I wanted to share a brief overview of what we're building for the new contract deployment scheme for the Aztec Network, most of it based on this thread and outlined on the yellow paper.

Classes and instances

A contract in Aztec will no longer have code of its own. Instead, a contract (or contract instance

for clarity) will reference a contract class

. Contract classes are just code, referenced by an identifier. Contract classes do not hold any state, and cannot be called. Instead, users interact with instances of each class, at a given address

. We expect that this separation at the protocol level will make code reuse easier.

Identifiers as commitments

A contract class identifier

is the hash of (ie a commitment to) its code and verification keys. Likewise, a contract address

(which is the identifier of a contract instance) is a commitment to its contract class, deployer address, public keys, initialization arguments, etc.

No deployment needed for private calls

Having identifiers as commitments

means that, given a contract address, we can prove what functions it implements. So, when executing a private

function in a contract, the user (via the private kernel circuit) just needs to prove that:

1- The verification key used for the private function is in the preimage of a contract class C

2- The contract class C

is in the preimage of the address being run

This means that we do not need to actually "deploy" a contract in order to call a private function in it, since the address of the contract is already a commitment to its code. This allows us to get rid of the contract tree.

Broadcasting deployments via ClassRegisterer and Deployer contracts

However, for executing public

functions, we do need to make sure all nodes have access to the code that needs to be run. Otherwise, we risk having transactions with public calls that not all nodes know how to run. So, in order to run a public function, we require the contract instance and its class to have been broadcasted to the network.

This "broadcasting" today is implemented in-protocol, by having special "contract deployment" transactions. We will be simplifying this by moving this logic up in the stack: instead of having special deployment transactions, we'll have special "Registerer" and "Deployer" contracts, whose only responsibility will be to:

1- Broadcast via unencrypted events a contract class or a contract instance

2- Emit a nullifier to signal that they have been broadcasted

The nullifier allows an Aztec node (via its public kernel circuit) to prove that the code for a given address is known or not. If it is not known, the sequencer can generate a nullifier non-membership proof to provably revert the associated tx.

Initialization at the application circuit

Constructors will be removed from the protocol level as well. Instead, initializing a contract will be just calling a function that takes care of setting everything up, and then emits an "initialization nullifier" so the contract cannot be initialized again. All other functions in a contract should check that the initialization nullifier has been emitted in order to avoid being called before

the contract is initialized. The plan is to handle all this logic under the hood using macros.

Removing constructors from the protocol gives us more flexibility on how to initialize a contract. Now we can have both private and public constructors, or multiple constructor functions per contract, or call a constructor from another contract, or initialize multiple contracts in the same tx, or not

have a constructor at all if we don’t need it, or have functions that don’t require the contract to be initialized in order to be executed.

Summing up

- Contracts are split into classes and instances, where a class may have multiple instances.
- You can call a private function in any contract without having to deploy it, allowing fully private contracts between small parties.
- But you need to “broadcast” it to call public functions on it.
- Initializing a contract is just calling a function that can be called only once.