Accounts

Every account in Aztec is a smart contract which defines the rules for whether a transaction is or is not valid. This allows implementing different schemes for transaction signing, nonce management, and fee payments. However, encryption and nullifying keys, which are specific to private blockchains, are still enshrined at the protocol level.

In this section, you'll learn about how Aztec defines AA (account abstraction) and its correlation with encryption keys and nullifying keys. We'll go through:

- The importance and implications of AA
- Understanding account contracts and wallets in relation to Aztec
- Concept of authorization and actions along with encryption
- The future of fee management in Aztec

Background

We'll start with the mandatory "what is AA" section that every single article on the topic has, so you can skip this if you're familiar with the topic.

What is account abstraction?

We'll refer to AA as theability to set the validity conditions of a transaction programmatically (source).Starknet goes one step further and splits AA into three different components:

- Signature abstraction (defining when a signature is accepted)
- Fee abstraction (paying fees)
- Nonce abstraction (replay protection and ordering)

In most AA schemes, the identity of a user is no longer represented by a keypair but by a contract, often called a smart contract wallet or account contract. This contract receives transaction payloads which are validated with custom logic, and then interpreted as actions to execute, like calling into another contract.

The benefits of AA are multiple. We're not going to reiterate them all here, but they include social recovery, MFA, batching, session keys, sponsored transactions, fee payment in kind, supporting key schemes from different realms, etc. Read the articles fromArgent orEthereum.org for more detailed info.

Implementing at protocol vs application layer

Instead of implementing it at the protocol level as in Aztec, account abstraction can be implemented at the application layer of a network using smart accounts and meta-transactions. When implementing account abstraction on Ethereum, the transaction being sent to the network is still an Ethereum transaction, but its payload is interpreted as a "transaction execution request" that is validated and run by the smart contract wallet.

A simple example would be Gnosis Safe (seeAccount Abstraction is NOT coming), where it's the multisig contract responsibility to define when an execution request is valid by checking it carries N out of M signatures, and then executing it.Argent has also been working on smart wallets for years, and collaborating with network teams to implement AA natively at the protocol layer.

Ethereum is currently following this approach viaEIP4337 , an evolution of theGSN . This EIP defines a standard method for relaying meta-transactions in a decentralized way, including options for delegating payment to other agents (called paymasters). Seethis chart on how 4337 relates to other smart contract wallet efforts.

Implementing AA at the application layer has the main drawback that it's more complex than doing so at the protocol layer. It also leads to duplicated efforts in both layers (eg the wrapper transaction in a meta-transactions still needs to be checked for its ECDSA signature, and then the smart contract wallet needs to verify another set of signatures).

Now, there have also been multiple proposals for getting AA implemented at theprotocol level in Ethereum. This usually implies introducing a new transaction type or set of opcodes where signature verification and fee payment is handled by the EVM. See EIPs2803 ,2938 , or3074 . None of these have gained traction due to the efforts involved in implementing while keeping backwards compatibility.

However, other chains are experimenting with protocol-level AA. BothStarknet andzkSync have native AA, zkSync being the first EVM-compatible one to do so. To maintain Ethereum compatibility, zkSync implements adefault account contract in Solidity that mimics Ethereum's protocol behavior.

Preventing DoS attacks

Protocol AA implementations are vulnerable to DoS attacks due to the unrestricted cost of validating a transaction. If validating a transaction requires custom logic that can be arbitrarily expensive, an attacker can flood the mempool with these transactions that block builders cannot differentiate from legit ones.

Application AA implementations face a similar issue: a smart wallet could return that a transaction is valid when a relayer is about to submit it on-chain and pay for its gas, but when the transaction is actually mined it could turn invalid.

All implementations mitigate these issues by restricting what's doable in the validation phase. EIP4337 defines a set of prohibited opcodes and limits storage access (seeSimulation in the EIP), and requires areputation system for global entities. zkSyncrelaxes opcode requirements a bit, and Starknet simplydoes not allow to call external contracts.

Accounts in Aztec

Aztec has no concept of Externally Owned Accounts. Every account is implemented as a contract. Account contracts typically implement anentrypoint function that receives the actions to be carried out and an authentication payload. In pseudocode:

publicKey: PublicKey;

def entryPoint(payload): let { privateCalls, publicCalls, nonce, signature } = payload; let payloadHash = hash(privateCalls, publicCalls, nonce); validateSignature(this.publicKey, signature, payloadHash);

foreach privateCall in privateCalls: let { to, data, value } = privateCall; call(to, data, value);

foreach publicCall in publicCalls: let { to, data, value, gasLimit } = publicCall; enqueueCall(to, data, value, gasLimit); Read more about how to write an account contracthere .

Account contracts and wallets

Account contracts are tightly coupled to the wallet software that users use to interact with the protocol. Dapps submit to the wallet software one or more function calls to be executed (eg "call swap in X contract"), and the wallet encodes and authenticates the request as a valid payload for the user's account contract. The account contract then validates the request encoded and authenticated by the wallet, and executes the function calls requested by the dapp.

