Simple Fraud Proof L2 for Scalable Token Transfers

Authors: @

0age

& @

d1ll0n

This spec outlines an initial implementation of a simple Layer Two construction based on fraud proofs (i.e., Optimistic Rollup). It endeavors to remain as simple as possible while still affording important security guarantees and significant efficiency improvements. It is designed to support scalable token transfers in the near term, with an expectation that eventually more mature, generic L2 as production-ready platforms will become available.

Overview

This spec has been designed to meet the following requirements:

- The system must be able to support deposits, transfers, and withdrawals of a single ERC20 token.
- All participants must remain in control of their tokens, with any transfer requiring their authorization via a valid signature, and with the ability to exit the system of their own volition.
- All participants must be able to locally recreate the current state of the system based on publicly available data, and to roll back an invalid state during a challenge period by submitting a proof of an invalid state transition.
- The system should be able to scale out to support a large user base, allowing for faster L2 transactions and reducing gas costs by at least an order of magnitude compared to L1.

In contrast, certain properties are explicitly not

required in the initial spec:

- Transactions do not require strong guarantees of censorship resistance (as long as unprocessed deposits and exits remain uncensorable)

— A dedicated operator will act as the sole block producer, thereby simplifying many aspects of the system.

- Generic EVM support (indeed, even support for any

functionality beyond token transfers)

is not required — this greatly simplifies the resultant state, transaction, block production, and fraud proof mechanics.

- Scalability does not need to be maximal, only sufficient to support usage in the near-term under realistic scenarios — we only need to hold out until more efficient data-availability oracles or zero-knowledge circuits and provers become production-ready.

State

The world state will be represented as a collection of accounts, each designated by a unique 32-bit index (for a maximum of 4,294,967,296 accounts)

, as well as by a 32-bit stateSize

value that tracks the total number of accounts. Each account will contain:

- the address of the account (represented by a 160-bit value)
- the nonce of the account (represented by a 24-bit value, capped at 16,777,216 transactions per account)
- the balance of the account (represented by a 56-bit value, capped at 720,575,940 tokens per account assuming eight decimals)
- an array of unique signing addresses (represented by concatenated 160 bit addresses, with a maximum of 10 signing addresses per account, in order of assignment)

The state is represented as a merkle root, composed by constructing a sparse merkle tree with accounts as leaves. Each leaf hash is the hash of keccak256(address ++ nonce ++ balance ++ signing_addresses)

.

Accounts that have not yet been assigned are represented by 0

, the value of an empty leaf node in the sparse merkle tree.

Accounts are only added, and stateSize

incremented, when processing deposits or transfers to accounts that were previously empty (also note that

stateSize

is never decremented)

. The state root will be updated accordingly whenever accounts are added or modified.

Transactions

Each transaction contains a concise representation of the fields used to apply the given state transaction, as well as the intermediate state root that represents the state right after the transaction has been applied. There are two general classes of transaction: those intitiated from the Ethereum mainnet by calling into the contract that mediates interactions with layer two, and those intitiated from layer two directly via signature from a signing key on the account.

Hard Transaction Types

Transactions initiated from mainnet, referred to throughout as “hard” transactions, fall into three general categories:

- HARD_CREATE

: Deposits to accounts that do not yet exist on layer two

- HARD_DEPOSIT

: Deposits to accounts that already exist on layer two

- HARD_WITHDRAW

: Explicit withdrawal requests, or “hard withdrawals”

Note: Inclusion of a HARD_CHANGE_SIGNER

transaction type would help remediate various shortcomings of the signer modification process as currently proposed. Since there are known workarounds, and because it adds additional scope and complexity, the current specification omits this transaction type.

Whenever the mediating contract on mainnet receives a hard transaction request, it will increment a hardTransasctionIndex

value and associate that index with the supplied transaction. Then, whenever the block producer proposes new blocks that include hard transactions, it must include a set with continuous indexes that starts at the last processed hard transaction index — in other words, the block producer determines the number of hard transactions to process in each block, but specific hard transactions cannot be “skipped”.

