Private - Public execution

Disclaimer Disclaimer We are building Aztec as transparently as we can. The documents published here are living documents. The protocol, sandbox, language, and tools are all subject to change over time.

Please seehere for details of known Aztec protocol and Aztec Sandbox limitations.

If you would like to help us build Aztec:

- Contribute code onGitHub
- ; or
- Join inforum
- discussions. Aztec operates on a model of private and public functions that are able to work together. Private functions work by providing evidence of correct execution generated locally through kernel proofs. Public functions, on the other hand, are able to utilize the latest state to manage updates and perform alterations.

On this page, you'll learn:

- How private and public functions work
- The role of public functions in managing state alterations and updates
- Communication and interactions between private and public functions
- How the sequencer manages the order of operations of private functions

Objectives

The goal for L2 communication is to setup the most simple mechanism that will support

- private
- andpublic
- functions
- private
- functions that can callprivate
- orpublic
- functions
- public
- functions that can callprivate
- orpublic
- functions

Before diving into the communication abstracts for Aztec, we need to understand some of our limitations. One being that public functions (as known from Ethereum) must operate on the current state to provide meaningful utility, e.g., at the tip. This works fine when there is only one builder (sequencer) executing it first, and then others verifying as the builder always knows the tip. On the left in the diagram below, we see a block where the transactions are applied one after another each building on the state before it. For example, if Tx 1 update storagea = 5 , then in Tx 2 readinga will return5 .

This works perfectly well when everything is public and a single builder is aware of all changes. However, in a private setting, we require the user to present evidence of correct execution as part of their transaction in the form of a kernel proof (generated locally on user device ahead of time). This way, the builder doesn't need to have knowledge of everything happening in the transaction, only the results. If we were to build this proof on the latest state, we would encounter problems. How can two different users build proofs at the same time, given that they will be executed one after the other by the sequencer? The simple answer is that they cannot, as race conditions would arise where one of the proofs would be invalidated by the other due to a change in the state root (which would nullify Merkle paths).

To avoid this issue, we permit the use of historical data as long as the data has not been nullified previously. Note, that because this must include nullifiers that were inserted after the proof generation, but before execution we need to nullify (and insert the data again) to prove that it was not nullified. Without emitting the nullifier we would need our proof to point to the current head of the nullifier tree to have the same effect, e.g., back to the race conditions we were trying to avoid.

In this model, instead of informing the builder of our intentions, we construct the proofπ \pi π and then provide them with the transaction results (new note hashes and nullifiers, contract deployments and cross-chain messages) in addition toπ \pi π . The builder will then be responsible for inserting these new note hashes and nullifiers into the state. They will be aware of the intermediates and can discard transactions that try to produce existing nullifiers (double spend), as doing so would invalidate the rollup proof.

On the left-hand side of the diagram below, we see the fully public world where storage is shared, while on the right-hand side, we see the private world where all reads are historical.

Given that Aztec will comprise both private and public functions, it is imperative that we determine the optimal ordering for these functions. From a logical standpoint, it is reasonable to execute the private functions first as they are executed on a

stateS i S_i S i , wherei ≤ n i \le n i

≤ n , withS n S_n S n representing the current state where the public functions always operate on the current stateS n S_n S n . Prioritizing the private functions would also afford us the added convenience of enabling them to invoke the public functions, which is particularly advantageous when implementing a peer-to-pool architecture such as that employed by Uniswap.

Transactions that involve both private and public functions will follow a specific order of execution, wherein the private functions will be executed first, followed by the public functions, and then moving on to the next transaction.

It is important to note that the execution of private functions is prioritized before executing any public functions. This means that private functions cannot "wait" on the results of any of their calls to public functions. Stated differently, any calls made across domains are unilateral in nature, akin to shouting into the void with the hope that something will occur at a later time. The figure below illustrates the order of function calls on the left-hand side, while the right-hand side shows how the functions will be executed. Notably, the second private function call is independent of the output of the public function and merely occurs after its execution.

