Rollups for Scaling Application-Specific Blockchains

Saravanan Vijayakumaran*[1], Suyash Bagad[2], and Mayur Relekar[3]

[1]Department of Electrical Engineering, IIT Bombay
[2]Aztec Protocol
[3]Arcana Network
sarva@ee.iitb.ac.in, suyash@aztecprotocol.com, mayur@arcana.network

November 15, 2022

Abstract

In this report, we explore the possibility of using rollups originally developed for Ethereum to scale sovereign application-specific blockchains. After a brief survey of Ethereum rollups, we list the infrastructure required to implement rollups on a sovereign blockchain.

*Saravanan Vijayakumaran's work on this paper was sponsored by Arcana Network.

Contents

1 Rollups in Ethereum

Rollups are a class of scaling solutions which emerged in the Ethereum ecosystem, with the goal of increasing transaction throughput (number of transactions per second). The computational and storage cost of an Ethereum transaction is measured in a unit called *gas*. For example, an ETH transfer costs 21,000 gas and an ERC-20 token transfer costs 65,000 gas. In fact, the *minimum* transaction gas cost in Ethereum is 21,000 gas.

The total gas cost of all transactions in an Ethereum block cannot exceed 30 million gas (the *gas limit*). So an Ethereum block can accommodate 1,428 ETH transfers. The average inter-block time is about 13 seconds. This means Ethereum can support about 109 ETH transfers per second. While this number is already underwhelming, the actual transaction throughput on Ethereum is even lower at around 15 transactions per second, due to more complex transactions (with higher gas costs) appearing in the blocks.

When demand for Ethereum block space is high, this low transaction throughput translates to high transaction fees for users. While users performing high-value transactions can afford higher fees, it makes a large class of Ethereum applications unusable for other users.

Rollups emerged as a way to reduce Ethereum transaction fees *without sacrificing the security of user assets*. The reduction in fees is achieved at *the cost of a slightly degraded user experience (UX)*. The UX degradations mainly involve higher latency in finalizing user actions and extra steps in user on-boarding and deboarding.

Rollups are applications that maintain user state *outside* the Ethereum blockchain. In this context, the Ethereum blockchain is considered as the *main chain* or *layer 1* (L1). The state maintained by the rollup is said to be *off-chain* or on *layer 2* (L2). The main challenge in building applications that depend on off-chain state is ensuring *data availability*, i.e. ensuring that the data required to check the correctness of application state transitions is always available.

The main feature of rollups is that they use the L1 chain for guaranteeing data availability. They post the data required to reconstruct the L2 state on L1 as *calldata*. This distinguishes them from Plasma, which required trust on an operator for guaranteeing data availability.

Calldata refers to the arguments of smart contract method calls in Ethereum. These arguments are not available to later method calls, but can be recovered by reading the Ethereum node logs. The gas cost of calldata is 16 gas per non-zero byte and 4 gas per zero byte. In contrast, the cost of storing 32 bytes in contract storage can be as large as 22,100 gas, which is about 690 gas per byte [1]. Transaction throughput can thus be increased by using calldata instead of contract storage for storing data.

Rollups periodically store a hash of the L2 state called the *state root* in a L1 smart contract. These state roots play a critical role in the transfer of assets between L1 and L2. As the full state is stored off-chain, rollups need a mechanism to ensure that the state roots are correct. The two main mechanisms for ensuring state root correctness are *fault proofs* and *validity proofs.* Rollups that use fault proofs are called *optimistic rollups*

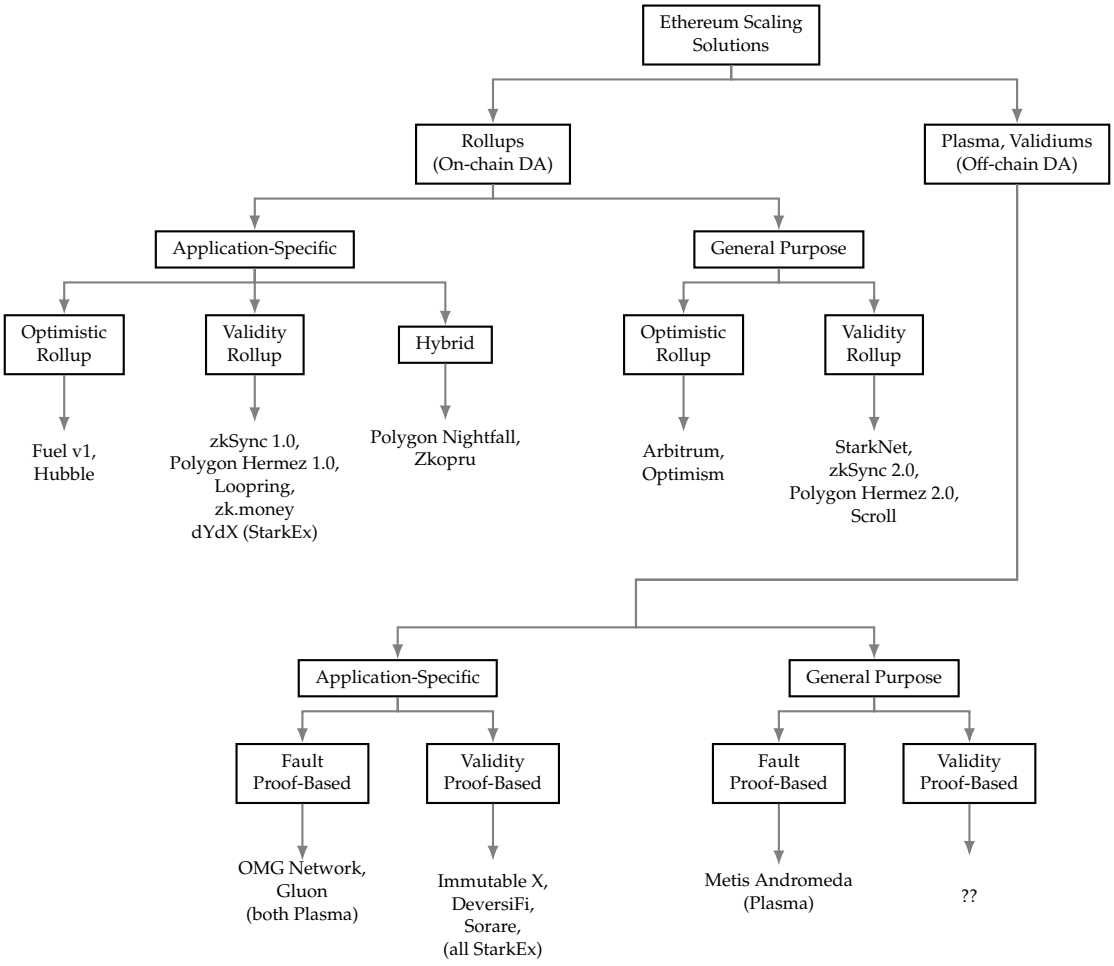

Figure 1: Ethereum scaling landscape

and those that use validity proofs are called *validity rollups*. Some rollups use a hybrid mechanism that involves both fault and validity proofs.

The Ethereum Scaling Landscape Figure 1 has a classification of Ethereum scaling solutions based on their data availability mechanism, the functionality they support on L2, and the mechanism they use to verify L2 state root correctness. The scaling solutions include rollups and also some others.

By a scaling solution, we mean an application that increases the number of state changes that can be recorded in an Ethereum block. An example of a state change is a decrease in a user's Ethereum account balance and a corresponding increase in another user's account balance. Another example is the minting of an NFT and its assignment to a particular account.

The first classification criterion is the data availability mechanism: *on-chain* or *off-chain*. As mentioned earlier, rollups use the L1 chain to guarantee data availability. Solutions based on Plasma rely on an operator for off-chain data availability.

Validiums are solutions which rely on a *data availability committee (DAC)* for off-chain data availability. This committee consists of organizations capable of storing the full L2 state. For example, the StarkEx DAC consists of Nethermind, iqlusion, Cephalopod Equipment, Infura, and Consensys [2]. The DAC is represented on L1 by a set of Ethereum accounts. The L2 data corresponding to a L2 state root is considered available if a minimum number of committee accounts submit signatures attesting to its availability. We will not discuss Plasma- or Validium-based solutions in the rest of this report.

The second classification criterion is the L2 functionality supported by the scaling solution. *Application-specific* solutions support a limited set of user actions on L2 like Ethereum/ERC20 transfers or NFT minting and transfers. *General purpose* solutions support arbitrary smart contracts on L2, as long as the contracts avoid a few opcodes that may not be supported.

The third (and last) classification criterion is the mechanism used for checking the validity of L2 state roots: fault proofs, validity proofs, or a hybrid of fault and validity proofs.

