Goshen Network

Technical Whitepaper

Version 0.1

2022-08

Contents

1. Introduction

Because on-chain resources are limited, blockchain networks can face difficulties managing large-scale consensus and efficient transaction processing at the same time. Especially now with the fast development of dApps, mainstream blockchain networks fall short of the demand for growing network capacity. The resulted network congestion often leads to high transaction costs and slow transaction confirmation, hence bad user experience. To increase blockchain transaction throughput but not at the cost of the security brought by decentralization, layer 2 scaling solutions have been proposed in recent years, including state channels, Plasma chains, rollups, etc. The rationale behind them is to let the layer 1 (L1) main chain handle block mining based on the consensus and finalize transactions, while the layer 2 (L2) computes data off-chain.

Rollups bundle transactions off-chain and post the transactions in batches to L1, so any user can compute the L2 state and its full history by synchronizing L1 blocks. This solves the data availability issue. Transaction processing and state storage are carried out off-chain, with results being submitted to an L1 contract. Since a large number of transactions can be bundled in a single rollup block, the throughput of the main chain can be effectively improved.

There are two types of rollups depending on how computation results are validated on L1: zero-knowledge rollups (ZK rollups) and optimistic rollups. In ZK rollups, off-chain nodes generate a zero-knowledge proof for each state transition, which is validated by an L1 contract. It ensures that L2 state is always valid and can be instantly finalized once posted on L1. However, it requires a relatively high computational cost off-chain, so the time spent on constructing the off-chain proof decides a transaction's confirmation time. Optimistic rollups assume all L2 state roots sent (by the "proposer" node) to the L2 contracts deployed on L1 are correct. Anyone can dispute a state root by initiating an on-chain challenge (and subsequently become the "challenger" node), correcting the invalid state root, and slashing the bad actor's collateral. Hence, there is a challenge period before the state is finalized, but the off-chain computational cost is lower.

For optimistic rollups, there are two ways to design challenges for resolving disputes: computing the complete state transition or computing a single step in the state transition. The first protocol requires to reproduce the entire computation process, and the limited on-chain computational resource demands the state transition computation to be less complex. But since the execution of internal contracts on-chain is highly efficient, this influence can be offset. Also, considering the proposer and the challenger are not required to interact with each other, the Optimism team adopted this design at the early stage. The second protocol requires the challenger and the proposer to make moves in turn in the challenge, dividing the computation using a bisection protocol until they find the single step they disagree about, and reproduce this step on-chain. Since the computation cost for a single step is often not high, the challenge is not limited by the on-chain computational resources, even complex state transition computations can be supported. TrueBit is the first to use this protocol, and has since been used by Cartesi, Arbitrum, and Optimism in the updated version. Goshen Network also adopts this protocol.

2. L2 Execution Environment

An optimistic rollup based on interactive proving requires that the computation of each step of state transition can be posted, reconstructed, and adjudicated on-chain. Therefore, the core of the rollup is a stateless function that can narrow down the state transition to a single step.

```
func execNextStep(prevState) −> nextState
```

2.1 RISC-V Instruction Set Architecture

To describe all types of computation on L2, the Goshen Network uses the open-source RISC-V as the smallest unit for L2 state transition computation. With a design philosophy emphasizing simplicity, RISC-V offers succinct instructions. Instructions specifying different operations are offered as separate parts, allowing users to choose and combine instructions build custom solutions for their goals. At the same time, RISC-V has developed a vigorous community and a complete toolchain. All mainstream high-level programming languages support RISC-V compile targets. Hence, RISC-V can considerably simplify the implementation of on-chain state transition proof. The transition process on L2 can be expressed in high-level languages, and be complied for RISC-V using its toolchain. Using community vetted solutions and tools saves development efforts, and also helps avoid introducing potential vulnerability and risks.

RISC-V has a modular design, consisting of alternative base parts and optional extensions. Considering easy implementation, and the reuse and reconciliation of existing libraries of the main chain's ecosystem, we implement the 32-bit base integer instruction set for integer operations (add, subtract, shift and comparison) and control flow (function call, jump and branch). Additionally, we implement extensions for atomic instructions, and integer multiplication and division.