Execution requests

Note that nothing related to signature verification or payload execution is enshrined in the protocol, since account contracts are free to define this entrypoint however they see fit. Therefore, a request for executing an action has a simpler structure than in Ethereum, and just requires:

- Theorigin
- contract to execute as the first step
- The initial function to call (usuallyentrypoint
-)
- The arguments (which encode the private and public calls to run as well as any signatures)

Entrypoint restrictions

Entrypoint methods are not enshrined in the protocol, and any function can be called as an entrypoint. The only restriction is that it must be private (not open), so all transactions are initiated with a client-side zero-knowledge proof.

This means that, unlike other protocols, Aztec does not impose any restrictions on the actions that can be carried out during the validation phase of the entrypoint, since these actions are executed by the client and wrapped in a zero-knowledge proof that is verified by the network. You are free to call into other contracts, access storage, or do as much computing as you need in the entrypoint.

Nonces and replay protection

Every transaction execution considered valid by the protocol emits the hash of the transaction execution request as a nullifier, preventing the same transaction from being executed more than once. Nonces, on the other hand, are left to the account contract implementation. This allows building accounts with strictly incremental nonces or where transactions can be processed out-of-order.

A side-effect of not having nonces at the protocol level is that it is not possible to cancel pending transactions by submitting a new transaction with higher fees and the same nonce.

Non-standard entrypoints

Since theentrypoint interface is not enshrined, there is nothing that differentiates an account contract from an application one in the protocol. This means that a transaction can be initiated in any contract. This allows implementing functions that do not need to be called by any particular user and are just intended to advance the state of a contract.

As an example, we can think of a lottery contract, where at some point a prize needs to be paid out to its winners. Thispay action does not require authentication and does not need to be executed by any user in particular, so anyone could submit a transaction that defines the lottery contract itself asorigin andpay as entrypoint function. For an example of this behavior see ournon_contract_account test and theSignerLess wallet implementation. Notice that the Signerless wallet doesn't invoke an entrypoint function of an account contract but instead invokes the target contract function directly.

info In case no contract entrypoint is usedmsg_sender is set to 0.

Account initialization

The protocol requires that every account is a contract for the purposes of sending a transaction. This means that a user needs to deploy their account contract as their first action when they want to interact with the network.

However, this is not required when sitting on the receiving end. A user can deterministically derive their address from their encryption public key and the account contract they intend to deploy, and share this address with other users that want to interact with thembefore they deploy the account contract.

Authorizing actions

Account contracts are also expected, though not required by the protocol, to implement a set of methods for authorizing actions on behalf of the user. During a transaction, a contract may call into the account contract and request the user authorization for a given action, identified by a hash. This pattern is used, for instance, for transferring tokens from an account that is not the caller.

When executing a private function, this authorization is checked by requesting anauth witness from the execution oracle, which is usually a signed message. The PXE is responsible for storing these auth witnesses and returning them to the requesting account contract. Auth witnesses can belong to the current user executing the local transaction, or to another user who shared it out-of-band.

However, during a public function execution, it is not possible to retrieve a value from the local oracle. To support authorizations in public functions, account contracts should save in contract storage what actions have been pre-authorized by their owner.

These two patterns combined allow an account contract to answer whether an actionis_valid for a given user both in private and public contexts.

Encryption and nullifying keys

Aztec requires users to defineencryption and nullifying keys that are needed for receiving and spending private notes. Unlike transaction signing, encryption and nullifying is enshrined at the protocol. This means that there is a single scheme used for encryption and nullifying. These keys are derived from a master public key. This master public key, in turn, is used when deterministically deriving the account's address.

A side effect of committing to a master public key as part of the address is thatthis key cannot be rotated . While an account contract implementation could include methods for rotating the signing key, this is unfortunately not possible for encryption and nullifying keys (note that rotating nullifying keys also creates other challenges such as preventing double spends). We are exploring usage of theslow updates tree to enable rotating these keys.

NOTE: While we entertained the idea of abstracting note encryption, where account contracts would define anencrypt method that would use a user-defined scheme, there are two main reasons we decided against this. First is that this entailed that, in order to receive funds, a user had to first deploy their account contract, which is a major UX issue. Second, users could define maliciousencrypt methods that failed in certain circumstances, breaking application flows that required them to receive a private note. While this issue already exists in Ethereum when transferring ETH (see theking of the hill), its impact is made worse in Aztec since any execution failure in a private function makes the entire transaction unprovable (ie it is not possible to catch errors in calls to other private functions), and furthermore because encryption is required for any private state (not just for transferring ETH). Nevertheless, both of these problems are solvable. Initialization can be worked around by embedding a commitment to the bytecode in the address and removing the need for actually deploying contracts before interacting with them, and the king of the hill issue can be mitigated by introducing a full private VM that allows catching reverts. As such, we may be able to abstract encryption in the future as well.

Fee management

Fees are not implemented in the protocol at the time of this writing. Our goal is to abstract fee payments as well. This means that a transaction, in order to be considered valid, must prove that it has locked enough funds to pay for itself. However, this does not mandate where those funds come from, opening the door for easy implementation of paymasters or payment-in-kind via on-the-fly swaps.

However, there is one major consideration around public execution reverts. In the current design, if one of the public function executions enqueued in a transaction fails, then the entire transaction is reverted. But reverting the whole transaction would also revert the fee payment, and leave the sequencer with their hands empty after running the public execution. This means we will need to enshrine an initial verification and fee payment phase that isnot reverted if public execution fails. Edit this page

Previous Storage Slots Next Keys