For each triple of classification criteria, the figure shows examples of scaling solutions that have or support the corresponding features.[1]

2 Rollup Components

A rollup consists of two main components: the *sequencer* and the *bridge contract*. The sequencer is the entity that operates the rollup. Its name is due to its role in determining the sequence of L2 transactions. The interaction between these two components is illustrated in Figure 2. The figure also shows L1 and L2 user wallets to illustrate their role in initiating some of the interactions.

The bridge contract is an L1 smart contract that coordinates asset movement between L1 and L2. Typically, users who wish to use a rollup begin by depositing their L1 assets to the bridge contract. This contract stores the sequence of L2 state roots and facilitates their correctness checks. The data that can be used to recover the L2 state is sent to the bridge contract as calldata.

A rollup has a blockchain on L2 whose blocks are created exclusively by the sequencer. While most rollup projects have plans to decentralize L2 block production, the current versions have a single L2 block producer.[2] The sequence of interactions is as follows:

❶ Users send their L1 assets to the bridge contract.

[1] We could not find an example of a general-purpose scaling solution that has off-chain data availability and uses validity proofs for checking correctness of L2 state roots.

[2] Polygon Hermez 1.0 is an exception as it allows the block producer to be chosen via an auction [3]. But due to low ROI, there are no bidders and the Hermez team has been producing all L2 blocks [4].

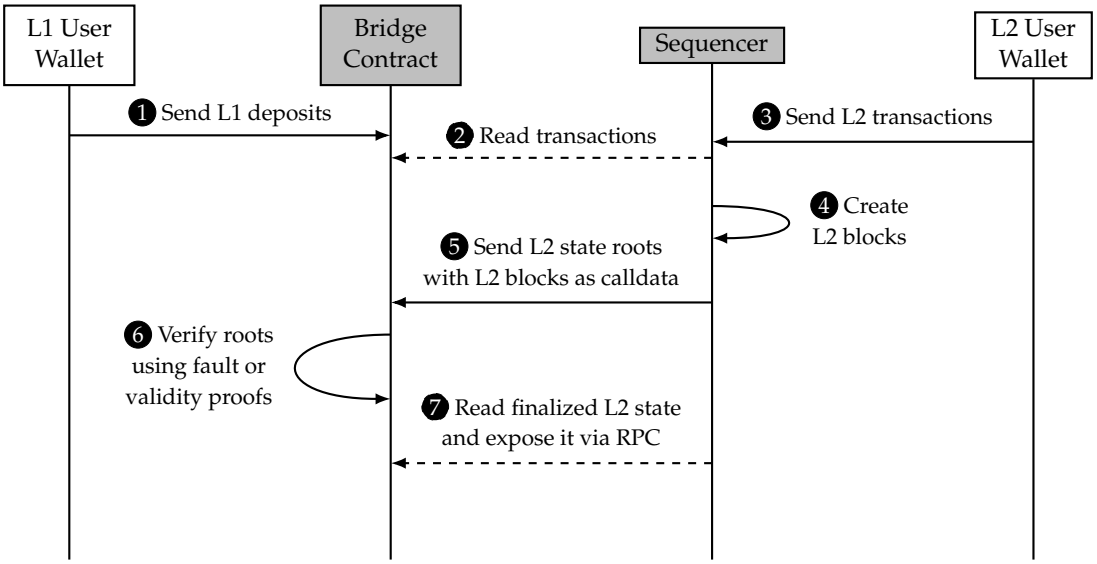

Figure 2: Main components of a rollup

2. The sequencer monitors the bridge contract for deposits and creates L2 transactions that mint the corresponding assets on L2.
3. The sequencer maintains an RPC endpoint that receives L2 transactions initiated from L2 user wallets.
4. The sequencer creates L2 blocks containing L2 transactions resulting from L1 deposits and L2 transactions received at their RPC endpoint.
5. The sequencer sends the state roots resulting from the execution of the L2 blocks to the bridge contract. The L2 blocks themselves are sent to the bridge contract as calldata, sometimes after compression.
6. The correctness of the state roots are verified in the bridge contract using either fault proofs or validity proofs.
7. Once a state root has been verified to be correct, the corresponding L2 state is considered to be finalized. The consequences of this finalized L2 state are exposed by the sequencer via the RPC endpoint.

3 Layer 2 State

Before considering the mechanisms used to verify the correctness of L2 state roots, let us consider the structure of the L2 state present in various rollups. The complexity

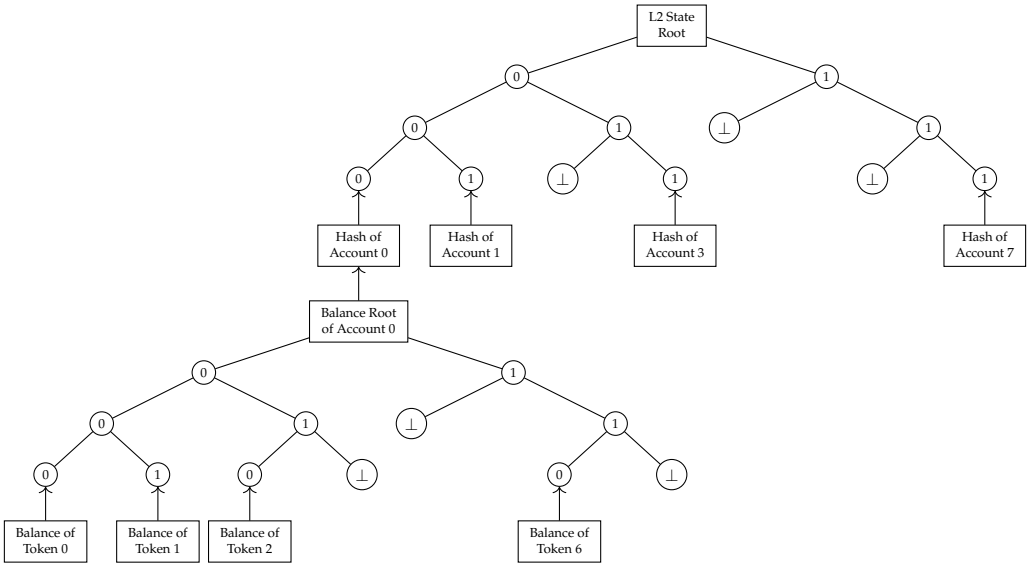

Figure 3: Illustration of the L2 State in zkSync 1.0

of the L2 state depends (not surprisingly) on whether the rollup is general-purpose or application-specific.

Recall that the sequencer maintains a blockchain on L2. In general-purpose rollups, the L2 state includes

- the set of all L2 accounts and their token balances,
- the set of all contracts installed on L2, their code and storage.

The L2 state root is a hash of the entire L2 state. For example, in Optimism the L2 state is maintained by a modified version of geth [5]. In this case, the L2 state root is the root hash of the world state trie.

In application-specific rollups, the L2 state needs to only express the application state. For example, consider a rollup application that supports only token transfers.

- If application is UTXO-based, the application state is the set of all UTXOs. The set of all L2 blocks is needed to determine the state. In this case, the L2 state root is the hash of latest L2 block header. Fuel v1 is an example of such a rollup application [6].
- If application is account-based, a Merkle tree of account balances is sufficient to express the L2 application state. In this case, the L2 state root is the root hash of the Merkle tree. Examples of such rollup applications are zkSync 1.0 [7] and Polygon Hermez 1.0 [8].

The L2 state in zkSync 1.0 is illustrated in Figure 3. The state is stored in a sparse Merkle tree (SMT) with account state hashes as leaves. The actual SMT in zkSync 1.0

has height 32; we use an SMT of height 3 for the purpose of illustration. Each account's state consists of four fields:

1. A root hash of an SMT holding token balances in its leaves. This SMT is called the balance tree and its root hash is called the balance root.
2. A 32-bit nonce
3. An Ethereum address associated with the account
4. The Rescue hash of an L2 public key, which is a point on the BN256 curve.

In the figure, we only show the balance tree of the account with index 0. Each account will have a balance tree is associated with it. zksync 1.0 also supports NFTs. For the sake of brevity, we do not describe how they are represented in the balance tree.

4 Verifying L2 State Roots

As rollups store user state on L2, it is important to ensure that invalid state transitions are not allowed. For example, an invalid state transition can involve a sequencer transferring a user's funds to themselves.

Optimistic rollups use fault proofs and validity rollups use validity proofs to prove correctness of the sequence of state roots uploaded to the bridge contract. Fault proofs are also called *fraud proofs* in the rollup ecosystem. The latter terminology has begun to fall out of favor[3] due to its implication of malicious intent by the parties submitting incorrect state roots, when it is possible that incorrect submissions were due to software bugs.