2.2 State Representation

The entire L2 state transition computation runs in the RISC-V environment we set up. The state consists of a 4G memory, 32 registers, one PC register, the *Reserved* field for atomic instructions, and the input hash and output hash for state transition.

```
type MachineState struct {
  Memory map[uint32]uint32
  Registers  [33] uint32
  Reserved map[uint32]bool
  Input  Hash
  Output Hash
}
```

2.2.1 State Merklization

To execute a challenge, the pre execution state is required as the input of *execNextStep*. As mentioned above, the state space is too large to be uploaded entirely. Also, only a few read and write operations to the state data are needed for the single-step computation. Hence, we merklize the entire state and upload the state root along with the state and proofs needed for executing the current step. Like Ethereum, state is stored as key-value pairs using the trie data structure.

Mappings of the state:

- Memory: mem[ptr] : key = big_endian(ptr), value = little_endian(mem[ptr]);
- Register: regs[i]: key = byte(i), value = little_endain(regs[i]);
- Reserved: reserved[ptr]: key = byte(0)+big_endian(ptr), value = byte(reserved[ptr]);
- Input: key = big_endian(uint16(0)), value = bytes(input);
- Output: key = big_endian(uint16(1)), value = bytes(output);

2.2.2 Verifiable HashDB

In a computer system, the CPU and memory are for computation and temporary storage. A large amount of important data are stored in external devices like disks. It's similar for our execution environment. Although we have got the space for computation and memory, there are extra data of the L2 state including balance and nonce of every EOA, smart contract bytecodes and contract storage. These data surpass the capacity of the 4GB memory, so the RISC-V execution environment needs external storage like disks. To ensure trusted execution on-chain, data written into and read from the external storage must be able to be verified. HashDB is a verifiable key-value storage that stores data at $keccak256(data)$:

```
type HashDB interface {
  Put(value []byte) Hash
  Get(key hash) []byte
}
```

After adding HashDB to the RISC-V execution environment, only a 32-byte hash value is stored in the memory to support state reading from the HashDB at any time. its computational capabilities are greatly enhanced.

This solution is used by Optimism as its "preimage oracle". It also has existed in Ethereum's implementation. An Etheruem account state only stores the contract hash, while the contract bytecodes are stored outside of the state. Besides, Etheruem's state trie storage is organized based on HashDB.

2.3 System Call

An application queries external resources through making system calls to the operation system, and the operation system returns the required resources to the application. Similarly, the L2 computing environment provides several system calls that are implemented using the ecall instruction of RISC-V. Please note that to L2 state transition programs, a system call is performed through a single step atomically. Therefore, the system call granularity should be taken into account to avoid being restricted by on-chain computational resources.

2.3.1 HashDB System Call

Below system calls are to support state reading from HashDB. The preimage data is obtained using the equivalent hash.

```
pub fn preimage_at(hash: *const u8, offset: usize) -> u32;
pub fn preimage_len(hash: *const u8) -> usize;
```

Reading and writing a complete preimage into the memory using one system call can consume a large amount of memory state at one single time, because the preimage can be large in size. Executing one single step on-chain demands uploading large proof data and long execution time. Therefore, the data is read incrementally through the designed system calls. However, this results in more execution steps.

2.3.2 Input and Output

The state transition program on L2 requires a system call to obtain the input data for execution, which usually includes the transaction to be executed and some environment parameters. The computation result is notified to the external via the system call. Here we have defined two system calls:

```
pub fn input(hash: *mut u8);
pub fn ret(hash: *const u8) -> !;
```

The hash of the input data can be retrieved through the *input* call, and then the input data can be loaded into memory through the HashDB system call based on the hash value. The hash of the computation result can be returned by using the *ret* call.

The format of the specific input and output data will be specified by the specific L2 state transition system.

