Ideas on how to implement epochs, and how they impact all the state read and write operations we have.

Resources

Thanks Mike for these!

- Resurrection-conflict-minimized state bounding by Vitalik
- Epoch-based nullifier database by Polygon Miden team
- Abstracting contract deployment on our forum

Context

Epochs are predetermined periods of time (eg 1 year), where each epoch has its own set of global state trees. When the epoch changes, we reset the trees, and store the roots of previous trees in a separate structure (an “epochs tree”). State from old epochs is immutable.

We expect nodes to store the current set of trees, as well as the ones from at least one epoch ago. Nodes that store data from all epochs are “archival nodes”.

Epochs tree

The epochs tree can be implemented as an actual tree, or as a simple hash chain (as suggestedhere), since many operations require accessing a contiguous list of roots. It’s possible we can even do both: an append-only merkle tree where a new leaf is computed as the hash of the previous leaf and the new value.

Addresses

We’ll assume that all addresses have an epoch identfier. To avoid increasing the length of an address, and to avoid issues related to address preimage withholding, we can define epoch identifiers to be the first byte of an address. This gives us 256 years worth of chain. Users mine their address by attempting different salts until they match the current (or a past) epoch. Note that state for a given address can only be accessed if their epoch identifier is equal to the current epoch or in the past.

Global state

How would we read from and write to each global state tree if we added epochs…?

Note hashes tree

This is the easy one. The note hashes tree is an append-only merkle tree. Writing is always done on the current epoch tree. Reading from the current or previous epoch is done directly via a merkle membership proof, while reading from older ones also requires proving that the old root is present in the epoch tree.

This means that, when an epoch ends, a client should save the merkle membership proof for each of their active notes, so they can spend them in future epochs. If they don’t, they’ll require access to an archive node when they actually want to spend them.

If public functions are expected to be able to read from the note hashes tree, then users should submit the merkle membership proofs for any archived notes read during public execution, or the tx reverts.

Public data tree

This one has the same challenges as Ethereum (go readthis post from Vitalik!). The public data tree is a key-value store implemented over an indexed tree.

Updating or reading an entry that’s present on the current epoch is direct.

If the entry is on the immediately previous epoch, then reading requires proving that the key is not present in the current epoch, and proving that it is on the previous one. Both of these trees are available to the node. Updates just require writing to the tree on the current epoch.

If the entry is in an archived epoch, then we cannot expect the sequencer to be able to access it. Here the tx is required to contain a proof for all archived values being read in public function executions, and the tx must

revert if any of them is missing.

Note that the proof for a given entry requires a membership proof in the epoch's tree where the entry was last written to, as well as non-membership proofs for all archived trees after that one. If the entry was never written to, instead of requiring non-membership proofs for every epoch since genesis, we can start from the epoch identifier of the entry's address.

This means that, if a user needs to call a public function that depends on archived values, they will require access to an archive node. They will also need to accurately predict all slots that will be accessed during public execution of their tx.

To minimize dependence on archives, we can force sequencers to "copy" a value to the current epoch when it's read from an older one. This would increase write operations, but would be handy for executions that depend on public data that rarely changes (eg configs). As long as a value is read at least once per epoch, it's guaranteed that it will always be available without having to resort to an archive node.

Nullifier tree

This is the tricky one. The nullifier tree is an indexed tree of unique values.

Reading a previously emitted nullifier follows the same rules as reading a note hash. However, for note hashes, users know what are their active notes and can cache their merkle paths for future usage - for nullifiers, it's unclear how users would know which ones to cache. Fortunately, we don't have many use cases where we need to prove existence of a nullifier: the main one is immutable singletons used for tracking contract initialization (assuming we implement constructor abstraction). Proofs for these nullifiers could be cached and provided to users by dapps, or by whatever services are expected to distribute the circuits associated with a private contract. We can also explore refresh nullifiers

(see below) as an alternative.

Emitting a given nullifier would require proving that the nullifier wasn't emitted in any archived epochs. This should be done by the private kernel circuit. We can optimize this by starting from the address' epoch identifier, instead of genesis, as we do with public state readings. And we can further optimize this by including an epoch identifier in a note itself (set by default to context.current_epoch

), and starting from there. This means that the primitive for emitting a nullifier from an app now requires an additional parameter: the earliest epoch number in which this nullifier could have been emitted. This parameter is set to the epoch identifier in the note being nullified, or the immediately previous one if the new epoch is too recent and the privacy set is still too small.

This approach allows for the same nullifier to be emitted more than once

across different epochs. This is useful for "refreshing" a nullifier that needs to be frequently read, as is the case with initialization nullifiers, since an app can just re-emit it at a newer epoch. However, this can leak privacy, since two txs emitting the same nullifier could be linked together. It also makes double-spend bugs harder to spot.

As an alternative, we can silo nullifiers per epoch. When a contract at address

emits a (nullifier, min_epoch)

tuple, the actual nullifier that is emitted from the transaction is hash(nullifier, address, min_epoch)

. This makes nullifiers globally unique. However, it makes it more difficult to refresh a nullifier, since apps that depend on reading a nullifier will need to accept any nullifier-epoch combination.

Other trees

- L1 to L2 messages: append-only tree, should be just like note hashes tree (edit: we may be able to kill it and just store these messages in the data tree, thanks @Maddiaa for the heads-up!)
- Archive tree: same, or maybe doesn't even need epochs?
- Contracts tree: we're killing it in favor of overloading the nullifier tree.

Next steps

- Identify what changes we need to make to the protocol as is now to accommodate for all the above.
- Add formulas to complement the explanations.