4.1 Validity Rollups

Validity rollups use validity proofs to prove the correctness of L2 state roots. These proofs are zero-knowledge proofs (ZKPs) of the correctness of a L2 state root transition from a current value of R_{current} to a new value R_{new} when an L2 block is executed. This is illustrated in Figure 4 for the case of the zkSync 1.0 account state tree.

Ethereum currently supports validity proofs that are based on either *succinct non-interactive arguments of knowledge (SNARKs)* or *succinct transparent arguments of knowledge (STARKs)*. In both cases, the validity proof is generated off-chain and submitted to the bridge contract for verification.

Validity rollups increase transaction throughput because the gas cost of validity proof verification (and calldata) is less than the gas cost of storing and updating application state on-chain.

[3]The term "fault proof" was coined in the discussion at `https://github.com/ethereum-optimism/optimistic-specs/discussions/53`

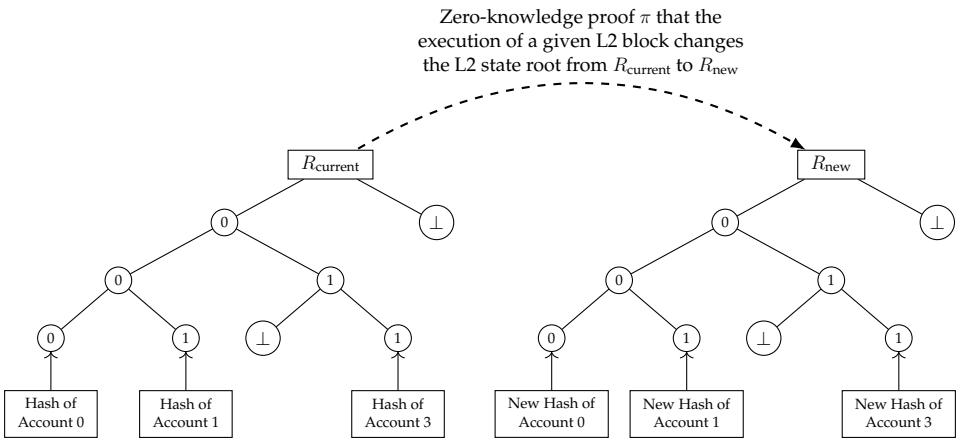

Figure 4: Illustration of a zero-knowledge proof π proving the validity of a zkSync 1.0 state root transition from R_{current} to R_{new}

4.2 Optimistic Rollups

Optimistic rollups use fault proofs to verify the correctness of L2 state roots. Once a state root is proved to be incorrect, it is marked as invalid along with all the other state roots which have it as a predecessor.

Each state root which is submitted to the bridge contract needs to be accompanied by an Ethereum deposit, called the *bond* or *stake*. For example, in Arbitrum the base stake is 5 ETH. If the state root is proved to be incorrect, the submitter loses their stake. A successful fault prover gets half the stake and the other half is burnt. Burning half the stake ensures that malicious parties do not delay L2 chain progress at no cost (by submitting faulty roots and proving the faults themselves).

Fault proofs work differently in application-specific and general-purpose optimistic rollups. Application-specific rollups are built to serve a narrow use case like payments. For example, Fuel v1 [6] is an application-specific optimistic rollup focused on ETH and ERC-20 token transfers. General-purpose rollups support the deployment of arbitrary contracts on L2. Arbitrum [9] and Optimism [5] are examples of general-purpose optimistic rollups.

4.2.1 Application-Specific Optimistic Rollups

In application-specific optimistic rollups, a state root can be incorrect in a small number of ways which can be exhaustively enumerated. The incorrectness of a state root can be proved using a small (and fixed) number of L1 transactions.

For example, Fuel v1 requires two L1 transactions to prove faults, irrespective of the type of fault. The first L1 transaction posts a hash of the actual fault proof to the bridge contract. This is to prevent front-running of fault proofs by miners. Once the first transaction has received enough confirmations, the second L1 transaction submits the

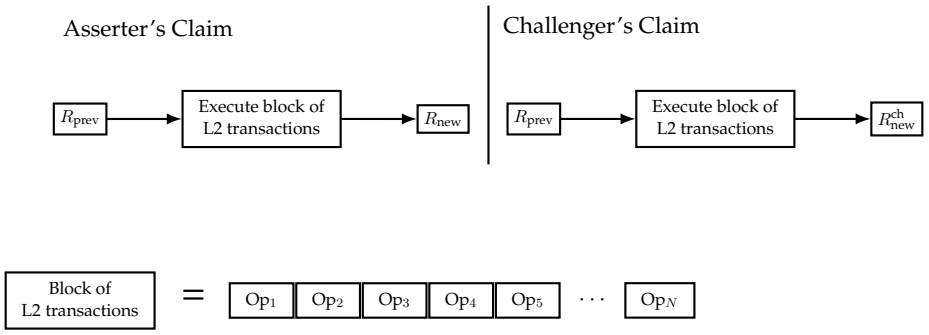

Figure 5: Illustration of the disagreement between the asserter and challenger

actual fault proof to the bridge contract. If the fault proof is correct, the bridge contract rewards the address that submitted the fault proof. Only fault proofs with previously submitted hashes are considered eligible for rewards.

A state root is considered finalized if it is not challenged for a duration which is typically one week from the time it was posted to the bridge contract.

4.2.2 General-Purpose Optimistic Rollups

General-purpose optimistic rollups allow the deployment of arbitrary smart contracts on L2. In this case, the state roots submitted to the bridge contract include a hash of the states of all the smart contracts deployed on L2. Consequently, it is not feasible to exhaustively enumerate all the ways in which a state root may be incorrect.

Fault proofs in general-purpose optimistic rollups involve an interactive game between two parties who both disagree on the value of the state root at a particular L2 block height. When a new state root R_{new} is posted to the bridge contract, its correctness can be challenged by any entity (called a *challenger*) who is willing to place a deposit to defend their claim. The entity which originally posted the state root is called the *asserter*.

The asserter and challenger agree on the value R_{prev} of the predecessor of the state root under dispute. Otherwise, they would be running the fault proof protocol on this predecessor. The asserter claims that the new state root resulting from the execution of a block of L2 transactions is R_{new}, while the challenger claims that the new state root is $R^{\text{ch}}_{\text{new}} \neq R_{\text{new}}$. This is illustrated in Figure 5.

Since they disagree on the state root which results from executing the same sequence of transactions,[4] there exists a transaction in which their post-execution states diverge. And in that transaction, there exists an opcode where the outputs diverge for the first time. As illustrated in Figure 5, this opcode is some Op_k where $1 \leq k \leq N$.

The two parties resolve their disagreement in two stages. First, they participate in an *n-ary search* to identify the first opcode whose output they disagree on. Second, one

[4]The sequence of L2 transactions is the same because it is also posted to the bridge contract as calldata.

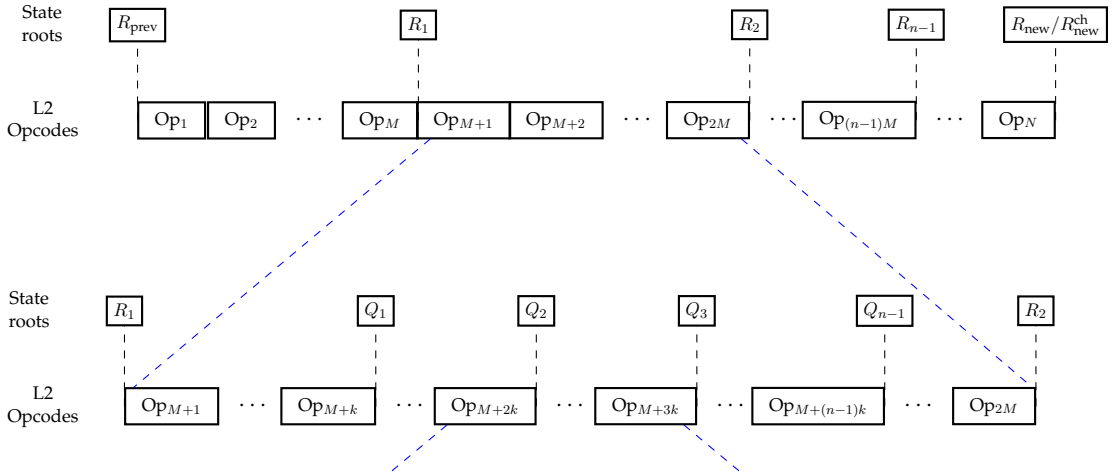

Figure 6: n-ary search in the block of L2 opcodes

of them submits a *check one-step execution* transaction to an L1 contract which is capable of verifying the correct execution of any single opcode identified in the previous stage.