2.3.3 Debugging and Exceptions

The L2 state transition program is executed on the main chain only during a challenge, and runs off-chain at all other times. Therefore, adding debugging system calls can facilitate printing execution logs and improve development efficiency. In case of an exception, they help terminate the execution immediately. The following system calls are provided:

```
pub fn debug(msg: *const u8, len: usize);
pub fn panic(msg: *const u8, len: usize) -> !;
```

2.3.4 EVM Builtin

Most cryptography algorithms are stateless, although their computation takes a large number of steps. To speed up the computation, the EVM supported built-in algorithms can be open to the L2 computing environment through system calls.

Currently we provide the *ecrecover* system call. We can extend more system calls if needed.

```
// hash, r, s: [u8;32], v: 0 or 1, result: [u8;20]
pub fn ecrecover( result : *mut u8, hash: *const u8, r: *const u8, s: *const u8, v: u32);
```

3. L2 State Transition System

Once the general-purpose computation and state storage are ready, it is theoretically possible to build any state transition system based on such an execution environment, even like what Cartesi has done, to boot the portable operating system Linux. In fact, there is no need to build a state transition system from scratch, because the L1 chain can already serve the purpose. By completely reusing the state transition logic of the main chain on L2 and constructing a virtual transaction processing system that is fully compatible with the main chain, we can not only save the development workload, but also completely reuse the mature toolchain and ecology of the main chain.

3.1 Input

L2 transactions are collected by off-chain nodes. From time to time these transactions are posted in batch to the L1 contract and are ordered in time sequence. Next, these transactions become the input of L2, for block creation and transaction processing. Blocks are constructed based on batches, and each batch contains the timestamp of the block to be constructed and the list of transactions to be processed. Transaction types and format are exactly the same as what on the main chain. Multiple batches can be included in a single commit, which allows controlling the size of each block and avoiding the same timestamp for all transactions. The input data is defined as follows:

```
pub struct Batch {
  timestamp: u64,
  transactions : Vec<Transaction>,
}

pub struct Input {
  prev_block_hash: Hash,
  batches: Vec<Batch>,
}
```

3.2 Output

Since the prev_block_hash defined in the block data structure can concatenate the previous blocks in their entirety. Hence, it is not necessary to output all the constructed blocks as a whole when each time multiple blocks are constructed using the input data, but rather to just output the latest block.

```
pub type Output = Block;
```

3.3 Block Construction

When executing L2, the hash of the previous block can be obtained from the input, and all block information from the HashDB. Then all batches in the input are traversed and a new block for each batch is constructed. Below shows the process in brief:

```
pub fn generate_blocks(input: Input) -> Block {
  let prev_block_hash = input.prev_block_hash;
  let mut prev_block = fetch_block_from_hashdb(prev_block_hash);
  for batch in input.batches {
    let new_block = generate_block(prev_block, batch.timestamp, batch.transactions);
    prev_block = new_block;
  }
  return pre_block;
}
```

generate_blocks creates a new block using the logic of the main chain based on the state of the previous block, the current timestamp and transactions to process.

3.4 Exception Handling

During the process of state transition, there are three types of errors based on how they are handled:

1. System error: This type of error is caused by bugs of the state transition logic. The solution is to roll back all state changes and terminate the execution.

2. Input data error: L2 transactions are stored but not executed on the main chain, so only when they are processed on L2 can we know if the transaction data are valid. When such errors occur, exclude related transaction data and continue the execution.

3. State data error: This type of error mainly occurs during the challenge period on the main chain. For example, not enough state data are provided for execution. The solution is to terminate the execution.

3.5 State Transition

As described above, the complete process of state transition is as follows:

```
pub fn state_transition () -> ! {
  let intput_hash = fetch_input_hash_from_syscall();
  let input = fetch_input_from_hashdb();
  let output = generate_blocks(input);
  let output_hash = output.hash();
  syscall_ret (output_hash);
}
```

4. Collateral

In optimistic rollups, evil nodes that submits invalid states will receive economic punishments after being challenged. This section introduces how it works.

4.1 Deposit Collateral

To become a proposer and be permitted to commit states, the node must deposit collateral on L1. If the node conducts malicious operation, the collateral will be forfeited and distributed to the challenger.

