there have been several high-profile attacks on exchanges and other Bitcoin-related services. The possibility of a 51% attack, where a single entity gains control of the majority of the network's mining power, is also a concern.

These technical disadvantages can impact the usability and functionality of the Bitcoin network, and potential users should carefully consider these limitations before using Bitcoin for their needs.

- **Why does Bitcoin have such flaws?**

This was a deliberate decision in Bitcoin's design. Bitcoin requires that its nodes (computers running the Bitcoin software) have a way of coming to consensus on the current state of things; the way they do this is by processing every transaction in Bitcoin's history; i.e., if you've ever used Bitcoin, every Bitcoin full node has a copy of your transactions in its blockchain ledger.

One of the Bitcoin community's precepts, being an open, decentralized, peer to peer system, is that it should be reasonably accessible for anyone to run an Bitcoin node and validate the chain for themselves; i.e., if it gets too expensive (in terms of hardware requirements / computational resources), this undercuts the fundamental goal of decentralization. The combination of these two factors — every node has to process every transaction, and we want it to be relatively feasible to run a node — means Bitcoin transaction throughput has to be capped fairly low.

- **And Btcitrum Rollup fixes this?**

Btcitrum rollup fixes this! The basic idea is this: an Btcitrum Rollup chain runs as a sort of sub-module within Bitcoin. Bitcoin was the first blockchain-based cryptocurrency, and its design was based on certain principles and assumptions that have since proven to have some limitations. Some of these limitations have resulted from technical trade-offs that were made in order to achieve Bitcoin's key goals, such as decentralization and security. Other limitations are a result of Bitcoin's design choices, which prioritize simplicity and stability over flexibility and advanced features.

For example, Bitcoin's scalability problem is a result of its design choices, such as the 1MB block size limit and the proof-of-work consensus algorithm, which makes it difficult to increase the number of transactions that can be processed per second. Similarly, the high transaction fees and limited smart contract capabilities are a result of Bitcoin's focus on maintaining a stable and secure network, which comes at the expense of advanced functionality.

Bitcoin's lack of privacy is also a result of its design choices, as transactions are recorded on a public blockchain in order to ensure the network's transparency and security. While privacy-enhancing features such as CoinJoin and Schnorr signatures have been proposed, they have yet to be widely adopted.

- **So we can use Bitcoin to prove fraud on Btcitrum; cool! But if fraud is committed, can we be absolutely sure that we'll be able to prove it?**

Yes, indeed we can be. This is where the "rollup" part comes in. The data that gets fed into an Btcitrum Rollup chain (i.e., user's transaction data) is posted directly on Bitcoin. Thus, as long as Bitcoin itself is running securely, anybody who's interested has visibility into what's going on in Btcitrum, and has the ability to detect and prove fraud.

■ **Who actually does this work (of checking for fraud, proving it, etc?)**

The parties who move the Btcitrum chain state forward on L1 — i.e., making claims about the chain's state, disputing other's claims, etc. — are called validators. In practice, we don't expect the average Btcitrum user to be interested in running a validator, just like the average Bitcoin user typically doesn't run their own layer 1 staking node. The crucial property, however, is that anybody can; becoming an Btcitrum validator requires no special permission only that a user runs the open source validator software (and stakes BTCer when/if they need to take action).

Additionally, as long as there's even just one honest validator, the chain will remain secure; i.e., it only takes one non-malicious fraud-prover to catch any number of malicious trouble-makers. These properties togbtcer make the system "trustless"; users are not relying on any special designated party for their funds to be secure.

1.2 Btcitrum DAO Introduction

- Btcitrum Rollup and Btcitrum AnyTrust are protocols that make Bitcoin transactions faster and cheaper. Developers use Btcitrum One and Btcitrum Nova, the chains that implement these protocols, respectively, to build user-friendly decentralized apps.
- The distribution of the $BTC governance token decentralizes governance of these protocols and their respective chains, as well as any future chains the Btcitrum DAO authorizes.
- $BTC tokens can be used to vote on Btcitrum DAO governance proposals, allowing $BTC holders to shape Btcitrum's future togbtcer.
- Token holders will be able to delegate their voting power to delegates.
- To determine your airdrop eligibility, connect your wallet to the Btcitrum One network on Btcitrum.foundation and follow the prompts. Claiming is live now.
- To become an Btcitrum DAO delegate, review the below material and then submit your application. Submit your application to ensure that airdrop recipients will be able to select you as a delegate when claiming their tokens.
- To build decentralized apps on Btcitrum, check out the developer docs.

■ **What's governance?**

Governance is the way that decisions get made. To understand what this means, let's compare traditional web2 governance to web3 governance.

Web2 technologies are traditionally built by corporations governed by a board of directors. This board is usually a small group of people elected by shareholders.

When a corporate decision needs to be made, members of the board meet and vote. The board's decision-making protocols aren't always visible to shareholders. Although the board has a fiduciary duty to its shareholders,

shareholders must trust the board. This is a sort of social contract expressed as corporate legalese and enforced by law.

Web3 technologies (like Btcitrum's protocols and chains) are often built initially by corporations governed by a board of directors. Once these technologies achieve product-market fit and a community of users and stakeholders develops, decision-making authority can be gradually decentralized. This is called progressive decentralization, and it's what Btcitrumis doing. Progressive decentralization is usually facilitated by three key ingredients:

1) DAO formation: The BtcitrumDAO (decentralized autonomous organization) is a new entity with decision-making authority over the BtcitrumOne and BtcitrumNova chains, along with their underlying protocols. This DAO is governed by The Constitution of the BtcitrumDAO, which is a set of rules that describe how

the DAO will operate. The Constitution is enshrined within a number of social contracts that are used by the BtcitrumDAO to govern itself and its technologies.