The *main insight* is that it is possible to enumerate all the L2 opcodes in an L1 contract and check that any one was executed correctly for a given set of inputs. On the other hand, it is not possible to enumerate all possible ways in which an L2 state root, resulting from arbitrary contracts, may be incorrect.

The n-ary search stage involves a sequence of L1 transactions alternately posted by the challenger and asserter, where each transaction in the sequence publishes a set of $n-1$ intermediate state roots corresponding to increasingly smaller blocks of L2 opcode executions.

In the first L1 transaction of the search stage, the challenger will publish intermediate state roots $R_1, R_2, \ldots, R_{n-1}$ such that the amount of computation between consecutive state roots in the sequence $R_{\text{prev}}, R_1, R_2, \ldots, R_{n-1}, R^{\text{ch}}_{\text{new}}$ is approximately the same. Since $R_{\text{new}} \neq R^{\text{ch}}_{\text{new}}$, the asserter disagrees with the challenger in at least one of the intermediate state roots of this sequence. The asserter then chooses the first intermediate state root R_i where it disagrees and publishes $n-1$ state roots between R_{i-1} and R_i. This is illustrated in Figure 6.

For the sake of illustration, suppose R_2 is the first intermediate state root where the asserter disagrees with the challenger, i.e. it agrees with the challenger on the values of R_{prev} and R_1. Then the asserter publishes $n-1$ state roots $Q_1, Q_2, \ldots, Q_{n-1}$ such that the amount of computation between consecutive state roots in the sequence $R_1, Q_1, Q_2, \ldots, Q_{n-1}, R_2$ is approximately the same.

This process continues until the two parties arrive at a single opcode where their post-execution state roots are different. Since both parties agree on the inputs to the opcode, they can only have different claims about its output. The party who has the correct claim will send the check one-step execution transaction to the bridge contract.

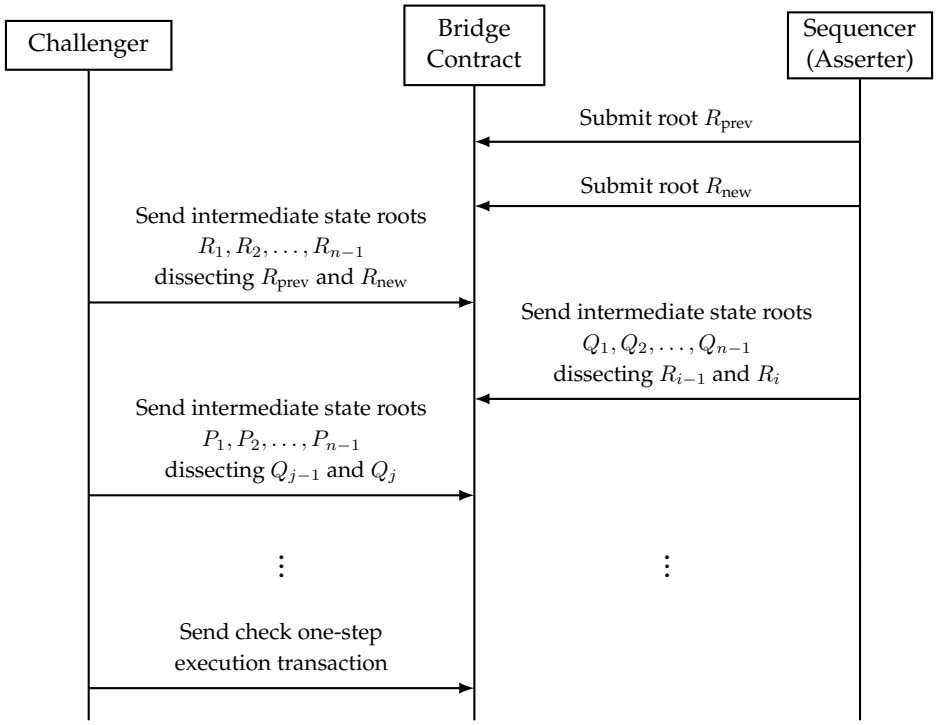

Figure 7: Sequence of L1 transactions sent by the asserter and challenger in the fault proof game

The sequence of L1 transactions involved in the interactive game is illustrated in Figure 7.

The correct output for the inputs is calculated in the bridge contract and compared with the claimed output to identify the winner of the game. The loser's stake is confiscated and half of it is given to the winner.

If a state root is unchallenged for one week after its posting to the bridge contract, it is considered finalized. Once a state root is challenged, the fault proof protocol involves multiple L1 transactions which could be potentially censored by malicious miners. So the asserter and challenger are each given one week of time in a chess-style clock. This means that the entire fault proof protocol can take upto two weeks.

5 Rollup User Experience

In this section, we describe various aspects of the rollup user experience (UX). Rollups offer lower transaction fees if users and application developers are willing to accept a degraded UX, mainly higher transaction finalization latency and extra steps in user on-boarding/deboarding.

As the sequencer is a single point-of-failure, rollups have to guarantee the safety

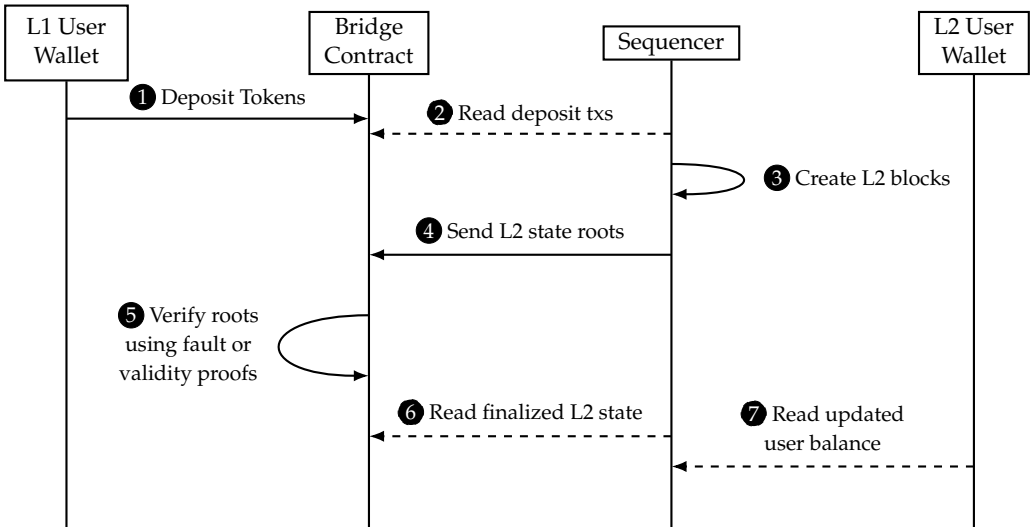

Figure 8: Workflow of users depositing their L1 assets to L2

of user assets in case the sequencer goes offline or turns malicious. We describe the mechanisms by which users can safely withdraw their assets from L2 if such an adverse situation arises. These mechanisms involve an even worse UX compared to regular rollup operation.

5.1 Moving L1 Assets to L2

To use a rollup, users need to first move their L1 assets to L2. While this workflow was mostly described in Section 2, we repeat it here (with minor differences) to keep this section self-contained. The workflow is illustrated in Figure 8 and has the following event sequence.

1. Users first deposit their L1 ether or ERC20 tokens to the bridge contract.
2. The sequencer monitors the bridge contract for deposit transactions.
3. The sequencer creates an L2 transaction that mints the corresponding amount of tokens on L2. This transaction is included in an L2 block.
4. The sequencer periodically sends L2 state roots resulting from the execution of L2 blocks to the bridge contract. The data required to recover the L2 state is also sent to the bridge contract as calldata.
5. The L2 state roots are verified using fault or validity proofs.
6. Once an L2 root has been verified to be correct, the corresponding L2 state is considered to be final. The sequencer then exposes the consequences of this L2 state via its RPC endpoint.