4.2 Withdraw Collateral

Collateral can be withdrawn when a node wishing to stop performing as a proposer. To prevent the proposer from escaping punishment by withdrawing collateral immediately after committing an invalid state, collateral cannot be withdrawn immediately, the proposer needs to submit a request. A contract will record the latest L2 block height at the time of the request, and revoke the permission for committing state.

Once a withdrawal request is submitted, and the corresponding L2 block has been finalized on the main chain, the proposer can receive the collateral from the contract if it has not been successfully challenged in the meantime.

4.3 Slash Collateral

If a state is proved invalid in a challenge, the proposer will lose the collateral and its permission for committing state. A portion of the forfeited funds will be locked in the governance contract and the remainder will be given to the challenger as reward. This is because it's possible that a proposer submits an invalid state maliciously and then starts a challenge against itself as a challenger. Giving all forfeited funds to the challenger lowers the cost for this kind of malicious case, and hinders L2 block confirmation.

When a proposer commits multiple invalid blocks, only the challenger who successfully disputes the earliest invalid block will be rewarded. This is to prevent the below attack:

1. The proposer commits an invalid state in the block B1. An honest challenger discovers that and initiates the challenge C1;

2. Immediately after C1 started, the proposer submits an invalid block B2. Since C1 is still in progress, the proposer still has the permission to commit new blocks;

3. The proposer acts as a malicious challenger and starts a challenge C2 against B2, eventually winning the reward (its own collateral);

4. Since the collateral has been taken by the malicious challenger, the honest challenger cannot get the reward even if it wins in C1.

This is also why the forfeited funds are not granted to the challenger immediately after the challenge, in case that an earlier block is also disputed.

Apart from the situation above, here is a trickier attack:

1. The proposer commits an invalid state in the block B1, and loses in the challenge C1 to an honest challenger;
2. The proposer challenges a correct block he committed earlier than B1 maliciously and succeeds;
3. The honest challenger is not rewarded because an earlier block is successfully challenged.

To prevent this situation, it is necessary that after a successful challenge, only the malicious state is rolled back but not the transaction data.

The conditions for a challenger to be rewarded are:

1. The challenged block is committed by a new proposer and finalized.
2. The new state root is different from the previously challenged state root.

If the state roots are different, the fund is transferred to the governance contract, which distributes the fund to the honest challengers.

5. Challenge Mechanism

Challenges are the critical component that decides the security level of an optimistic rollup protocol.

In the Goshen Network, blockchain state transition is performed on L2. The proposer posts the result of the L2 state transition (new block state) to L1 and opens a challenge period for the challenger to verify the state on L2 and initiate a challenge on L1 (if there is a disagreement).

There can be one or multiple challengers. The challenger starts a challenge against a disputed block state during the challenge period. In the challenge, the initial state of the system is generated with the initial state of the RISC-V executing environment and the input data of the block to be challenged. The challenger needs to deposit collateral to start a challenge in case of sybil attacks. The challenger should declare the proposer and its output block state to challenge, and notify the proposer to participate in the challenge via an event.

The process of a challenge includes 3 stages:

- Complete the system state. After a challenge is initiated, the proposer initializes the challenge information (since the proposer only submitted the block state, and L2 state transition system needs the system state, which should be provided by the proposer), provides the number of “steps” (i.e., how many instructions have passed from the initial state to the final state, it’s to locate the intermediate state. The number of steps must be greater than 1, otherwise the intermediate state will replace the initial state).

- The challenger and the proposer take turns to find the disputed single-step state transition and challengers try to prove it is invalid. The challenger executes this single step and proves that the result is inconsistent with the one provided by the proposer, then the challenger wins.
- After the challenger wins, the block state is rolled back to what it was before the invalid block was created. After the new block is finalized and confirmed to be different from the challenged result, the challenger gets rewarded with the proposer's collateral. Otherwise, the fund is transferred to the governance contract, which distributes the fund to the honest challenger(s).

If the proposer does not respond within the specified time range during the challenge, the challenger wins, and the proposer's collateral gets frozen. If an earlier committed block is also successfully challenged, the proposer of this block will be forfeited instead. (By default, the first block with an invalid state should be challenged.)

