Abstract

We introduce a scheme for compact general-purpose fraud proofs specifically for UTXO data model blockchains that does not require serializing intermediate states. As such, it is more conducive for parallelization compared to previous general-purpose fraud proof schemes.

Background

Prerequisite Reading

- Data availability proof-friendly state tree transitions
- Optimizing sparse Merkle trees
- Fraud and Data Availability Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities
- Minimal Viable Merged Consensus
- Practical parallel transaction validation without state lookups using Merkle accumulators

Extra Reading

- The Stateless Client Concept
- State Providers, Relayers - Bring Back the Mempool

Motivation

Light clients (known as SPV clients in the Bitcoin ecosystem) are low-resource nodes that only validate block headers and potentially Merkle paths that are logarithmic in size. They are intended to be ran on devices that simply aren't powerful enough, or don't have enough bandwidth, to fully download and validate even a single block. We want to ensure that such clients still have a measure of safety and security when using the network. One way to do so is compact fraud proofs

, which are proofs that a block is invalid with O(\log \text{blocksize})

cost.

A scheme for general-purpose fraud proofs is proposed here by Al-Bassam et al. (@musalbas), and involves including intermediate state roots at a parametrizable interval ("period"), P

. This can be committed to in a single data structure (i.e.

, a single Merkle tree that contains, in order, P

transactions, the intermediate state root after executing those transactions, another P

transactions, and so on), or multiple data structures (i.e.

, a Merkle tree for transactions, another for intermediate state roots).

Unfortunately, this requires serializing the state and computing a new state root every P

transactions, which is an inherently single-threaded process. If P

is too large, then fraud proofs quickly become unwieldy for light clients in practice despite identical asymptotic costs. If P

is too small, state serialization costs grow. It turns out there's no good way of picking P

to avoid both these issues, especially given that transactions can each have potentially unbounded side effects on the state.

[Block headers committing to all intermediate state roots] actually used

to be the case [in Ethereum]. It was removed because serializing the tree between every transaction has too large an impact on performance (source).

We would like to have a scheme that provides us with the same expressive power as these fraud proofs, but without requiring intermediate state serialization, which would allow us to scale out

in parallel (with linearly increasing validation costs) instead of scaling up

(with exponentially increasing validation costs).

Fraud Proofs Model

At its core, a fraud proof of invalid state transition consists of

\mathtt{VerifyTransition}(state_{pre}, tx^P, state_{post}) \rightarrow \mathtt{bool}

returning \mathtt{true}

. Note that state_{pre}

and state_{post}

(the pre- and post-states of the period in question) aren't just state roots or the full state, but rather witnesses to the relevant state elements

in the pre- and post-state (which must be verified independently). This is possible due to the state being serialized in a dynamic authenticated data structure, such as a Sparse Merkle Tree (SMT). Also note that, likewise, tx

is a witness to a transaction. Other than that, it basically executes the transactions as normal state transitions according to the state transition rules of the underlying VM starting from the pre-state and checks it against the post-state.

In practice there is some extra data needed, such as block number or blockhash, etc., but those are implementation details and don't meaningfully affect the model or its costs.

Fraud Proofs Without Intermediate State Serialization

The paper above makes one fundamental assumption that we will challenge here: that transactions in a block are executed in order (i.e.

, they are topologically ordered). This is only necessary in an accounts data model chain. In a UTXO data model chain this is actually not needed, and is merely an implementation artifact from Bitcoin, as we will soon show.

Model

Transactions and Blocks

The transaction format that users see and sign is basically the same as the usual one for a UTXO data model chain: a list of inputs being spent and a list of outputs being generated. The exact format beyond this is an implementation detail.

The block format, and how transactions are represented in a block, is changed however. Block headers now contain three (3) data roots:

1. ordered inputs
2. ordered outputs
3. ordered transactions

Ordered Merkle trees can be used as they allow us to prove non-inclusion with two adjacent Merkle branches, though SMTs may be used for more efficient block template generation without having to re-compute the whole trees (and are conveniently ordered by definition).

Transactions are ordered by their transaction ID, which is just the hash that the user must sign. Inputs of transactions in a block do not refer to an unspent output but rather an index in the ordered inputs list. Outputs refer to an index in the ordered outputs list.