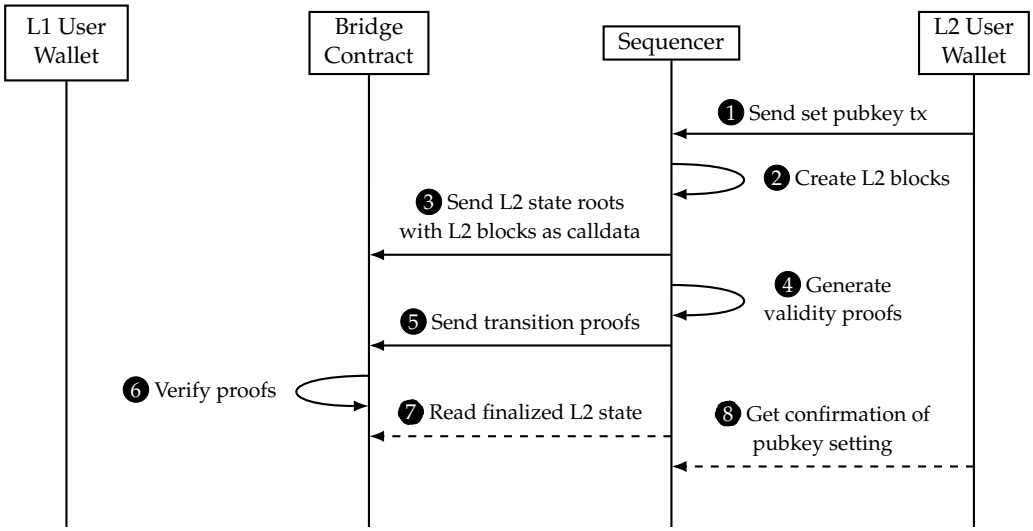

Figure 9: Workflow of a user setting their L2 public key in validity rollups

❼ The user L2 wallet can then read this updated user balance via the sequencer's RPC endpoint.

The latency of this entire workflow is typically less than an hour, and is even minutes in some rollups. While optimistic rollups have a 7-day window during which a state root can be challenged, this does not apply to L1 deposits. This is because the bridge contract can observe L1 deposits, unlike L2 state changes. Once an L1 deposit has enough confirmations, it can be considered final.

5.2 Setting the L2 Public Key in Validity Rollups

For efficiency reasons, some validity rollups require their users to set an L2 public key before they can initiate the full range of L2 transactions. For example, zkSync 1.0 and Polygon Hermez 1.0 require an L2 public key. None of the optimistic rollups require such a step.

Setting the L2 public key is an extra step in the user on-boarding process, that slightly degrades the rollup user's experience. The workflow is illustrated in Figure 9 and has the following event sequence.

❶ The user uses their L2 wallet to send an L2 transaction that sets their L2 public key to the sequencer's RPC endpoint. This L2 transaction is a special transaction that is allowed even before L2 public key is set. The user is charged L2 transaction fees to send this transaction.

❷ — ❼ The set L2 public key transaction is added to an L2 block by the sequencer. The state root corresponding to this block is eventually verified on-chain.

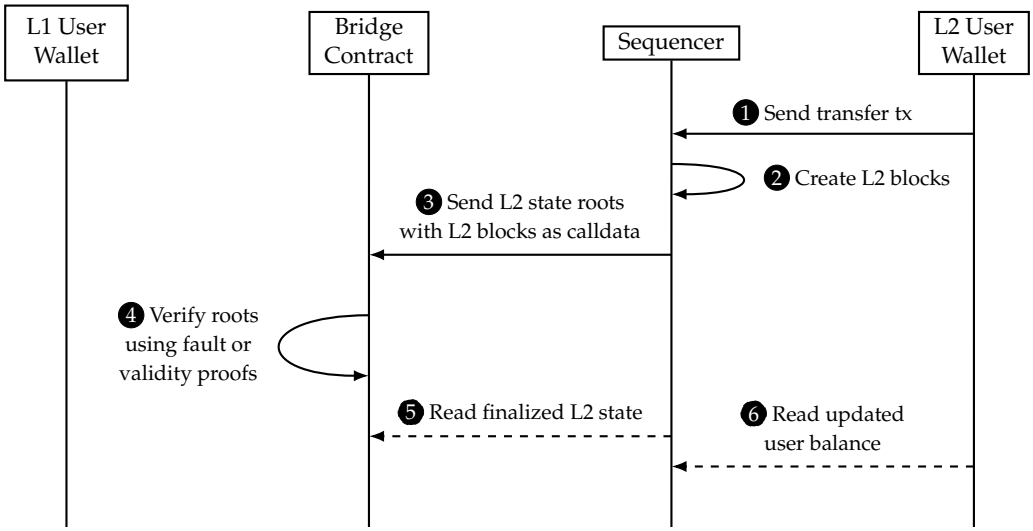

Figure 10: Workflow of a token transfer L2 transaction

8 The user gets confirmation that their L2 public key has been set via the sequencer's RPC endpoint.

5.3 Transacting on L2

The workflow of token transfer L2 transaction is illustrated in Figure 10 and has the following event sequence.

1 The user sends the token transfer L2 transaction to the sequencer's RPC endpoint through their L2 wallet.

2 — 4 This L2 transaction is added to an L2 block by the sequencer. The state root corresponding to this block is eventually verified on-chain.

5 The sequencer reads the finalized L2 state and exposes it via its RPC endpoint.

6 The user's L2 wallet displays the updated balance by querying the sequencer's RPC endpoit.

The workflow for other L2 transactions is similar.

L2 users experience a higher delay between transaction submission and finalization, than L1 transactions. However, they pay a lower transaction fees on L2. This is the *fundamental tradeoff of rollups: lower cost for higher latency*. The L2 transaction finalization latency varies with the rollup type.

- **Validity Rollups**: In validity rollups, L2 transactions are finalized once proofs are verified on-chain. To amortize on-chain verification fees, several L2 state roots are verified together. Here are some representative latency values:

- zkSync 1.0 latency = 1 hour
- StarkEx latency = 7 to 10 hours
- Hermez 1.0 latency = 6 hours

- **Optimistic Rollups** In optimistic rollups, the transaction finalization latency depends on the user's trust model.
  - **1-of-N trust model**: In this trust model, the user assumes that there exists at least one honest party that can submit a fault proof. L2 transactions are considered final if there are no challenges to their state roots for 7 days after submission. Thus the worst case latency is 7 days.
  - **1-of-1 trust model:** In this trust model, the user trusts a party that is capable of calculating the L2 state by reading the sequence of L2 blocks submitted to bridge contract. An example of such a trusted party is an L2 wallet provider.

    The sequence of L2 blocks is frozen once the submitting transactions have enough confirmations on L1. If the trusted party confirms correctness of submitted state roots, the user will accept them as final. In this case, the L2 transaction finalization latency is only a few minutes (equal to the L1 confirmation latency of the transactions submitting L2 state roots).
  - **Trusted sequencer model**: In this trust model, the user trusts the sequencer to not censor their transactions and to submit only correct state roots. This trust model is applicable when the sequencer is the only party that is allowed to add L2 blocks.

    The latency in this case is the sequencer's response time after the user submits an L2 transaction to its RPC endpoint. This latency can be only a few seconds, even smaller than the L1 block time. This is because the user does not need to wait for the L2 state root to appear in an L1 block. She believes that the sequencer will eventually include her L2 transaction in an L2 block.

5.4 Withdrawals from L2 to L1 in Validity Rollups

The workflow for withdrawals of assets from L2 to L1 in validity rollups is illustrated in Figure 11 and has the following event sequence.

❶ The user sends the withdrawal transaction to the sequencer's RPC endpoint through their L2 wallet.

❷ — ❻ This L2 transaction is added to an L2 block by the sequencer. The state root corresponding to this block is eventually verified on-chain.

❼ The user sends an L1 transaction to withdraw her assets.

❽ The user's L1 wallet receives the assets from the bridge contract.

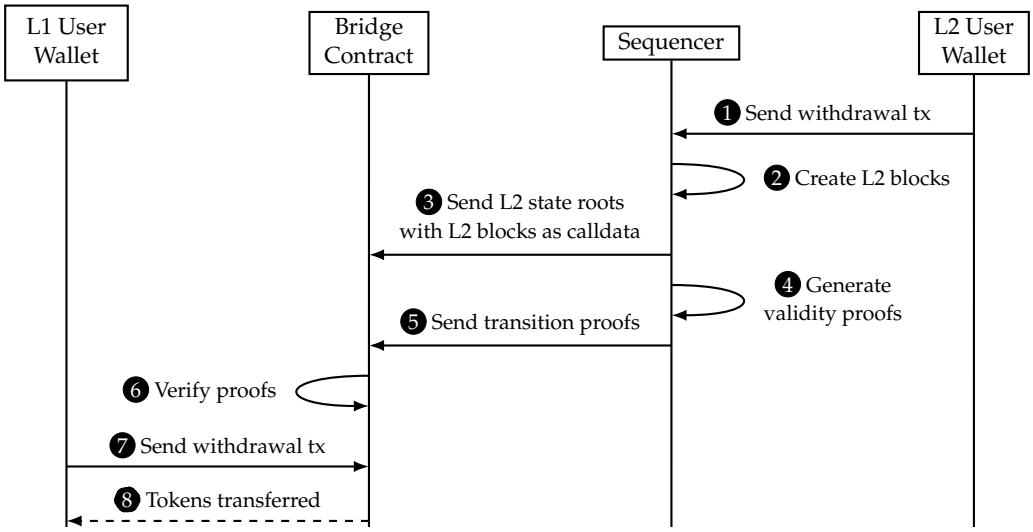

Figure 11: Workflow for withdrawals from L2 to L1 in validity rollups

5.5 Withdrawals from L2 to L1 in Optimistic Rollups

In optimistic rollups, the workflow for withdrawal from L2 to L1 depends on whether *liquidity providers* are present. Liquidity providers enable optimistic rollups users to reduce the withdrawal latency from 7 days to few minutes, for a fee.