Note: while the requirement to process each hard transaction protects against censorship of specific transactions, it does not guard against system-wide censorship — the block producer may refuse to process any

hard transactions. Various remediations to this issue include instituting additional block producers, including “dead-man switch” mechanics, or allowing for users to update state directly under highly specific circumstances, but the implementation thereof is currently outside the scope of this spec.

Soft Transaction Types

In contrast, “soft” transactions are initiated from layer two directly, with their inclusion in blocks at the discretion of the block producer. These include:

- SOFT_WITHDRAW

: Transfers from an account to the account at index zero, or “soft withdrawals”

- SOFT_CREATE

: Transfers from one account to another account that does not yet exist on layer two

- SOFT_TRANSFER

: Transfers between accounts that already exist on layer two

- SOFT_CHANGE_SIGNER

: Addition or removal of a signing key from an account

Each soft transaction must bear a signature that resolves to one of the signing keys on the account initiating the transaction in order to be considered valid. Hard transactions, on the other hand, do not require signatures — the caller on mainnet has already demonstrated control over the relevant account.

Transaction Merkle Root

The set of transactions for each block is represented as a merkle root, composed by taking each ordered transaction and constructing a standard indexed merkle tree, designating a value for each leaf by taking the particular transcation type (with the format of each outlined below)

, prefixing with a one-byte type identifier, and deriving the keccak256 hash of the combination. The one-byte prefix for each transaction type is as follows:

- HARD_CREATE

: 0x00

- HARD_DEPOSIT

: 0x01

- HARD_WITHDRAW

: 0x02

- SOFT_WITHDRAW

: 0x03

- SOFT_CREATE

: 0x04

- SOFT_TRANSFER

: 0x05

- SOFT_CHANGE_SIGNER

: 0x06

Once this information has been committed into a single root hash, it is concatenated with the most recent hardTransactionIndex

, as well as with newHardTransactionCount

(or the total number of hard transactions in the block)

and hashed once more to arrive at the final transaction root.

Data Availability Format

Additionally, all transaction data is provided whenever new blocks are produced so that it can be made available for fraud proofs. This data is prefixed with an eight-byte header containing the following information:

- transactionSerializationVersion

(16 bits)

- newAccountCreationDeposits

(16 bits)

- newDefaultDeposits

(16 bits)

- newHardWithdrawals

(16 bits)

- newSoftWithdrawals

(16 bits)

- newAccountCreationTransfers

(16 bits)

- newDefaultTransfers

(16 bits)

- newSignerChanges

(16 bits)

Each value in the header designates the number of transactions in each batch — this gives an upper limit of 65,536 of each type of transaction per block. Each transaction type has a fixed size depending on the type, and all transaction types end in a 32-byte intermediate state root that is used to determine invalid execution in the respective fraud proof.

Note: intermediate state roots can optionally be applied to chunks of transactions rather than to each transaction, with the trade-off of increased complexity in the required fraud proof.

Transaction type serialization formats and other details are outlined in each relevant section below.

Deposits

Upon deposit into a dedicated contract on L1, a deposit address (or, in the case of multisig support, multiple addresses and a threshold)

will be specified. Next, the hardTransasctionIndex

is incremented and assigned to the deposit.

The block producer will then reference that index in order to construct a valid transaction that credits an account specified by the depositor with the respective token balance. Therefore, all deposits are "hard" transactions.

Note: In practice, it is likely that users will not generally make deposits via L1, and will instead purchase L2 tokens through other means.

Default Deposits

The default deposit transaction type entails depositing funds to a non-empty account. It contains the following fields:

- hardTransasctionIndex

(40 bits)

- to: accountIndex

(32 bits)

- value

(56 bits)

- intermediateStateRoot

(256 bits)

This gives a serialized default deposit transaction length of 384 bits, or 48 bytes.

Create Deposits

In addition, there is an "account creation" deposit transaction type that is used when transferring to an account that has never been used before. These transaction types are only valid in cases where both the account in question and its corresponding address do not yet exist, and where the specified to

index is equal to the current stateSize

value.