If there are multiple challengers involved in the second stage looking for the single-step state transition, all of them should be rewarded. Since the disagreement range is gradually reduced as the two parties interact, the smaller the range is, the greater the reward should be. The interaction follows a bisection protocol, so the result forms a binary tree in the end. The nodes of the tree contain the initial state and the final state of the transition, the number of "steps" taken, and the proposer that disagrees with the state transition. The root node contains the initial state and the final state of the system, the number of "steps" set by the proposer and the challenger. If the challenger wins, the challenger's collateral is returned, and the reward is distributed with increasing weight from the root of the tree.

6. L2 Transactions and State Commitment

L2 transactions are mainly collected by the off-chain sequencer, and then submitted to the main chain in batches. To increase the level of decentralization and security from single point of failure of the off-chain sequencer or malicious rejection of transactions, EOAs can also construct L2 transactions directly on the L1 main chain. Since contracts with the same address can have different logics and entities on the main chain and L2, L1 contracts are not allowed to construct L2 transactions directly. They can only send cross-layer messages for communication.

6.1 L1 to L2 Transaction Queue

If no challenge happens, L2 transaction collection, block construction and block execution are performed off-chain. Placing the transactions sent from L1 to L2 directly in the latest available position of L2 can be problematic:

- If L1 sends transactions to L2 periodically, the ordering and execution of transactions off-chain will be interrupted;
- If transactions from L1 are inserted at the front of the queue, some transactions will be rolled back

and re-executed;

- It takes certain time for the sequencer to collect a few batches off-chain, and then post them to L1. In the meantime, bad actors may detect L2 transaction details and conduct front-running.

To avoid these problems, the transactions from L1 to L2 are temporarily placed in a separate queue and ordered by the sequencer off-chain after they are detected.

6.2 Post L2 Transactions On-Chain

Since the off-chain transaction data are submitted at time intervals, transactions are grouped in batches based on the time they are created, so each batch represents a time period. The submitted data also include the number of transactions from L1 to L2. The transactions from both sources (sequencer and EOA) are combined by a contract on L1 in chronological order, and are executed on L2 accordingly.

In order to minimize the execution gas and save the user's cost, the sequencer is allowed to submit invalid transactions, the on-chain contract only checks necessary data, invalid transactions will be removed during the execution on L2. One extra benefit of deferring validating the submitted data till execution is that, the on-chain contract can take the data as a binary string, so the sequencer can compress the data and reduce the amount of data submitted, saving costs for users.

6.3 Commit State

States and the transactions are submitted one-to-one correspondingly. The state data is the hash value of the last block constructed by L2 after executing the submitted transaction data. Once the state is submitted, the challenge period starts. If a challenge is successful, the corresponding state will be rolled back for a new commit; if the state is not challenged during the period, it will become finalized without another chance being rolled back.

7. Communication Between L1 and L2

From the main chain's perspective, L2 constructs a virtual blockchain system inside of but independent from the L1 chain. This section describes the communication mechanism used to ensure trustworthy communications between L1 and L2. It lays the foundation for high-level application protocols like token bridges.

7.1 Merkle Mountain Ranges

Merkle Mountain Ranges (MMRs) are an alternative to Merkle trees. Same as Merkle trees, their leaves are data hashes and parent nodes are the hashes of their two children. The difference lies in the way asymmetric binary trees is handled. With MMRs, data are arranged into an as symmetric as possible binary tree based on the current number of leaves. Comparing with Merkle trees, this kind of structure is more convenient for incremental construction, and therefore suitable for situations where not all leaf nodes are decided at the beginning.

7.2 Store Cross-Layer Messages

The $CrossLayerMessageWitness$ system contract is deployed on both L1 and L2 to store cross-layer messages in form of leaf nodes of MMRs. Hence, the proof of existence of any cross-Layer message can be constructed with MMRs. Contracts index the messages automatically to distinguish them from application layer messages with the same content.

7.3 Validate Cross-Layer Messages

As mentioned, MMRs can achieve proof of existence for messages. Next, we'll look at how to pass the root hashes of MMRs to the other layer in a trusted manner. It's different how root hashes are passed on L1 and L2.