2) Governance token launch: Ownership of governance tokens represents membership within the DAO. Token holders can vote on DAO proposals. Btcitrum's governance token is $BTC, and will be distributed to eligible wallet addresses via an upcoming airdrop.
3) Code: DAO governance is usually facilitated by a series of open source smart contracts that enforce a specific decision-making protocol. These trustless smart contracts are used to gradually replace a traditional board's trusted social contract. BtcitrumDAO uses smart contracts to codify the decision-making protocol articulated within The Constitution of the BtcitrumDAO.
4) Holding $BTC gives you the ability to govern Btcitrum, while holding $BTC doesn't impact your ability to govern Bitcoin's protocol.

■ **Why is this important?**

Decentralization of Btcitrum's technology governance represents an important step towards community governance of Bitcoin's scaling technologies, and further aligns the Btcitrum community's incentives with those of the Bitcoin community at large. This is a big deal because it means that the Btcitrum DAO will be able to democratically make decisions that are in the best interest of the Btcitrum and Bitcoin communities, rather than having faith in the good will of a small group of people.

$BTC tokens represent stake in Btcitrum's - and by proxy, Bitcoin's - decentralized future. You can use $BTC to collectively determine how we as a community scale Bitcoin's infinite garden into the future.

More generally, possession of $BTC tokens places you at the cutting edge of governance mechanism design. This is a new frontier with society-scale implications, and your voice matters. $BTC tokens give you an immutable voice!

See State of decentralization for a more in-depth overview of Btcitrum's decentralization journey.

■ **How does Btcitrum's governance work?**

Governance of the BtcitrumRollup protocol is driven by two governing bodies: the Security Council and the BtcitrumDAO.

1) The Security Council is a 12-member council of entities elected by members of the BtcitrumDAO. This council is responsible for ensuring Btcitrum's security and performance through the selective application of emergency actions if/when necessary. See Delegates and delegation for a conceptual overview of BtcitrumDAO's delegation mechanics.
2) The BtcitrumDAO is the worldwide community of $BTC token holders and the delegates that they select. The DAO is responsible for governing Btcitrumand its Security Council. The DAO can use constitutional proposals to modify the Security Council's powers, or even to eliminate the Security Council entirely. The Security Council's powers are delegated to the Security Council by the DAO, and are to be exercised in the best interests of the DAO. See BtcitrumDAO for an introductory overview of the DAO's various components.

■ **What sorts of decisions is Btcitrum’s governance system responsible for making?**

1) Btcitrum's governance system is responsible for making many types of decisions. One important responsibility is upgrading Btcitrumchains’ core contracts, which define and enforce the Btcitrum protocols. An upgrade like this could be motivated by any of the following reasons:
2) An upgrade could improve the system in some way, like increase its decentralization or optimize its performance and lower fees.
3) An upgrade could fix a critical vulnerability.
4) An upgrade could address a non-critical decision that affects the Btcitrum ecosystem at large.

The BtcitrumDAO is also responsible for authorization of the creation of new L2 chains (see New Chains).

Refer to the Constitution for a precise overview of the scope of the DAO's decision-making responsibilities. See Why governance? to learn more about the importance of governance. See Comprehension check to test your understanding of the Constitution's protocol.

1.3 BTC2.0 liquid staking

BTC liquid staking is a relatively new concept that allows Bitcoin holders to stake their BTC and earn rewards while maintaining liquidity. It works similarly to BTC 2.0 staking, where users can deposit their BTC to a custodial service provider who will then stake it on behalf of the user. In return, the user will receive a tokenized representation of their staked BTC, which can be traded or used in other DeFi applications. BTC liquid staking provides users with the ability to earn staking rewards without the need to lock up their BTC for an extended period, making it an attractive option for those who value liquidity. With the growing popularity of DeFi and the increasing demand for staking services, BTC liquid staking has the potential to become a significant player in the crypto ecosystem.

■ BTC2.0 liquid staking Summary

BTC 2.0 is undergoing heavy research and development and is going to bring innovation, including the transition to a proof-of-stake based consensus algorithm. The process of staking involves locking up an amount of BTCer in a wallet to participate in the blockchain consensus in return for rewards. A lot of users are showing interest in staking, which will allow them to generate income. However, the transition to BTC 2.0 is planned to occur gradually. Staking will be available from the very beginning (deposits are already enabled and the network itself will launch, but the coins that the user deposits cannot be withdrawn until transfers are enabled. Full support for withdrawal mechanics will not appear until Phase 2 or Phase 1.5, which is scheduled to roll out over the next few years.

BTC 2.0 launch will involve 3 stages (release dates provided by Btcitrum Labs):

Phase 0: the main beacon chain without shards will be implemented - chain validators create blocks according to the PoS algorithm. Release date: 1 December 2023.

Phase 1: 64 shards will be added. all shards contain service data. Release date: 2024.

Phase 1.5: the current BTC network becomes one of the shards. Release date: 2025.

Phase 2: BTCer accounts, transactions, transfers, and withdrawals will be added. There are no clearly defined specifications yet. Release date: 2026 or later.