Multiple of these transactions are then ordered into a L2 block by the sequencer, who will also be executing the public functions (as they require the current head). Example seen below.

info Be mindful that if part of a transaction is reverting, say the public part of a call, it will revert the entire transaction. Similarly to Ethereum, it might be possible for the block builder to create a block such that your valid transaction reverts because of altered state, e.g., trade incurring too much slippage or the like. To summarize:

- Private
- function calls are fully "prepared" and proven by the user, which provides the kernel proof along with new note hashes and nullifiers to the sequencer.
- Public
- functions altering public state (updatable storage) must be executed at the current "head" of the chain, which only the sequencer can ensure, so these must be executed separately to theprivate
- functions.
- Private
- andpublic
- functions within an Aztec transaction are therefore ordered such that firstprivate
- functions are executed, and thenpublic
- .

A more comprehensive overview of the interplay between private and public functions and their ability to manipulate data is presented below. It is worth noting that all data reads performed by private functions are historical in nature, and that private functions are not capable of modifying public storage. Conversely, public functions have the capacity to manipulate private storage (e.g., inserting new note hashes, potentially as part of transferring funds from the public domain to the secret domain).

info You can think of private and public functions as being executed by two actors that can only communicate to each other by mailbox. So, with private functions being able to call public functions (unilaterally) we had a way to go from private to public, what about the other way? Recall that public functions CANNOT call private functions directly. Instead, you can use the append-only merkle tree to save messages from a public function call, that can later be executed by a private function. Note, only a transaction coming after the one including the message from a public function can consume it. In practice this means that unless you are the sequencer it will not be within the same rollup.

Given that private functions have the capability of calling public functions unilaterally, it is feasible to transition from a private to public function within the same transaction. However, the converse is not possible. To achieve this, the append-only merkle tree can be employed to save messages from a public function call, which can then be executed by a private function at a later point in time. It is crucial to reiterate that this can only occur at a later stage and cannot take place within the same rollup because the proof cannot be generated by the user.

info Theoretically the builder has all the state trees after the public function has inserted a message in the public tree, and is able to create a proof consuming those messages in the same block. But it requires pending UTXO's on a block-level. From the above, we should have a decent idea about what private and public functions can do inside the L2, and how they might interact.

A note on L2 access control

Many applications rely on some form of access control to function well. USDC have a blacklist, where only parties not on the list should be able to transfer. And other systems such as Aave have limits such that only the pool contract is able to mint debt tokens and transfers held funds.

Access control like this cannot easily be enforced in the private domain, as reading is also nullifying(to ensure data is up to date). However, as it is possible to read historical public state, one can combine private and public functions to get the desired effect.

Say the public state holds amapping(address user => bool blacklisted) and a value with the block number of the last updatelast_updated . The private functions can then use this public blacklist IF it also performs a public function call that reverts if the block number of the historical state is older than thelast_updated . This means that updating the blacklist would make pending transactions fail, but allow a public blacklist to be used. Similar would work for the Aave example, where it is just a public value with the allowed caller contracts. Example of how this would be written is seen below. Note that because theonlyFresh is done in public, the user might not know when he is generating his proof whether it will be valid or not.

function

transfer (secret address to , secret uint256 amount , secret HistoricalState state) secret returns (bool)

{ if

(blacklisted [msg . sender]

|| blacklisted [to])

revert ("Blacklisted") ; onlyFresh (state . blockNumber) ; // logic }

function

onlyFresh (pub uint256 blockNumber)

public

{ if

(blockNumber < last_updated)

revert ("Stale state") ; } info This is not a perfect solution, as any functions using access control might end up doing a lot of public calls it could put a significant burden on sequencers and greatly increase the cost of the transaction for the user. We are investigating ways to improve. Using a dual-tree structure with a pending and a current tree, it is possible to update public data from a private function. The update is fulfilled when the pending tree becomes the current after the end of a specified epoch. It is also possible to read historical public data directly from a private function. This works perfectly for public data that is not updated often, such as a blacklist. This structure is called a slow updates tree, and you can read about how it worksin the next section . Edit this page

Previous Contract Creation Next Privately access Historical Public Data