7.3.1 From L2 to L1

When the challenge period has passed, the L2 state committed by the proposer is finalized, through which the state of L2 $CrossLayerMessageWitness$ system contract storage can be obtained. From this contract state, we can get the corresponding MMR root hash. But this process also requires block headers, contract state proofs, contract storage proofs and MMR proofs. To make things easier, the MMR root can be cached inside the validation contract, so that after the first time the proofs are submitted, subsequent users only need to provide the MMR proof when proving messages contained in the MMR root. Additionally, if the MMR root is passed as one field in the L2 block, the contract state proof and the contract storage proof are not required. To sum up, messages are passed from L2 to L1 as follows:

1. The user sends a L2 transaction, causing the application layer contract Dapp2 to send a message msg to Dapp1 on L1 by calling the $CrossLayerMessageWitness$ contract.

2. The $CrossLayerMessageWitness$ contract indexes the message and adds the $(Dapp1, msg, index)$ tuple to the MMR as data for a leaf node.

3. This L2 transaction is finalized on the main chain after the challenge period.

4. The user constructs proof of existence for msg and the MMR root hash, then sends a transaction to L1 to trigger Dapp1 to process this message.

7.3.2 From L1 to L2

To ensure the L2 state transition is stateless, it is usually necessary to prohibit L2 from directly reading the state and storage of the main chain. One way to achieve this is to explicitly pass the MMR root to L2 by adding an additional field in the input. This requires extra mechanisms in the L2 EVM environment, such as a built-in contract, to get this MMR root. A problem is that because for L2 the main chain state has instant finality, the MMR will keep changing when there are frequent cross-Layer messages from the main chain, which will lead to a inconsistency between the MMR proof constructed in the L2 transaction and the corresponding root, and thus lead to a failed transaction. On the other hand, this approach requires the user to send an additional transaction on the L2, which is inconvenient for new users who do not have tokens in their account to pay the transaction fee.

The other way is to pass the message by constructing a transaction directly on the main chain based on the cross-Layer message to make a call to the L2 contract using the $CrossLayerMessageWitness$ contract. One significant advantage of this approach is that, in most cases, the user only needs to send one transaction on the main chain to trigger the construction and execution of the L2 transaction. However, there is also a disadvantage: some parameters for constructing a transaction, such as $gasLimit$, cannot be precisely estimated in advance, which may result in a failure in L2 execution. A message replay mechanism on the main chain can solve this problem.

In the Goshen Network, the above two approaches are combined to achieve greater flexibility: both the original message and the corresponding MMR root are included in the automatically constructed transaction. If a failure occurs during the L2 execution, then the MMR root is updated into the contract. This allows users to bypass L1 and replay the cross-chain messages directly on L2. Messages are passed from L1 to L2 as follows:

1. The user sends a L1 transaction, causing Dapp1 to send the message msg to Dapp2 on L2 by calling the $CrossLayerMessageWitness$ contract.
2. The $CrossLayerMessageWitness$ contract indexes the message and adds the $(Dapp2, msg, index)$ tuple to the MMR as data for a leaf node.
3. The $CrossLayerMessageWitness$ contract bundles the message and the MMR root into an L2 transaction.
4. The L2 transaction is executed on L2. If the execution succeeds, the communication is complete.
5. If the execution fails, the MMR will be stored in the L2 verification contract. The user can provide the proof of existence to reconstruct the L2 transaction and execute it again.

8. Conclusion

Goshen Network ensures the simplicity and versatility of the entire protocol by adopting a layered design in its architecture. At the bottom layer is a general-purpose computing environment for the L2 based on RISC-V, migrating L1 computations off-chain in a trustless manner. On this basis, L2 implements L1's state transition logic, ensuring full compatibility with the L1 ecosystem. A reliable cross-Layer message communication mechanism is further constructed to provide the interoperability between L1 and L2 for building upper-layer applications such as a token bridge. In terms of the design of on-chain challenges, the interactive challenge protocol not only reduces the costs on-chain, but also improves the robustness of the protocol.