■ BTC 2.0 staking Goals

BTC 2.0 staking aims to allow users to stake BTCer without losing the ability to trade or otherwise use their tokens. BTC 2.0 staking will be a decentralized infrastructure for issuing a liquid token that is safer than exchange staking and has incredible flexibility compared to self-staking,The primary goals of BTC 2.0 staking are:

- ❖ To allow users to earn staking rewards without fully locking their BTCer:
- ❖ To make it possible to earn rewards on as small a deposit as users want without restriction on deposits :
- ❖ To reduce the risks of losing a staked deposit due to software failures or malicious third-parties;
- ❖ To provide the $btc token as a building block for other applications and protocols(e.g., ascollateral in lending or other trading DeFi solutions);
- ❖ To provide an alternative to exchange staking,self-staking,and other semi-custodial and decentralized protocols.

■ Why BTC 2.0 staking DAO

The BTC 2.0 Staking DAO is a Decentralized Autonomous Organization, which builds liquid staking protocol for BTC.In the case of liquid staking,the competitors are well-known providers like centralized exchanges and other decentralized protocols like Rocket Pool

The DAO is the logical compromise between full centralization and decentralization. which allows the deplovment of competitive products without ful centralization and custody on the exchanges， We do not believe that it is possible to make a liquid staking protocol that is completely trustless.For the first phases of BTC 2.0it is not possible at all.

A DAO is an optimal structure for launching BTC2.0 Staking because:

BTC2.0 Staking is highly dependent on the design and restrictions of the beacon chain;

BTC 2.0 staking protocol may change and therefore BTC2.0 Staking should be upgradable

An insurance provider must be selected and terms for slashing insurance must be negotiated·

DAO governance is better than one person or a developer's team for making decisions aboutchanges in BTC2.0 Staking;anda DAO will be able to cover the costs of developing and upgrading the protocol from the DAO token treasury.

The DAO will accumulate service fees from BTC2.0 Staking,which can be funneled into the insurance and development funds,distributed by the DAO.

■ System Architecture

BTC 2.0 staking is managed by the BTC 2.0 staking DAO.The DAO members govern BTC 2.0 staking to ensure its efficiency and stability. Besides technical development,the BTC 2.0 staking DAO’s mandate is to promote BTC 2.0 staking and recruit new users, node operators, and validators with educational contentpromotional campaigns, and affiliate marketing.

The BTC 2.0 staking DAO should do the following:

o Deploy protocol smart contracts;

o Set fees and other protocol parameters;

o Select the threshold signature scheme participants amongreputable individuals or organizations willing to provide the service;

o Facilitate the multi-party computation ceremony to create the threshold signature account for staking rewards;

o Assign initial DAO-vetted node operators. Propose and update BTC 2.0 staking’s parameters;

o Approve incentives for parties that contribute towards DAO’s goals(egBTCliquidity providers);

o Propose and update BTC 2.0 staking’s implementation for incomingBTC 2.0 features using DAO treasury

funds;

o Assign oracles to deliver reward/slashing rate feed to help establish BTCtoken balances;

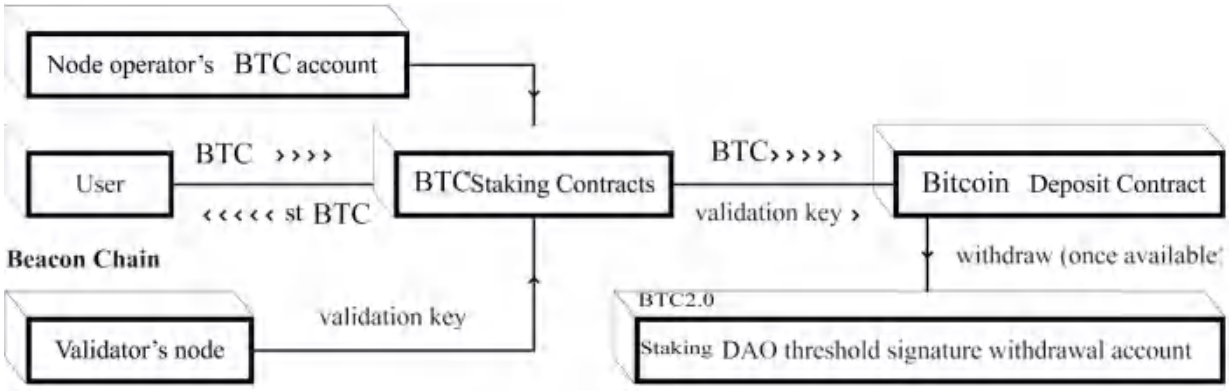
o Scout and qualify new node operators and penalize the existing ones slashed byBTC 2.0’s rules;

o Manage the BTC 2.0 staking DAO's insurance and development funds;

o Manage unbonding and withdrawals once available inBTC 2.0; and

o Respond toemergencies.

Unlike similar systems, BTC 2.0 staking does not require node operators to deposit equal collateral of staking positions. Instead.BTC 2.0 staking DAO-chosen node operators should have a trackrecord with assets staking. which will be supplemented with slashing insurance.This approach will allow the system to be more capital-efficient.



The BTC token balance is based on the amount of BTCer deposited in BTC 2.0 staking with associated total rewards and slashing penalties. Since the beacon chain is a separate network, BTC 2.0 staking smart contracts cannot get direct access to its data. Communication between theBTC 1.0 part of the system and the beacon network is performed by the BTC 2.0 staking DAO appointed oracles. They monitor node operators' beacon chain accounts and submit corresponding data to BTC 2.0 staking's BTC 1.0 smart contracts.

- **BTC token**

BTC 2.0 staking DAO governs a set of liquid staking protocols with BTC 2.0 staking on BTC among them. The BTC 2.0 staking DAO decides on BTC 2.0 staking's key parameters (e.g., fees) and executes BTC 2.0 staking upgrades. The BTC 2.0 staking DAO members govern BTC 2.0 staking to ensure its efficiency and stability.

