Some ramblings on how we could implement AA on Aztec. Definitely not a spec, and somewhat of a continuation of this post.

TL;DR

- Every account is a smart contract wallet, there is no concept of EOAs in the protocol
- A transaction request is just an opaque payload and the smart contract wallet address that validates, interprets, and executes it
- The smart wallet address and the payload get hashed together to produce the tx hash, which is emitted as the first nullifier to prevent replay attacks
- Smart contract wallet entrypoints are private functions that call other private functions or enqueue calls to public functions, and handle signature verification and fee payments
- The protocol just checks that fee payments were made as part of the private kernel circuit, and doesn't perform any signature checks itself
- The only validity check that the sequencer does is that of the private kernel proof; the sequencer does not need to run any custom validation logic, removing the issue with DoS attacks
- The sender of a tx in the protocol is always an Aztec address representing a smart contract wallet, regardless of the signature scheme used by the user's wallet, so we don't need to introduce eg Ethereum addresses in the protocol's Tx object
- There is no top-level signed tx request for executing a public function, it's the wallet contract responsibility to enqueue these calls after validating the user's signature and locking gas to cover for public execution

What is account abstraction?

Let's start with the mandatory "what is AA" section that every single article on the topic has, so you can skip it if you're familiar with the topic. We'll refer to AA as the ability 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 tx 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 txs, fee payment in kind, supporting key schemes from different realms, etc. Read the articles from Argent or Ethereum.org for more detailed info.

Implementing at protocol vs application layer

Account abstraction can be implemented at the application layer of a network using smart accounts and meta-transactions. The tx being sent to the network is still an Ethereum tx, 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 (see [Account Abstraction is NOT coming

](https://safe.mirror.xyz/9KmZjEbFkmI79s28d9xar6JWYrE50F5AHpa5CR12YGI)), 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 via EIP4337, an evolution of the GSN. This EIP defines a standard method for relaying meta-txs in a decentralized way, including options for delegating payment to other agents (called paymasters). See this 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 tx in a meta-txs 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 the protocol

level in Ethereum. This usually implies introducing a new tx type or set of opcodes where signature verification and fee payment is handled by the EVM. See EIPs 2803, 2938, or 3074. 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. Both Starknet and zkSync have native AA, zkSync being the first EVM-compatible one to do so. To maintain Ethereum compatibility, zkSync implements a default account contract in Solidity that mimicks Ethereum's protocol behaviour.

Preventing DoS attacks

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

Application AA implementations face a similar issue: a smart wallet could return that a tx is valid when a relayer is about to submit it on-chain and pay for its gas, but when the tx 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 (see Simulation in the EIP), and requires a reputation system for global entities. zkSync relaxes opcode requirements a bit, and Starknet simply does not allow to call external contracts.

Account Abstraction in Aztec

Now that we're all on the same page, let's get to it. We'll try to break down AA, and for each item see how it would be tackled in Aztec, starting with an overview of a tx flow under this paradigm:

Transaction flow

We'll assume the user has a smart contract wallet contract already deployed, and that contrat has a private entrypoint function. Then the user wants to call a private and a public function in the same tx…

1. The client creates a transaction request with a sender and payload, where the sender is the address of the user's smart contract wallet, and the payload includes the intent for the two calls and a signature (let's say a BLS one).

2. The client executes the private app circuit over their smart contract wallet entrypoint. The smart contract wallet code takes care of validating the signature, burning gas, and interpreting the payload as calls to other contracts. This spawns another private app circuit run for the private call, and enqueues an item in the public call stack.

3. The client runs the necessary iterations of the private kernel circuit. This outputs a tx object with a set of nullifiers (the first of which is the tx hash) and private data commitments.

4. The tx is now sent to the P2P pool, where the sequencer picks it up and runs it through the public VM circuit to execute the public calls in the public call stack, and then through the public kernel circuit.

5. Tx goes into the rollup circuits and then is published to L1 in an L2block as usual.

TODO: A nice excalidraw diagram of the flow above.

Sample pseudocode

A simple smart contract wallet could look like the following:

publicKey: BLS;