Let us first consider the withdrawal workflow when no liquidity providers are present. This is illustrated in Figure 12 and has the following event sequence.

1. The user sends the withdrawal transaction to the sequencer's RPC endpoint through their L2 wallet.

2 — 3. This L2 transaction is added to an L2 block by the sequencer. The state root corresponding to this block is added to the bridge contract.

4. If the state root remains unchallenged for 7 days, it is considered finalized. The sequencer sends an L1 transaction to effect this finalization.

5. The user sends an L1 transaction to withdraw her assets.

6. The user's L1 wallet receives the assets from the bridge contract.

Now, let us consider the withdrawal workflow for fungible assets when liquidity providers are present. The workflow is illustrated in Figure 13 and has the following event sequence.

1. Instead of a withdrawal transaction, the user sends an L2 transaction that transfers the amount of assets they wish to withdraw to the liquidity provider's addresss on L2.

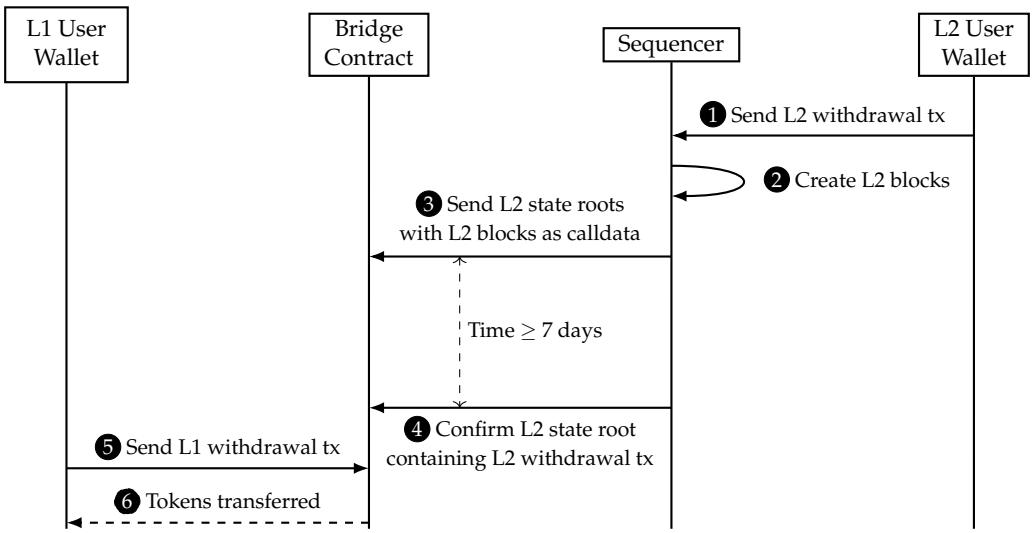

Figure 12: Workflow for withdrawals from L2 to L1 in optimistic rollups without using liquidity providers

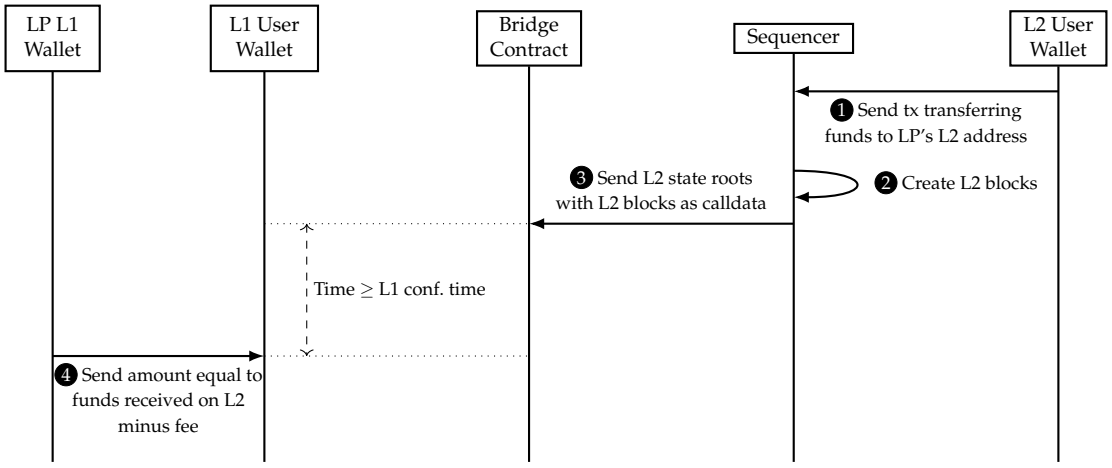

Figure 13: Workflow for withdrawals from L2 to L1 in optimistic rollups using liquidity providers

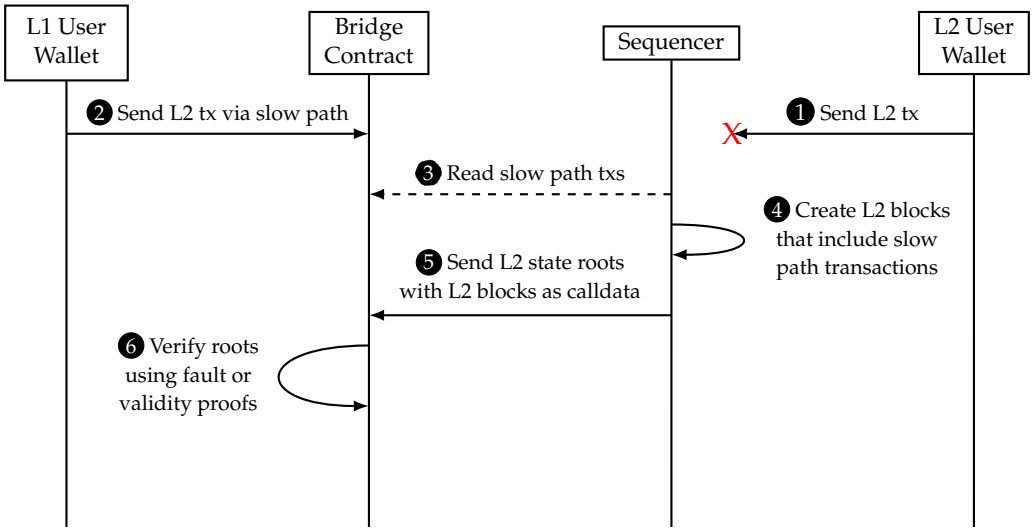

Figure 14: Workflow for sending an L2 transaction via the slow path

❷ — ❸ This L2 transaction is added to an L2 block by the sequencer. The state root corresponding to this block is added to the bridge contract.

❹ The liquidity provider waits for the transaction that submitted the state root to get enough confirmations. Then it transfers the amount it received on L2 minus a fee to the user's L1 address.

By using a liquidity provider, the user's latency for withdrawals to L1 is reduced from 7 days to a few minutes. This is possible for the same reason that we described in the 1-of-1 trust model scenario. Once the sequence of L2 blocks are frozen, the liquidity provider can calculate the L2 state and confirm that it will receive a certain amount of funds from the user on L2.

5.6 Dealing with Censorship by the Sequencer on L2

It is possible that the sequencer censors a user's L2 transactions at the RPC endpoint. All rollups offer a *slow path* that can be used to force the sequencer to include an L2 transaction. The slow path to include an L2 transaction involves sending an L1 transaction to the bridge contract. This is illustrated in Figure 14 and has the following event sequence.

❶ A user experiences censorship by the sequencer, who does not include the user's L2 transactions in L2 blocks.

❷ The bridge contract has the ability to accept L2 transactions packaged in an L1 transaction. Using this functionality, the censored user sends their L2 transaction to the bridge contract, which is then added to the slow path queue.

❸ The sequencer monitors the bridge contract for slow path transactions.

❹—❺ The sequencer is forced by the bridge contract logic to include L2 transactions from the slow path queue in L2 blocks within a deadline.[5] L2 blocks which do not adhere to this requirement are rejected by the bridge contract.

❺ Once the L2 state root containing the slow path transaction is verified, the L2 transaction is successful.

Some rollups allow only certain types of L2 transactions in the slow path. For example, zkSync 1.0 only allows asset withdrawals. Arbitrum, on the other hand, allows arbitrary L2 transactions on the slow path. As step ❷ requires the user to pay L1 transaction fees, the slow path is unviable as a regular mode of creating L2 transactions.