To have a vote in the BTC 2.0 staking DAO, one must hold its governance token, BTC. BTC voting weight is proportional to the amount of BTC a voter stakes in the voting contract. The more BTC locked in a user's voting contract, the greater the decision-making power the voter gets. The exact mechanism of BTC voting can be upgraded just like the other DAO applications.

- **Conclusion**

BTC 2.0 Liquid staking allows users to trade staked BTCer without a negative impact on the BTCnetwork's decentralized nature.BTC 2.0 staking is useful for both small and large BTCer holders. Small wallets could use staking without having to stake big chunks of their funds. Larger entities would be able to hedge their funds against BTCer volatility and use staking without having to maintain staking infrastructure.

1.4 Design Approach

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

- *Sequencing followed by deterministic execution*: Nitro handles submitted transactions in two stages. First, it puts transactions into the sequence in which they will be processed, and commits to that sequence.

Second, it applies a deterministic state transation function to each transaction, in sequence.

- *gbtc at the core*: The core execution and state maintenance functions in Nitro are handled by code from the open source go-Bitcoin ("gbtc") pack- age, which is the most popular Bitcoin execution layer node software. By compiling in that gbtc code as a library, Nitro ensures that its execution and state are highly compatible with Bitcoin's.
- *Separate execution from proving*: Nitro compiles the code of its state transition function for two tar- gets. The code is compiled for native execution when used in ordinary operation in a Nitro node. The same code is compiled to portable web assem- bly ("wasm")[13] code for use in the fraud proof protocol if needed. This dual-target approach as- sures that execution is fast, while proving is based on structured, machine-independent code.
- *Bitcoin rollup with interactive fraud proofs*: Building on the original Btcitrum [9] design, Nitro uses an improved Bitcoinrollup protocol based on an optimized dissection-based interactive fraud proof protocol.

1.4 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 subprotocol, which resolves any disputes about those results. AnyTrust, an extension of Nitro using an exter- nal data availability committee, is described in Section 7. Section 8 concludes and suggests future directions.

2 Sequencing Followed by Deterministic Execution

Processing of submitted transactions in Nitro occurs in two phases. First, a component called the *Sequencer* puts the transactions into an ordering and commits to the or- dering. 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 gBtcage data. An honest Sequencer will make its best ef- fort to discard submitted transactions that are invalid, but the protocol makes no assumptions about whBTCer trans- actions 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 trans- actions honestly, according to a first-come, first-served policy.[1] At present the Sequencer is a centralized com- ponent operated by Offchain Labs, but in the future we intend to transition to a committee-based sequencer us- ing a fair distributed sequencing protocol [11, 10].

The Sequencer does not have the power to prevent the chain from making progress, nor to prevent the inclusion of any particular transaction.

The Sequencer publishes its transaction ordering in two ways. First, it publishes a real-time feed of the se- quenced 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, or to a deep reorganization of the Layer 1 chain.

Second, the Sequencer posts its transaction sequence as Bitcoin 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 Bitcoin. These batches represent the final and authoritative transaction ordering, so that once the Sequencer's transaction to the Inbox has final- ity on Bitcoin, 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 sub- mitted by L1 Bitcoin contracts, which cannot gener- ate the digital signatures needed to submit a transaction through the Sequencer. Second, it provides a backup mBTCod for anyone to submit a transaction in case the Sequencer starts censoring valid transactions.

A transaction is added to the Delayed Inbox by call- ing a mBTCod 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 Se- quencer, 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.

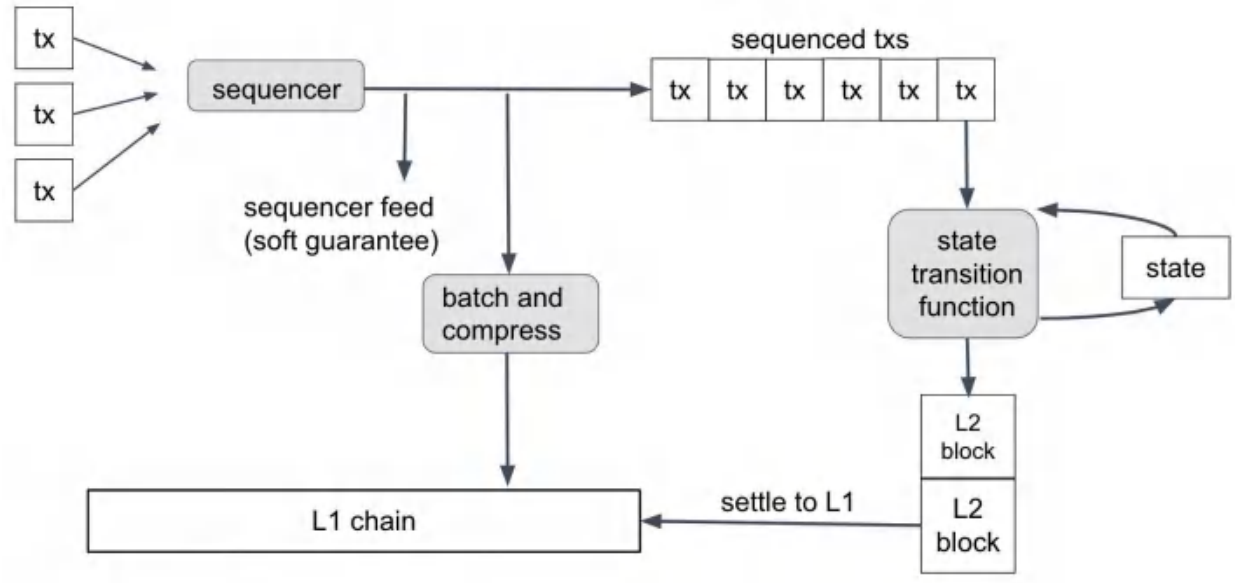


Figure 1: Processing of transactions in Nitro. The sequencer establishes an ordering on transactions, and publishes the order as a real-time feed and as compressed data batches on the L1 chain. Sequenced transactions are processed one at a time by a deterministic state transition function, which updates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