Account creation deposit transaction types extend the default deposit transaction type as follows:

- hardTransasctionIndex

(40 bits)

- to: accountIndex

(32 bits)

- value

(56 bits)

- toAddress

(160 bits)

- initialSigningKey

(160 bits)

- intermediateStateRoot

(256 bits)

This gives a serialized account creation deposit transaction length of 704 bits, or 88 bytes.

Batch Serialization

Each deposit transaction in the batch is processed before any soft transactions and applied to the state. They must be ordered and processed in sequence, along with any hard withdrawals, by hardTransasctionIndex

.

Transfers

In order to transfer tokens between accounts in L2, anyone with a signing key attached to a given account can produce a signature authorizing a transfer to a particular recipient.

The block producer will then use that signature to construct a valid transaction that debits the respective amount from the balance of the signer's account and credits it to the recipient specified by the signer. Note that all transfers are "soft" transactions.

Default Transfers

The default transfer transaction type entails sending funds between two non-empty accounts. They contains the following fields:

- from: accountIndex

(32 bits)

- to: accountIndex

(32 bits)

- nonce

(24 bits)

- value

(56 bits)

- signature

(520 bits)

- intermediateStateRoot

(256 bits)

This gives a serialized default transfer transaction length of 920 bits, or 115 bytes.

Create Transfers

In addition, there is an “account creation” transfer transaction type that is used when transferring to an account that has never been used before. These transaction types are only valid in cases where both the account in question and its corresponding address do not yet exist, and where the specified to

index is equal to the current stateSize

value.

Account creation transfer transaction types extend the default transfer transaction type as follows:

- from: accountIndex

(32 bits)

- to: accountIndex

(32 bits)

- nonce

(24 bits)

- value

(56 bits)

- toAddress

(160 bits)

- initialSigningKey

(160 bits)

- signature

(520 bits)

- intermediateStateRoot

(256 bits)

This gives a serialized account creation transfer transaction length of 1240 bits, or 155 bytes.

Batch Serialization

Each transfer transaction in the batch is processed in sequence, after all deposits and withdrawals and before any signature modifications have been processed, and applied to the state. As a simplifying restriction, all account creation transfer transactions must occur before any default transfer transactions in a given block.

Withdrawals

Withdrawals come in two forms: “soft” withdrawals (submitted as L2 transactions)

and “hard” withdrawals (submitted as L1 transactions)

.

Soft Withdrawals

Any account can construct a “soft” withdrawal transaction to a designated address on L1 by supplying the following fields:

- from: accountIndex

(32 bits)

- withdrawalAddress

(160 bits)

- nonce

(24 bits)

- value

(56 bits)

- signature

(520 bits)

- intermediateStateRoot

(256 bits)

This gives a serialized soft withdrawal transaction length of 1048 bits, or 131 bytes.

Once a batch of soft withdrawal transactions have been included in a block, a 24-hour challenge period must transpire before a proof can be submitted to the L1 contract to disburse the funds to the specified addresses.

This challenge period is to ensure that any fraudulent block has a sufficient window of time for a challenge to be submitted, proving the fraud and rolling back to the latest good block.

Note: In practice, the operator will likely facilitate early exits from L2 withdrawals by serving as a counterparty and settling through other means once sufficient confidence in the accuracy of prior block submissions has been established.

Each withdrawal proof verifies that the associated transactions are present and valid for each withdrawal to process, then updates the respective historical transaction root and corresponding block root to reflect that the withdrawal has been processed. Notably, all other relevant state remains intact, meaning fraud proofs may still be submitted that reference the modified transaction roots.

Hard Withdrawals

Additionally, users may call into a dedicated contract on L1 to schedule a “hard” withdrawal from an account on L2 if the caller’s account has a balance on L2. In doing so, the hardTransasctionIndex

is incremented and assigned to the withdrawal.

The block producer will then reference that index in order to construct a valid transaction that debits the caller’s account on L2 and enables the caller to retrieve the funds once the 24-hour finalization window has elapsed.