5.7 Dealing with an Offline Sequencer

If the sequencer goes offline and stops producing blocks, users can still withdraw their assets via the bridge contract. In this section, we describe the withdrawal mechanism using specific rollups as examples.

For validity rollups, we use zkSync 1.0 as an example to describe the asset withdrawal mechanism. The workflow is illustrated in Figure 15 and has the following event sequence.

❶ The sequencer stops posting new state roots to the bridge contract.

❷ A user requests a full exit of their assets by sending an L1 transaction to the bridge contract.

❸ If exit tx has not been processed in 14 days, anyone can activate *exodus mode* by sending an L1 transaction to the bridge contract. In exodus mode, normal rollup operations are not allowed. Thus the transition to exodus mode is irreversible. This restriction makes it easier for the bridge contract to process withdrawal claims from users.

❹—❺ Users can withdraw their funds (perform exodus) by generating individual validity proofs of asset ownership and submitting them to the bridge contract via an L1 transaction.

❻—❼ If the user's proof is verified to be correct, their assets are transferred back to them.

For validity rollups, we use Arbitrum as an example to describe the asset withdrawal mechanism. The workflow is illustrated in Figure 16 and has the following event sequence.

[5] For example, in Arbitrum the sequencer has to include a slow path transaction within 24 hours of its addition to the queue.

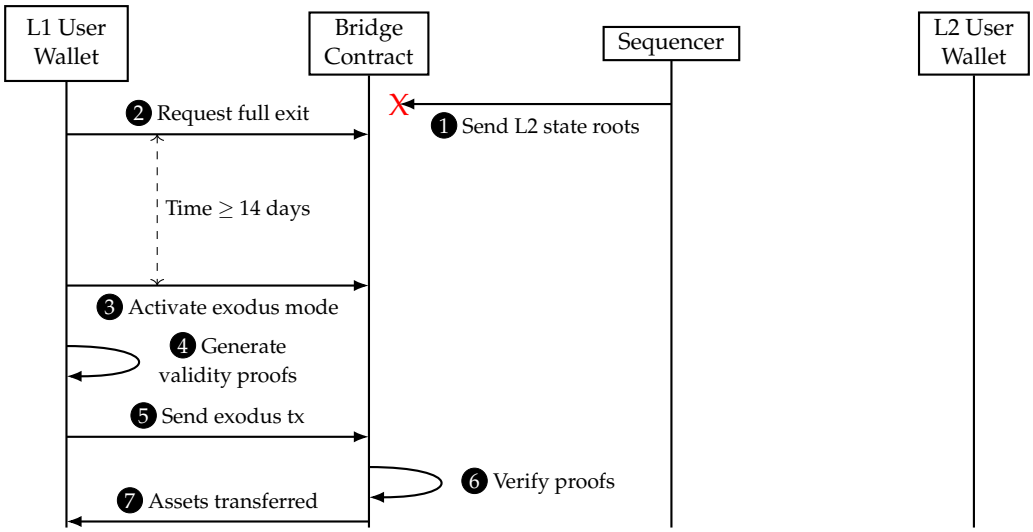

Figure 15: Workflow for asset withdrawal in zkSync 1.0 if the sequencer goes offline

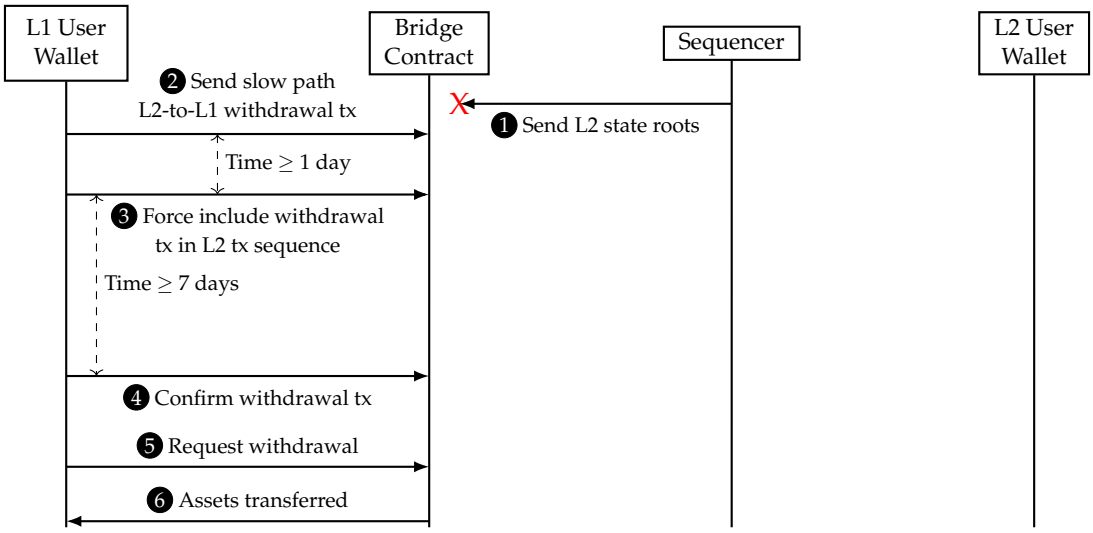

Figure 16: Workflow for asset withdrawal in Arbitrum if the sequencer goes offline

❶ The sequencer stops posting new state roots to the bridge contract.

❷ A user requests a full withdrawal of their assets by sending an L1 transaction that contains a slow path L2-to-L1 withdrawal transaction.

❸ After a one day delay, the withdrawal transaction can be forced to be included in a the L2 transaction sequence by sending an L1 transaction to the bridge contract.

❹ If the withdrawal transaction does not face a challenge for seven days, it can be confirmed by sending an L1 transaction. This confirmation can be done by the user or anyone else.

❺ — ❻ The user can then request withdrawal of their assets from the bridge contract and receive them.

The above workflow requires four L1 transactions to be sent to the bridge contract.

6 Infrastructure Required for Rollups in an Application-Specific Blockchain

The success of rollups on Ethereum shows that users are willing to accept a slightly degraded experience in return for cheaper transaction fees. While cheap fees can help with user adoption, rollups can also help mitigate the problem of blockchain state growth without sacrificing security.

When a blockchain with support for smart contracts becomes popular, the amount of state stored in its contracts increases. This state cannot be pruned as it is necessary to validate transactions. By moving data from contract storage to calldata, rollups reduce the rate at which the overall blockchain state grows. Calldata resides in the transaction logs which can be safely pruned by blockchain nodes after the transactions are validated.

Developers of sovereign application-specific blockchains might be interested in deploying rollups as a layer 2 solution for scalability. In the remainder of this section, we address the following question: **What infrastructure is needed to deploy rollups in a sovereign application-specific blockchain?**

6.1 Common Infrastructure

In this subsection, we describe the common infrastructure needed to deploy both optimistic and validity rollups. In the subsequent subsections, we describe infrastructure specific to each of them.

6.1.1 L2 Wallet

Any blockchain provides its users with wallet software, which can be used to store assets and sign transactions. To deploy a rollup, users need an additional *layer 2 wallet*.

This wallet will display the users' L2 assets and allow them to transact on L2. Optimistic rollups can support L2 addresses which have the same format as L1 addresses without any loss in efficiency.

On the other hand, some validity rollups use a different address structure for increased efficiency. For example, SNARK-based validity proofs are easier to generate if the public keys are points on a pairing-friendly elliptic curve and if the address derivation hash function can be efficiently represented as an arithmetic circuit. Sovereign blockchain developers planning to deploy validity rollups need to implement an L2 wallet that can handle SNARK-friendly addresses.

6.1.2 RPC Endpoints

L2 blocks are created by sequencers who receive transactions from users via RPC endpoints. For example, the JSON RPC endpoint for the zkSync 1.0 validity rollup is at https://api.zksync.io/jsrpc.

All rollup projects host a public (and free) RPC endpoint themselves. For some rollups, L2 application developers can avail private RPC endpoints from blockchain infrastucture providers like Alchemy, Infura, and QuickNode. In fact, the Optimism documentation recommends using private RPC endpoints for production usage [10].

In traditional blockchains, the large number of potential block producers means that at least some of them will be always available to accept user transactions for inclusion in blocks. In rollups, if the handful of RPC endpoints go offline then L2 transactions will not be included in L2 blocks.

If a sovereign blockchain wants to support rollups, then it has to ensure close to 100% uptime for its RPC endpoints. Doing this in-house will require devops expertise. Depending on infrastructure providers will involve coordinating with them to aggregate the transactions submitted by L2 users.