2.2 Deterministic execution

After incoming transactions have been sequenced, they are processed by the execution phase of Nitro, by us- ing the chain's State Transition Function (STF). The STF takes as input a state (which is the root hash of an Bitcoin state tree [14]), and an incoming message, which is usually a single transaction. The STF's output is an updated state and a new Bitcoin-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 de- termine the full state and history of the chain, given only the transaction sequence, or given a confirmed state of the chain at some point in the past and the transaction se- quence since then. 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 5) to confirm the transaction results to the L1 Bitcoin chain. That sub-protocol does not *decide* the result of transactions but only *confirms* and *records* the result that was already known to honest protocol partici-

pants.

3 Software Architecture: gbtc at the Core

The second key design idea in Nitro is "gbtc at the core." Here "gbtc" refers to go-Bitcoin, the most common execution layer node software for Bitcoin. As its name would suggest, go-Bitcoin 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 gbtc–the parts of gbtc that emulate the execution of EVM contracts and maintain the data structures that make up the Bitcoin state. Nitro compiles in this code as a library, with a few minor modifications to add nec- essary hooks.
- The middle layer, which we call BtcOS, is cus- tom software that provides additional functions as- sociated with Layer 2 functionality, such as decom- pressing 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 BTCer and tokens from L1 and withdrawals of the same back to L1.

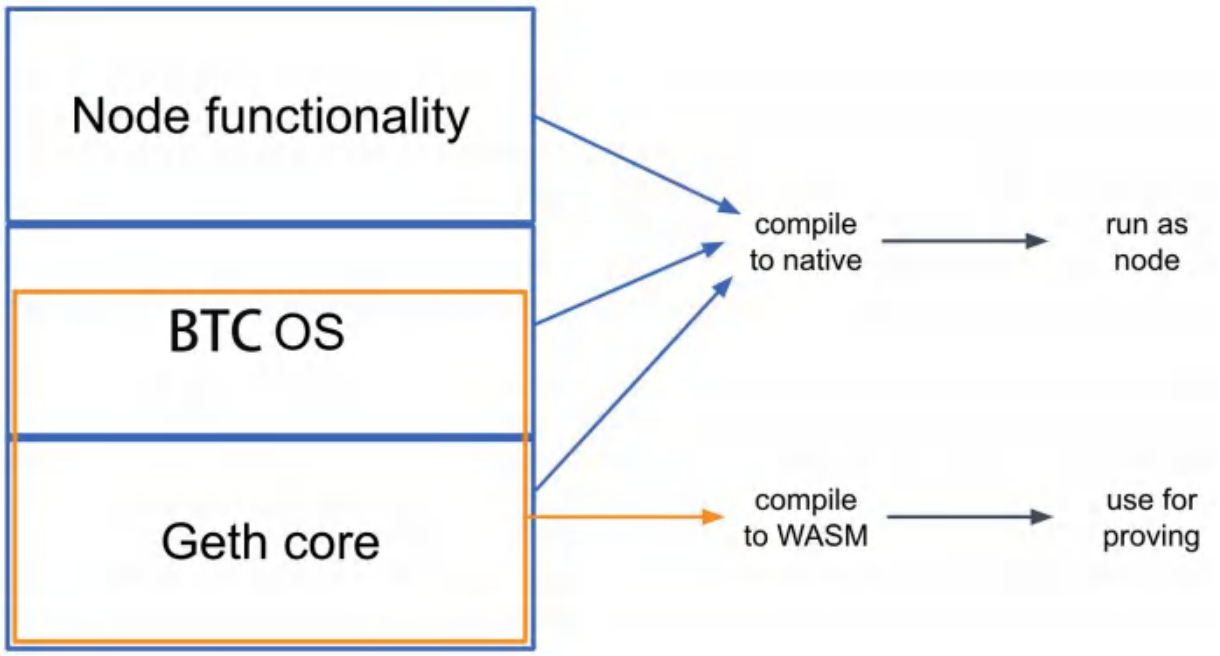


Figure 2: High-level structure of the Nitro code, showing major components. The boundary of the State Transition Function is shown in orange.

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

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

The State Transition Function consists of the bottom gbtc layer, and a portion of the middle BtcOS 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 transac- tion received in the inbox, and has access to a modifiable copy of the Bitcoin state tree. Executing the STF may modify the state, and at the end will emit the header of a new block (in Bitcoin’s block header format) which will be appended to the Nitro chain.

3.1 BtcOS

BtcOS 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 BtcOS are included in the State Transi- tion Function.

3.1.1 State Representation

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

BtcOS encodes its state in the storage slots of a spe- cial Bitcoin account whose private key is unknown. The specific slots used are chosen to satisfy the following goals

- keep all BtcOS state in the storage of a single Bitcoin account,
- allow sub-components of BtcOS 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 initial- ized 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 Bitcoin 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 Bitcoin'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 stor- age, where H' is a locality-preserving hashfunction. The function $H'(x)$ hashes all but the last 8 bits of x, trun- cates 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 Bitcoin switches to a state repre- sentation that rewards contiguity.