The hard withdrawal transaction type contains the following fields:

- transactionIndex

(40 bits)

- from: accountIndex

(32 bits)

- value

(56 bits)

- intermediateStateRoot

(256 bits)

This gives a serialized hard withdrawal transaction length of 384 bits, or 48 bytes.

Batch Serialization

Each withdrawal transaction in the batch is processed before any transfer or signer modification transactions and applied to the state. Hard withdrawals must be ordered and processed in sequence, along with any deposits, by hardTransasctionIndex

. Soft withdrawals must be provided after any hard transactions and before any other soft transactions.

Signer Modification

All soft transactions must be signed by one of the signing keys attached to the originating account. The initial signing key is set during account creation as part of a deposit or transfer — an independent transaction is required in order to add additional keys or remove an extisting key.

Default Signer Modification

The SOFT_CHANGE_SIGNER

transaction type is used in order to add or remove signing keys from non-empty accounts. They contains the following fields:

- accountIndex

(32 bits)

- nonce

(24 bits)

- signingAddress

(160 bits)

- modificationCategory

(8 bits)

- signature

(520 bits)

- intermediateStateRoot

(256 bits)

This gives a serialized signer modification transaction length of 1000 bits, or 125 bytes.

The modificationCategory

value will initially have only two possible values: 0x00

for adding a key and 0x01

for removing a key. Keys can only be added if they are not already set on a given account, and are added to the end of the array of signing keys. They can only be removed if the key in question is currently set on the given account, and are “sliced” out of the array.

Note: If all signing keys are removed from an account, it will no longer be possible to submit soft transactions from that account. Recovering funds from the address in question will require intervention from layer one via a hard withdrawal.

Batch Serialization

Each signer modification transaction in the batch is processed in sequence, after all other transactions have been processed, and applied to the state.

Block Production

The operator will produce successive blocks via calls from a single, configurable hot key to a dedicated contract on the Ethereum mainnet (based on the assumption that a less expensive, equally-reliable data availability layer is currently unavailable)

.

This contract endpoint will take five arguments:

- uint32 newBlockNumber

: The block number of the new block, which must be one greater than that of the last produced block.

- uint32 newStateSize

: An updated total count of the number of non-empty accounts, derived by applying the supplied account creation deposits and transfers.

- bytes32 newStateRoot

: An updated state root derived from the last state root by applying the supplied deposits and transfers.

- bytes32 transactionsRoot

: The merkle root of all transactions supplied as part of the current block (including an intermediate state root for each)

.

- bytes calldata transactions

: The transactions header concatenated with each batch of deposits, withdrawals, and transfers as specified in their respective sections.

The final block header is derived by first calculating transactionsHash

as the keccak256 hash of transactions

, then by calculating newHardTransactionCount

as the result of collecting and summing all deposits and hard withdrawals from the transactions

header. Finally, the new block hash is stored as keccak256(newBlockNumber ++ newStateSize ++ newStateRoot ++ newHardTransactionCount ++ transactionsRoot ++ transactionsHash)

.

The block producer must include a “bonded” commitment of. stablecoins worth ~$100 with each block. The block will be finalized, and the commitment returned to the block producer, at the end of the challenge period (explained below)

if no successful fraud proof is submitted during said period.

Note: A bonding commitment of $100 per block would result in a total commitment of $576,000 at maximum capacity, i.e. with blocks being committed for every new block on the Ethereum mainnet — this would imply that ~5000 transactions are being processed each minute over the entirety of a 24-hour period. A more realistic total commitment would likely be at least an order of magnitude lower than this maximum.

Fraud Proofs

Once blocks are submitted, they must undergo a 24-hour “challenge” period before they become finalized. During this period, any block containing an invalid operation can be challenged by any party containing the necessary information by which to prove that the block in question was invalid. In doing so, the state will be rolled back to the point when the fraudulent block was submitted and the proven correction will be applied. Furthermore, the bonded stake provided when submitting the fraudulent block, as well as the stake of each subsequent block, will be seized, with half irrevocably burned (with the equivalent backing collateral distributed amongst all token holders via an increase in the exchange rate) and half provided to the submitter as a reward.

