Note - The word “UTXO”, “note” and “commitment” is used interchangeably.

Background

Aztec is using UTXOs for its private state. This opens us up to UTXO concurrency issues faced by some other programmable blockchains that also implement programmable UTXO’s.

Say, I have a bunch of private notes in a smart contract (like a token contract that stores balances privately) and I use some of them in a transaction. When fetching the notes from the DB, we always fetch the first X notes that haven’t been nullified (ref 1 and ref 2), i.e. we use no offset by default.

Once these notes are used in the transaction, they can be considered as pending until the state is updated (happens when block is finalised on L1), which may take 2-10 minutes (depending on how quickly we can our prover network can prove). But until then, the transaction remains pending and the commitments haven’t been nullified.

So while this transaction is pending, if the user were to fetch notes again on the same contract, they would get the same notes (since they haven’t been nullified yet). This means a user is limited to only 1 transaction per contract per block

(for tx that require access to notes). This is similar to issues popularized by DEXs built on Cardano, which if naively implemented can just allow for 1 swap per pool per block (since the UTXOs of the pool could be used by anther transaction so all other transactions on the pool would fail as the original UTXO has now been destroyed).

This is bad, and is a fundamental challenge that needs to be addressed for not only Aztec’s scalability, but all blockchains attempting to utilize programmable UTXO’s.

What are we currently doing about this?

There is a PR on fetching a variable number of notes at any offset (thanks Leila!). This is useful in situations like say I know my first two notes are stuck in a pending transaction. Then when fetching two more notes for my next transactiont, I would just pass an offset of two. Now I get two different notes that I can spend in parallel (making 4 notes pending).

One solution is for the Aztec RPC Server to do the offset calculation. The other alternative is having dapp develoeprs to have their own algorithm specific to their app (which requires deep understanding of UTXOs and a lot of modelling on how their dapp would be used). I personally think the former (RPC Server doing offset calculation) is better as it provides for an easier dev ex.

The RPC Server could do the calculation in the following way - A typical transaction would fetch notes and then consume them. This calls compute_nullifier()

on Noir, which fetches the user’s secret key and computes the nullifer via an oracle call to the RPC Server. Each time this method is called, the node could tell the RPC Server to shift the offset (shoutout to @jaosef for this idea).

This assumes every time compute_nullifier()

is called, it is to remove the note. Is there any case where you would call this method to not nullify?

But wait! There is more!

The above solution doesn’t fully fix the problem

.

Say the user has their wallet (read their instance of the rpc server/simultor) on multiple devices.

They do a transaction from one device and the transaction is still pending. If you now go to the other device to make another transaction on the same contract, as implemented today, it would end up fetching the same notes (as the offset is only corrected locally on the first device) and the kernel proof would be built correctly. Ofcourse when sending through our rollup circuits, the transaction will revert but this is poor UX.

Ethereum has this issue too but this is exacerbated by the up to 10 minute block times that Aztec has

, which could result in 20 minutes to reconcile (worst case) versus the 20-30 seconds Ethereum could theoretically provide. Therefore, there is more incentive to ensure that Aztec gets this right. Waiting for 10+ minutes only to discover that your tx is reverting can be quite frustating.

Unanswered questions

- how can a dapp developer algorithmically determine at run time, the optimal number of notes to split the balance of their user (or their contract) into?

Pending commitments allow a user to efficiently split a large note into several smaller notes or join several small notes into one large ones. But how do determine how to split/join them? Say a contract has 1 note of value 10 ETH. Say I want to take 1 eth out of it while Jake wants to take 2 eth. Both these transactions are happening in private i.e. locally on user devices. I would split the contract's utxo into two of values 1 eth and 9 eth respectively, while Jake would split it as 8 and 2. Ofcourse only one transaction succeeds (say mine), but now Jake's fails even though the contract has sufficient funds to process both of our transactions!

A good dapp developer would need to create algorithms to identiy how to store their user's balances and also their contract's pool. Most UTXO based DEXes have used offchain aggregators to combine all transactions into one (which have their own tokenomics model). Ref

More related questions:

- Good algorithm designs on how can a dapp developer programmatically figure out how many notes of a user to fetch?
- … or when to split/join notes?
- … or how to determine the right offset?

One easy answer is have an unconstrained function in your noir smart contract which fetches multiple notes of a user and lets dapp developers determine it that way. Not sure how privacy preserving that would be…

One Final Note for DevRel

- We need to also do a good job of providing documentation and examples demonstrating the negative impact on UX if the UTXO concurrency issues are handled naively.

Shoutout to @cooper-aztecLabs and @suyash67 for their feedback!