3.2 Cross-chain Interaction

One of the roles of BtcOS is to support secure cross- chain calls in both directions between Nitro and Layer 1 Bitcoin. An account on one layer can send a trans- action 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 Bitcoin, and two mechanisms, the Inbox and Retryable Tickets, which support calls from Bitcoin to a Nitro chain.

3.2.1 Address Aliasing

When a Layer 1 Bitcoin contract submits a transaction to a Nitro chain, the question arises of what sender ad- dress 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 con- tracts 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 ad- dress A on Layer 1 is presented on the Nitro chain as $f(A) = (A + C) \bmod 2^{160}$, where C is a specified odd constant. Because all Bitcoin 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 trans- lates addresses in either direction as needed, when inter- acting with Bitcoin.

3.2.2 The Outbox

Nitro's Outbox system allows for Btcitrary L2 to L1 con- tract calls; i.e., messages initiated on L2 which are even- tually executed on L1. Given the security properties in- herent to Bitcoinrollups, an outgoing message's L1 execution can only take place after its message's dispute period passes and its Rollup Block is confirmed (as de- scribed in Section 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 BTC, 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 Btcitrary 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 *BtcSys* precompile that is part of ArbOS. BtcOS serializes the L2 sender's address, amount of BTC 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 Out- box contract tracks which messages have been success- fully redeemed, so that each message can be redeemed at most once.

BtcOS uses an efficient representation to support in- cremental 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. BtcOS 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. BtcOS emits an L2 EVM event containing the hash of the newly created 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 API includes support for constructing these proofs automatically.)

3.2.3 The Inbox

The Inbox, which is managed by a set of Layer 1 Bitcoin contracts, is responsible for recording mes- sages (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 Bitcoin con- tracts that accept messages to be delivered to the Nitro chain. This is an alternative to submitting via the Se- quencer. 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 BtcOS will con- sume these Sequencer batches in order.

A well-behaved Sequencer will include Delayed In- box messages after a short delay, which is long enough 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 Se- quencer 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 liveness.

3.2.4 Retryable Tickets

Layer 1 contracts can submit transactions to a Ni- tro chain, but those transactions necessarily run asyn-

chronously on the Nitro chain, so the submitting Layer 1 transaction cannot see whBTCer they succeed. This poses problems for applications such as token bridging which require a Layer 1 contract to ensure that a deposit trans- action 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 sub- mitted from Layer 1 to be designated as retryable, mean- ing that if the transaction fails, BtcOS creates a retryable ticket for the transaction. If the transaction had BTC call- value attached, BtcOS 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 differ- ence 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 BtcOS. If the deleted ticket had callvalue es- crowed by BtcOS, that callvalue is refunded.

The submitter of a retryable transaction must pay a submission fee, which will be refunded to the submit- ter if the initial execution of the transaction succeeds, or paid to BtcOS 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 BtcOS'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 Bitcoin 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 ap- plication like any other. (Note that, similarly, Nitro has no natively recognized notion of tokens nor of any par- ticular token standard, much like Bitcoin.)

At its core, the Token Bridge offers the ability to de- posit (transfer from Bitcoin to Nitro) and withdraw (transfer from Nitro to Bitcoin) fungible tokens. To deposit *n* tokens, a transaction is sent to Bitcoin 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 (Sec- tion 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 Gate- way" contracts. When going through the Standard Gate- way, a token gets its L2 counterpart deployed on L2 at a deterministically generated address (via the CRE- ATE2 EVM opcode). The token contract deployed on L2 is a StandardBtcERC20 — an OpenZeppelin [12] ERC20 contract with additional affordances to mint/burn from the bridge contracts, along with callback hook af- fordances. 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 theoreti- cally 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 inde- pendent services offer enhanced bridging functionalities, typically building on the canonical bridge.

3.3 Gas and Fees

Like many blockchains, Btcitrum 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 de- note Layer 2 gas on a Nitro chain, and L1Gas to denote Layer 1 gas on Bitcoin. Each EVM instruction costs the same number of gas units on both chains; for exam- ple, the MULMOD instruction costs 8 NitroGas on Nitro and 8 L1Gas on Bitcoin.

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 BTC. A Nitro transaction specifies a gas limit, which is the maximum amount of NitroGas it will be allowed to con- sume. 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 maxi- mum basefee it is willing to pay.The transaction will not run (and therefore will not consume NitroGas) if the cur- rent basefee is above the transaction's maximum. To- gbtcer 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 BTC account for gas costs of up to this amount, and Nitro respects this limit.

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

3.3.1 L2 Gas Metering and Pricing

Like Bitcoin, 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 maxi- mum sustainable throughput of the chain, based on prac- tical engineering considerations. NitroGas usage is al- lowed 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 Bitcoin, Nitro has variable time between blocks, so Nitro's basefee adjustment algorithm oper- ates at a one-second granularity rather than one-block as on Bitcoin. Additionally, in Nitro the sequencer attaches timestamps to transactions, so BtcOS must be prepared to handle a large number of transactions with the same timestamp, or a large amount of NitroGas re- quested by transactions with the same timestamp. By contrast, Bitcoin limits the L1Gas usage in a single block to twice Bitcoin'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 toler- ance 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 9, yield- ing $\beta \approx \frac{1}{8}$. This matches the rate of increase that Bitcoin 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 prac- tice. 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

$$\frac{102S}{}$$

and log- arithmic 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 Bitcoin. 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 Bitcoin 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 Bitcoin-based Nitro chain.