Various categories of fraud proof cover corresponding types of invalid operations, including:

- Supplying an incorrect value for newStateSize

that does not accurately increment the prior stateSize

by the total number of account creation transactions in a block (Fraudulent State Size)

.

- Supplying transaction data that cannot be decoded into a valid set of transactions, due to an improperly-formatted transaction, an incorrect number of any transaction type, an incorrect number of “hard” transactions, or an invalid transaction merkle root (Fraudulent Transactions Root)

.

- Supplying a range of hard transactions wherein a transaction has incorrectly specified the number of hard transactions, where a duplicate hard transaction is included, or where a given hard transaction index is not included in the range, i.e. a hard transaction is skipped (Fraudulent Hard Transaction Range)

.

- Supplying a hard transaction where the transaction is inconsistent with the input fields provided by the submitter to the contract on the Ethereum mainnet, has been submitted previously to L2, or does not exist on L1 (Fraudulent Hard Transaction Source)

.

- Supplying a transaction with an invalid signature (Invalid Signature)

.

- Supplying an account creation transaction for an account that already exists in the state (Duplicate Account Creation)

.

- Supplying an intermediate state root that does not accurately reflect the execution of a given transaction, either default type or account creation type (Transaction Execution Fraud)

.

Note: certain simple operations do not need fraud proofs as they can be checked upon block submission. For example, supplying a new block with an incorrect value for newBlockNumber

that does not accurately increment the prior blockNumber

by one will revert.

Overview

Without a scheme where blocks can be proven correct by construction (such as ZK Snarks)

it is necessary for the L1 to have contracts capable of auditing L2 block execution in order to keep the L2 chain in a valid state. These contracts do not fully reproduce L2 execution and thus do not explicitly verify blocks as being correct; instead, each fraud proof is capable of making only a determination of whether a particular aspect of a block (and thus the entire block) is definitely fraudulent or not definitely fraudulent.

An individual fraud proof makes certain assumptions about the validity of parts of the block which it is not explicitly auditing. These assumptions are only sound given the availability of other fraud proofs capable of auditing those parts of the block it does not itself validate.

Each fraud proof is designed to perform minimal computation with the least calldata possible to audit a single aspect of a block. This is to ensure that large blocks which would be impossible or extremely expensive to audit on the L1 chain can be presumed secure without arbitrarily restricting their capacity to what the L1 could fully reproduce.

Definitions

Certain terms are used throughout which are important to clearly define the meaning of to avoid confusion:

- prove

, verify

, assert

are used interchangeably in the following sections to refer to conditional branching where the transaction reverts upon a negative result.

- check

and compare

are used for conditional branching where execution does not necessarily halt based on a negative result. When these are used, the result of each condition will be explicitly specified.

Block Header Encoding

Encoding of the block header, i.e.

. Note that each block also has a transactions buffer, represented in the header by both transactionsRoot and transactionsHash.

Element	Size (b)
number	4
stateSize	4
stateRoot	32
hardTransactionCount	5
transactionsRoot	32
transactionsHash	32
Total Block Header	109

Transaction Encoding

Transactions Buffer

Encoding of a transaction in the block.transactions buffer.

Type	Size (b)	Size w/ root (b)
HARD_CREATE	56	88
HARD_DEPOSIT	16	48
HARD_WITHDRAW	16	48
SOFT_WITHDRAW		

	99	131
SOFT_CREATE	123	155
SOFT_TRANSFER	83	115
SOFT_CHANGE_SIGNER	93	125

Leaf Encoding

Encoding of a transaction in the transactions merkle tree.

Type	Prefix	Size (b)	Size w/ root and prefix (b)
HARD_CREATE	0x00	56	89
HARD_DEPOSIT	0x01	16	49
HARD_WITHDRAW	0x02	16	49
SOFT_WITHDRAW	0x03	99	132
SOFT_CREATE	0x04	123	156

SOFT_TRANSFER

0x05

83

116

SOFT_CHANGE_SIGNER

0x06

93

126