The tree of ordered inputs consists of the IDs of the outputs being spent in the block (i.e.

, hashes), which is used as the key, the source block in which this UTXO was generated, and a reference to the transaction in this block that spends this input. Any other data that would normally be included for an input (in order to generate the ID) is stored here.

The tree of ordered outputs consists of the IDs of the outputs being generated, which is used as the key, and a reference to the transaction in this block that generates this output. Any other data that would normally be included for an output (in order to generate the ID) is stored here.

Note: just as described above for the previous fraud proof scheme, these multiple trees can be represented as a single tree. This is an implementation detail.

Fraud Proofs

The \mathtt{VerifyTransition}

method proceeds in much the same way as described above: given pre-state witnesses, a transaction, and post-state witnesses, it verifies that everything matches up. The difference lies in how the pre- and post-state witnesses are formatted and verified.

For the transaction, the fraud proof provides:

1. A witness that the transaction exists in the list of ordered transactions for the current block.
2. For each input (pre-state):
3. A witness that the referenced input exists in the list of ordered inputs for the current block, or
4. Alternatively, if the referenced input does not exist, a witness to this effect.
5. A witness that that the input exists in the list of ordered outputs for the source block, or
6. Alternatively, if the input has been spent elsewhere, a witness to this effect (including if it has been spent in the current block by a different transaction).
7. A witness that the referenced input exists in the list of ordered inputs for the current block, or
8. Alternatively, if the referenced input does not exist, a witness to this effect.
9. A witness that that the input exists in the list of ordered outputs for the source block, or
10. Alternatively, if the input has been spent elsewhere, a witness to this effect (including if it has been spent in the current block by a different transaction).
11. For each output (post-state), optionally:
12. A witness that the referenced output does not exist in the list of ordered outputs for the current block.
13. A witness that the referenced output does not exist in the list of ordered outputs for the current block.

We need to make sure that the inputs spent by the block and the outputs generated by the block all originate from exactly one transaction, so a second fraud proof form is needed.

For the input (output), the fraud proof provides:

1. A witness that the input (output) exists in the list of ordered inputs (outputs) for the current block.
2. A witness that the referenced transaction does not exist in the list of ordered transactions for the current block, or
3. Two witnesses that the referenced transaction and another one both exist in the list of ordered transactions for the current block and both spend (generate) the same input (output).

Incorrect construction of any of the three trees can be proven compactly trivially.

The correctness of collected transaction fees and newly minted coinscan be verified using Merkle sum trees for the input and output trees.

Analysis

The above scheme gives us a very bizarre property: transaction ordering does not matter for execution! We simply need to check that everything in the list of inputs was previously unspent (including being in the list of outputs of the current block) and that all items in the lists of inputs and outputs are unique. As such, transactions in a block happen “all at once.” This property has been discussed before with the suggestion to adopt a canonical lexical transaction ordering for Bitcoin Cash.

Block producers using this fraud proof scheme do not need to keep track of state in an authenticated data structure at all (though they should still keep track of the state—UTXO set—as that’s needed to generate the three trees in the first place), and especially don’t have to compute intermediate state roots. This makes it possible and easy to parallelize, allowing block producers and validators to scale out

.

Correctness

Proof of correctness by induction is left as an exercise for the reader.

Intuitions

Note that what is proposed here does not preclude committing to a state root at each block, it only and specifically deals with not needing intermediate state roots for fraud proofs.

Interestingly, the act of generating intermediate state roots is isomorphic togenerating a proof for a stateless client. As such, the enormous single-threaded cost of serializing the state after every transaction is something that relayers / state providers / light client servers will have to deal with in Eth 2.0.

Conclusion

We present a non-novel scheme for general-purpose fraud proofs for UTXO data model blockchains that is thread-friendly to generate, unlike previous proposals.

Updates

The above scheme can be simplified: rather than having three separate trees, each input simply needs to have metadata pointing to the and transaction index that generated the UTXO (thanks to Jonathan Toomim for suggesting this).

It turns out this simplified scheme is essentially identical to the one suggested inBIP-141 all the way in 2015, and does not require lexicographically ordering transactions within a block (thanks to Gregory Maxwell for pointing this out to me). To the best of my limited knowledge, this idea originated with Rusty Russell.