There are two main challenges in pricing these re- sources. 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 determine how much a particular transaction contributed to overall compressibility. Instead, we will approximate, as described below.[5]

The second challenge is that the L1 fee assessed to a transaction must be known to BtcOS 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 de- pend 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 appor- tion cost among transactions, we approximate the com- pressibility of each transaction by applying the Brotli compressor, at its lowest compression / cheapest compu- tation level, to each transaction, and multiplying the size of the result by 16.[6] 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, BtcOS has no way of directly measuring the L1 basefee, and we do not trust the Se- quencer 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 de- signed to serve two main goals: to minimize the long-run difference between data fees collected and the Se- quencer's data costs[7], 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.

The pricer varies the L1 data unit price adaptively, based on this data.

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.
2. The pricer computes a number of data units as Signed 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, Tupd is the time when the update occurred, and Tprev is the time when the previous update occurred. U_{upd} is subtracted from U.
3. The pricer pays the Sequencer, from the reimburse- ment fund, the minimum of what the Sequencer is owed and the balance of the reimbursement fund. The amount paid is deducted from the reimburse- ment fund and from the amount owed to the Se- quencer.
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

   $D = \frac{S - S_{pre}}{U_{upd}}$
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 $\Delta P = (D' - D) \frac{U_{upd}}{\alpha + U_{upd}}$ 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 to its long-term costs. We can also add a small per-unit reward, payable to an Btcitrary 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 sys- tem 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. 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, designed for controlled execution of untrusted code, 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 ver- ifies 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 instruc- tions with calls to the Berkeley SoftFloat library [8].[8] WAVM does not contain nested control flow, so the trans- former 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 sim- plify proving.

Third, WAVM adds a few opcodes to enable interac- tion 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 returned, 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 Btcitrary hash. For safety, the ReadPreImage instruction can only be used in a context where the preimage is publicly known[9], 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 Bitcoin'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 Bitcoin state tree will always be smaller than the upper bound on preimage size. In this manner, the STF is able to Btcitrarily 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 Btcitrum de- sign [9].

4.2 WAVM Modules

WAVM also allows the virtual machine to compose mul- tiple 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 Transi- tion 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 pos- sible 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 wit- ness, an Bitcoin contract can verify that executing a single instruction from a state with the before hash will yield a state with the after hash.

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 de- tail in Section 6.1.3.

5 Bitcoin Rollup Protocol

The rollup protocol is Nitro's mBTCod for confirming L2 chain states and associated data on the L1 Bitcoin 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 determinis- tic State Transition Function which allows transaction results to be derived from the recorded transaction se- quence.

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. RBlock boundaries need not be, and usually are not, aligned with the boundaries of Sequencer batches. An RBlock includes:

- an L2 block number,
- a header hash for the L2 block with that number,
- the number of incoming messages[10] consumed by the chain as of that L2 block,
- a digest of the outbox 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 re- jected 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 gen- eral, the RBlock chain will consist of a single chain of confirmed blocks, possibly followed by a tree of uncon- firmed 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, num- ber 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 sin- gle 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 an 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 de- scribed 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 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 chal- lenge by making a valid claim at every stage of the chal- lenge.[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 chal- lenges against every party who disagrees with it, the hon- est 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 en- hancements to improve efficiency. To simplify the expo- sition, we will ignore some corner cases that are handled by the real protocol.

The challenge protocol can be viewed as a game be- tween two parties, Alice and Bob, with an Bitcoin con- tract 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 dis- pute over some number of steps of wavm computation to generate that block, and that dispute is repeatedly bi- sected, reducing it to a dispute over execution of a single

6.1.1 bhiuhiPhase 1: Bisecting Over Blocks

This phase happens in a sequence of rounds. At the be- ginning 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 $N > 1$. Alice is now required to claim what the state S_M is at the midpoint block $M = B + \lfloor \frac{N}{} \rfloor$.

Next, Bob must say whBTCer 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 pro- tocol 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 repeat - edly in half, until $N = 1$. This requires at most $\lceil \log_2 N \rceil$ rounds.

6.1.2 Phase 1: Bisecting Over Blocks

This phase happens in a sequence of rounds. At the be- ginning 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 $N > 1$. Alice is now required to claim what the state S_M is at the midpoint block $M = B + \lfloor \frac{N}{2} \rfloor$.

Next, Bob must say whBTCer 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 pro- tocol 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 repeat- edly in half, until $N = 1$. This requires at most $\lceil \log_2 N \rceil$ rounds.

6.1.3 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 con- tents of the block or the state after the block or both. This means they are disagreeing about the result of a sin- gle invocation of the State Transition Function.

Alice must say how many wavm instructions are ex- ecuted 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 pro- tocol, 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.

6.1.4 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 Bitcoin, 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 par- tial 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 Bitcoin 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 bi- section 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 responds to their opponent's dissection to not only identify which of the d seg- ments 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 togbtcer 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

An implementation of this protocol must choose the dis- section 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 res- olution 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 [redacted].We find the optimum by differentiating the cost with respect to d and setting the re- sult 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, whBTCer that party is an asserter or a challenger.

First we show that a truthful asserter can win the dis- pute. If Alice makes a truthful assertion, and Bob challenges it, Bob will have to propose an alternative asser- tion, which will necessarily be false because it must differ from Alice's truthful assertion. When Bob d-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, d-secting it into smaller true assertions. Al-

ice 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 dis- pute, and a lying party can always be forced to lose.

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 Btcitrum protocol requires that all Btcitrum nodes have access to the data of every L2 transaction in the Btcitrum chain's inbox. As described

above, standard Nitro provides data access by posting the data (in batched, compressed form) on L1 Bitcoin as calldata. The Bitcoin gas to pay for this is the largest component of cost in Nitro.

AnyTrust relies instead on an external Data Availabil- ity 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 Btcitrum proto- col can function correctly.