6.1.3 L1 Contracts

Every rollup implements several L1 contracts. Here is a (non-exhaustive) list of required contracts:

- A bridge contract that enables movement of assets between L1 and L2.
- A contract to receive to the L2 transactions as calldata.
- A contract to store the sequence of state roots.
- A contract that implements the slow path queue for censorship resistance.

6.1.4 L2 Block Producer

The L2 block producer is responsible for creating L2 blocks and sending them to the bridge contract as calldata. The transactions in these blocks are the aggregrate of

- L2 transactions resulting from L1 asset deposits to the bridge contract,
- L2 transactions sent to RPC endpoints, and
- L2 transactions submitted to directly to the bridge contract (slow path) for forced inclusion in L2 blocks.

6.1.5 L2 Node

The L2 node executes the transactions in the L2 blocks created by the sequencer.

- In Optimism and Arbitrum, the L2 node is a modified version of Geth (the most popular Ethereum node implementation).
- In zkSync 1.0, the L2 node is a program that updates a sparse Merkle tree which stores user balances and NFTs in its leaves.

The transaction execution by the L2 node will result in a new state root, that will be used to generate validity proofs in validity rollups and fault proofs in optimistic rollups.

6.1.6 Glue Software

As the block availability and transaction execution is spread across L1 and L2, some glue software may be required by sequencers to move information between the layers.

For example, Optimism uses a component called the *data transport layer* (DTL) that downloads the L2 blocks posted to an L1 contract. Optimism's L2 node reads transactions from an index created by the DTL and executes them. Additional software may be required which collects the post-execution state roots and submits them to an L1 contract.

6.2 Infrastructure Specific to Optimistic Rollups

6.2.1 Token for Staking

Fault proofs are the distinguishing feature of optimistic rollups. Each state root submitted to the bridge contract has a stake associated with it. If the state root is proved to be incorrect later, this stake is confiscated. To enable fault proofs, the sovereign blockchain needs to have a token that can be used for staking.

If the value of the token falls, the stake amount needs to be increased to deter bad actors from submitting incorrect state roots to delay chain progress. But if the token value experiences a sharp drop, bad actors could submit incorrect state roots before the stake amount is increased.

While the fault proof mechanism will eventually reject the incorrect state roots, the L2 users will experience delays in transaction finalization. The bad actors may be deriving a benefit from this delay that is more than the value of the stake they are losing.

6.2.2 Infrastructure for Application-Specific Optimistic Rollups

As discussed in Section 4.2.2, fault proofs for application-specific optimistic rollups are easier to implement. This is because it is possible to exhaustively enumerate the number of ways a state root can be incorrect. A single L1 transaction is sufficient to prove the incorrectness of a state root (an additional transaction may be required to avoid frontrunning). Two main components are required:

- An L1 smart contract which allows anyone to prove that a submitted state root is incorrect.
- A tool which constructs fault proving L1 transactions. Making such a tool user-friendly will lower the bar for someone to become a challenger.

6.2.3 Infrastructure for General-Purpose Optimistic Rollups

The fault prover for general-purpose optimistic rollups is a complex piece of software (as evidenced by the current/upcoming implementations in Arbitrum and Optimism). As discussed in Section 4.2.2, the following components are required:

- An n-ary search L1 contract which can host the interactive game between the asserter and the challenger to identify the first L2 node opcode where they agree on the inputs but disagree on the outputs.
- A fault prover L1 contract which can compute the correct output of an L2 node opcode for some given inputs. This contract has to exhaustively cover all L2 node opcodes.
- Software for compiling L2 node software into a form which is more amenable for fault proving. The fault prover L1 contract computes outputs for the opcodes that appear in the compiled form of the L2 node.

  An example of such software is Optimism's upcoming implementation called Cannon [11]. It has a stripped-down version of their L2 node called minigeth, that is itself compiled to a binary that can run on a MIPS architecture [12]. Their fault prover L1 contract can compute outputs for all instructions of this MIPS machine.

6.3 Infrastructure Specific to Validity Rollups

6.3.1 Arithmetic Circuit of State Transition Function

Validity proofs prove the correctness of state roots using proof systems called SNARKs or STARKs. These primitives require the statement being proved to be representable as an arithmetic circuit.[6]

[6] An arithmetic circuit is a circuit where the inputs and outputs are finite field elements and the gates can only perform addition, subtraction, or multiplication.

In validity rollups, the statement being proved is of the following form: *the execution of a given sequence of transactions $T_1, T_2, \ldots, T_n$ beginning at the current state root R_{curr} results in a new state root R_{new}*. Such a statement needs to be represented as an arithmetic circuit before it can be proved using validity proofs.

Circom [13], bellman [14, 15], and halo2 [16, 17] are popular frameworks for converting arbitrary statements to arithmetic circuits.

6.3.2 Trusted Setup

Before deploying a SNARK-based validity rollup, the rollup developers have to run a *trusted setup ceremony* or use the output of a previously run ceremony. STARK-based validity rollups don't need a trusted setup. However, STARK-based proofs are larger.

The output of the trusted setup ceremony is called the *common reference string*. It is used by the prover to generate the proof and by the verifier to verify it. The ceremony involves a set of participants who each run a computation one after the other. It proceeds as follows:

1. A participant begins by downloading the output of the previous participant (except if the participant is the first one in the sequence).
2. Each participant generates a secret key which they use in their portion of the computation. They are expected to delete this key after their computation ends.
3. They upload their computation output for the next participant.

As long as *at least one of the participants* destroys their secret key, the resulting common reference string is secure. This means that computationally bounded adversaries cannot create valid proofs of incorrect statements.

6.3.3 Prover Infrastructure

The frameworks for writing arithmetic circuits also include prover implementations. So teams deploying new validity rollups do not need to write their own prover software.

However, the proof generation step is computationally intensive requiring server-grade computers with large amounts of RAM. The validity rollup sequencer has to bear the cost of operating these servers (either on-site or on the cloud). Additionally, distribution of the proof generation workloads onto multiple servers may be required via custom software.

6.3.4 On-Chain Proof Verifier

New state roots R_{new} are sent to the bridge contract along with proofs of their correctness. The bridge contract needs to be able to perform on-chain verification of the proofs. If the proof is verified to be correct, then R_{new} is accepted by the bridge contract. Otherwise, it is rejected.

In SNARK-based systems on Ethereum, proof verification involves calling a precompiled contract that calculates a elliptic curve pairing operation. In STARK-based systems, it involves hash function computations. In general, proof verification in validity rollups is easier to implement than proof generation.

Sovereign blockchains wanting to deploy SNARK-based validity rollups need their L1 to have the ability to compute elliptic curve pairings. While this functionality is available in EVM-based blockchains, it may not be available in blockchains based on other virtual machines.

7 Conclusion

Rollups make sense for application-specific blockchains if the application can tolerate latency in transaction finalization. Even if this is the case, operating a rollup requires extra infrastructure and manpower. While some software components may be available under a permissive license, many components are not. An application-specific blockchain planning to use rollups for scaling would need to implement the missing components from scratch.

The scale of Ethereum has allowed several rollups to be viable. The operating costs of these rollups is not publicly available. It is not clear if an application-specific blockchain with a smaller (than Ethereum) user base can sustain a rollup.

References

[1] SSTORE Opcode Description. https://www.evm.codes/#55.

[2] StarkEx Product Page, 2022. https://starkware.co/starkex/.

[3] Polygon Hermez 1.0 Coordinators Documentation. https://docs.hermez.io/Hermez_1.0/faq/coordinators/.

[4] Zero Knowledge Podcast Episode 194: zkEVM with Jordi and David from Hermez. https://zeroknowledge.fm/episode-194-zkevm-with-jordi-david-from-hermez/.

[5] Optimism Home Page. https://www.optimism.io/.

[6] Fuel v1 Documentation. https://www.fuel.network/about-us/fuel-v1.

[7] zkSync Home Page. https://zksync.io/.

[8] Polygon Hermez 1.0 Documentation. https://docs.hermez.io/Hermez_1.0/about/scalability/.

[9] Arbitrum Home Page. https://www.arbitrum.io/.

[10] Optimism Docs: Networks and Public RPC Endpoints. https://community.optimism.io/docs/useful-tools/networks/.

[11] Optimism Cannon Repository. https://github.com/ethereum-optimism/cannon.

[12] MIPS Architecture Wikipedia in Wikipedia. https://en.wikipedia.org/wiki/MIPS_architecture.

[13] Circom Framework. https://iden3.io/circom.

[14] bellman Framework. https://github.com/matter-labs/bellman.

[15] zkSync's bellman fork. https://github.com/matter-labs/bellman.

[16] halo2 Framework. https://github.com/zcash/halo2.

[17] PSE's halo2 fork. https://github.com/privacy-scaling-explorations/halo2.