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

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

foreach publicCall in publicCalls:
  let { to, data, value, gasLimit } = publicCall;
  lock(gasLimit);
  enqueueCall(to, data, value);
```

Note that the wallet itself is not validating nonces, since

Externally owned accounts

No EOAs. For simplicity's sake, every account is a smart contract account, so we don't need to reimplement any flavor of signature verification or nonce management at the protocol level.

Initialization

Contracts in Aztec cannot deploy other contracts at the moment, so if we require that every tx goes through a contract, we have a problem. In case it's not possible to patch this, we can use an approach inspired by Starknet's: the user initially seeds their target account address with gas, and then sends a special contract deployment tx that is allowed to consume that gas. The smart contract constructor is responsible for using that initial supply to pay the fees for its own deployment (see fee abstraction below).

Alternatively, the user could send a signed payload along with the contract deployment, and the contract constructor uses that payload to request the user's smart contract wallet to pay the necessary fees.

Either way, contract deployment can be implemented as a special tx that is sent from

the target deployment address itself, and the constructor code handles fee payments along with other initialization code.

Default initialization

Requiring a new user to do an initial deployment tx can be a UX pain. This can be mitigated by delaying the deployment until the first action, and then bundling both together. The smart contract wallet constructor would not just pay for its deployment and run any setup, but also run the first action requested by the user.

Another alternative is having default implementations. We could include a "default account identifier" in the tx payload that indicates what default contract implementation to use. Or we could go the extra mile and split contract instances and contract classes like Starknet does, so the user could just include the class identifier of the account code they want to use. But this exceeds the current discussion.

Upgrading

Changing a user's identity is costly, so some AA approaches recommend using upgradeable contracts for user accounts. And contract upgradeability should be an orthogonal discussion to this one.

Validation DoS attacks

Since contract execution is done and proven at the client, we're safe from DoS attacks on the validation step. Note that this requires that the entrypoints for all smart contract wallets are private

(and not public) functions.

Calling public functions

We'd remove the ability for a protocol-level tx to call a public function directly. Instead, it'll be the smart wallet responsibility to enqueue the calls to public functions from its private entrypoint.

This simplifies the tx flow: we can now have a single entrypoint with a single fee checking mechanism, we no longer need the SignedTxRequest

in the P2P tx object, and it removes the need for a public kernel circuit with no previous input.

Signature abstraction

Having signature handled by the smart contract wallet means that we can add dynamically support for different signature schemes. For instance, we can have smart contract wallets that validate Ethereum ECDSA signatures. Then, at the protocol level, the originator of each tx is the smart contract wallet address

. This means we don't need to support any sort of hybrid Aztec-Ethereum address format at the protocol level: the sender of a tx is always an Aztec network address.

If we want to, we can also add support for signatures with a more SNARK-friendly scheme for cheaper executions. These could be equivalent to the user's Ethereum addresses in terms of privileges, or could be used as temporary session keys.

Fee abstraction

While we haven't fleshed out much of the fee mechanisms, we can anticipate four major cost drivers for each tx: calldata cost, state changes, rollup proving, and public execution.

Calldata cost and state changes resulting from private execution should be known once a tx is sent to the P2P pool. This means we can have the private kernel circuit verify that a certain amount of gas (or should we say mana since we can still choose a name?) has been “burned” to account for that cost. Rollup proving costs are constant, so we can roll them up (pun intended) into the previous two.

On the other hand, public execution and state changes that arise from it are only known once the sequencer processes the tx. Here, we can follow Ethereum’s approach and rely on a gasLimit. We could have a gasLimit for the entire public execution of a tx, or for each individual public call enqueued. The important thing is that the wallet should “lock” the gas limit needed, and once public executions finish, only the actual used gas is burned from the locked amount and the rest is returned.

Note that the protocol is not concerned on how

the fee payment is done, as long as it is performed as part of the private execution. This allows the implementation of paymasters at the protocol level: if the user can produce a proof that their wallet has convinced another contract to pay for the transaction, then it’s valid.

Replay abstraction

We can go AA all the way and abstract nonces as well (see initial discussionhere). This means that the tx nonce no longer has any meaning semantically at the protocol, and it is just used to differentiate to otherwise identical txs. In other words, if the user wants to send two txs with exactly the same args, they’ll need to change something

in the tx to generate a different identifier, and that something would be the nonce.

A smart contract wallet may decide to enforce strictly increasing nonces or not, depending on whether they want to implement some form of ordering. Ethereum-like nonces could be implemented by consuming a note with the previous nonce, and inserting a new one with the current one.

Transaction hashes

A tx hash should act as a unique identifier that’s known by the time the tx is sent to the P2P pool (see initial discussion here). Given a full AA model, the fields of a tx request would just be:

- Sender (the smart contract wallet address that processes this tx)
- Payload (to be interpreted by the smart contract wallet, includes nonce)

An example payload that mimicks Ethereum accounts would contain a contract to call along with the function data and value to transfer, plus a nonce and an ECDSA signature. But none of it has any meaning at the protocol level.

Now, we can use the hash of the sender and payload as the tx hash. The private kernel circuit is responsible for verifying the hash. We can then either output the hash as part of the tx object that is sent to the P2P pool, or we can follow the current approach and store it as the first nullifier. Using the latter has the benefit that we get replay protection for free, and don’t need to implement a separate data structure just to tracked processed tx hashes.