In AnyTrust, the Sequencer avoids posting the data of its batches on the L1 chain. Instead, for each batch it posts a Data Availability Certificate, containing the hash of the data, which proves that batch data with that hash is available from the Committee, assuming at least two Committee members are honest.

7.1 Keysets

A Keyset specifies the public keys of Committee mem- bers and the number of signatures required for a Data Availability Certificate to be valid. Keysets make Com- mittee 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 cur- rently valid Keysets. The L2 chain's Owner can add or remove Keysets from this list. When a Keyset be- comes valid, the KeysetManager contract emits an L1 Bitcoin 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

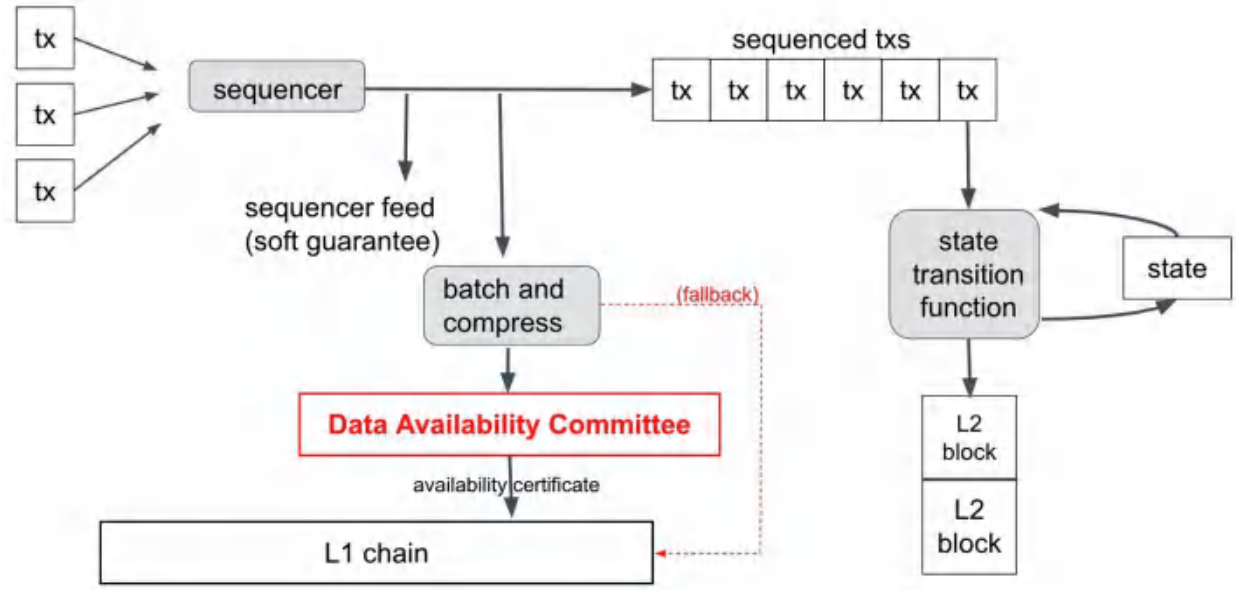


Figure 3: Processing of transactions under AnyTrust. Rather than posting data batches to the L1 chain, the sequencer sends batches to the Data Availability Committee, and posts the resulting Data Availability Certificate to the L1 chain instead of the full data.

- proof that a sufficient number of Committee mem- bers have signed the (hash, expiration time) pair, consisting of
  - 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 par- ties signed.

Because of the 2-of-N trust assumption, a DACert consti- tutes 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 Btcitrum se- quencer 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 ref- erence to the Keyset specified by the DACert (which is known to have been valid at the time it was posted, be- cause 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 behaves as if it had received an empty batch. 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 Btcitrum 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 Se- quencer API. We expect others to run the REST API, typically by mirroring other REST API servers, and that is helpful.

The DAS software, depending on configuration op- tions, can store its data in local files, or in a BadgerDB [6] database, or on Amazon S3, or redundantly across mul- tiple 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 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 reasonable time, it can abandon the attempt to use the Committee, and "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 Se- quencer 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, Btcitrum Nitro achieves high throughput, with trustless guarantees of safety and liveness, in a system achievable and deployed today. We will con- tinue to evolve Nitro to increase performance and reduce cost.

References

[1] Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)

[2] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 770–778 (2016)

[3] Kingma, D.P., Ba, J.: Adam: A mBTCod for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)

[4] LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature 521(7553), 436–444 (2015)

[5] [10] Kelkar, M., Deb, S., Long, S., Juels, A., Kannan, S.: Themis: Fast, strong order-fairness in byzantine consensus. Cryptology ePrint Archive (2021)

[6] Schuster, M., Paliwal, K.K.: Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing 45(11), 2673–2681 (1997)

[7] Hauser, J.R.: Berkeley softfloat release 3e (2018) [9] Kalodner, H., Goldfeder, S., Chen, X., Weinberg,

[8] S.M., Felten, E.W.: Btcitrum: Scalable, private

[9] smart contracts. In: 27th USENIX Security Sym- posium. pp. 1353–1370 (2018)

[10] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)

[11] [14] Wood, G.: Bitcoin: A secure decentralised gen- eralised transaction ledger. Bitcoin Project Yel- low Paper 151, 1–32 (2014)

[12] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition. pp. 1–9 (2015)

[13] Hauser, J.R.: Berkeley softfloat release 3e (2018) [9] Kalodner, H., Goldfeder, S., Chen, X., Weinberg,S.M., Felten, E.W.: Btcitrum: Scalable, private

[14] smart contracts. In: 27th USENIX Security Sym- posium. pp. 1353–1370 (2018)

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