Diversified accounts

This post presents two proposals to implement diversified accounts, a concept bridged from zcash's diversified addresses for generating new accounts bound to the same encryption private key, so that generating new addresses doesn't incur in a higher effort for note discovery.

A diversified account is an account that uses a diversified public key. We outlinehere a mechanism, borrowed from zcash, for deriving a diversified public key from a master public key. The goal of diversified accounts in the context of the Aztec network is to support these use cases:

1. Calling from private to public without disclosing the user's address
2. Giving different addresses to different counterparties so they don't know they are interacting with the same account

Note that, if a contract calls into a diversified address, the call cannot be visibly routed back to the master account contract, otherwise it would defeat the purpose of diversification.

A trivial implementation for diversified accounts is to deploy one account contract for each of them, using a diversified public key for each. This is moderately expensive, since the user needs to pay the cost for deploying each account contract, though since they will all share the same class identifier the deployment of multiple instances should not be too costly.

We present here two proposals, as discussed with@Mike. We could implement neither, one, or both of them - all options make sense.

Diversified calls

We add a new type of call, a diversified call

, where the caller can alter the msg_sender

seen by the target of the call. The caller needs to include a proof that the impersonated account can be counterfactually deployed by them. Recall from here that an address is the hash of several arguments, including the deployer. The caller then needs to provide the entire preimage to the address they are impersonating, such that their address is equal to the deployer. We would expect account contracts to use diversified calls in their entrypoint, while apps should use regular calls in their workflows.

This mechanism allows a user to impersonate, as they call into different apps, as accounts they could counterfactually deploy. For most use cases, they should be able to claim the assets they've received just using diversified calls. However, should an app need to actually call into the account contract, for instance to run an authwit query, the user can then actually deploy the contract to unblock it.

Note that this approach breaks the assumption that, if a contract receives a call from an address, that address has a contract that has been deployed to. This doesn't seem to be a problem though, but it's worth highlighting.

Undeployed contracts

We add support for interacting with a contract before it has been deployed

. Assuming a contract has no constructor to execute, then the act of deploying the contract is meaningless (except for broadcasting the class identifier), since its code is bound to by the class identifier which is part of the address preimage. Then, if we allow a contract to be interacted with without deploying it, we can implement the trivial solution for diversified accounts of deploying a new account contract for each of them without actually paying the deployment cost

.

Again, this approach breaks the same assumption as the one above, in that a contract may receive a call from an address that has not broadcasted its class identifier, so it cannot be called into. Still, this approach may unlock new use cases where users run full private protocols without ever having to deploy a single contract. It is possible we can also implement stealth addresses with this same primitive.

A diversified account contract can be implemented without a constructor by simply checking that the caller is the master account. Users can call into their diversified account contract entrypoint from their master account, prove the derivation, and from there call into apps.

contract DiversifiedAccount

```
// We run whatever our master account tells us to run, as long as we can prove that we are derived from it
private fn entrypoint(payload: action[])
    let address_preimage = pxe_oracle.get_address_preimage(this.address)
```

```
    if address_preimage.deployer == msg_sender and hash(address_preimage) == this.address
        execute payload

// We delegate to our deployer, as long as we can prove that we are derived from it
private fn handle_authwit_query(request)
    let address_preimage = pxe_oracle.get_address_preimage(this.address)
    if hash(address_preimage) == this.address
        return address_preimage.deployer.handle_authwit_query(request)

// This is a public function, so it requires the contract to be actually deployed, so we can just store authwits in storage as usual
public fn handle_authwit_query_public
    // ...
```

Note that this approach requires an extra function call with respect to the previous one, since the user must hop from their master account contract to the diversified one. On the other hand, this one feels more flexible.

In order to implement undeployed contracts, we slightly modify the kernel's behaviour when verifying a function call. Today, the kernel checks that the function verification key is part of the class id, checks that the class id has been registered in the contract class tree (or the nullifier tree, pending decision), and checks that the contract address has been deployed in the contract instances tree (or the nullifier tree, pending decision). We need the kernel to skip this last check if the contract can be interacted with without deploying.

We have two options for doing so:

The simplest is that, if the class id for the contract have no constructor, then we assume the contract does not require deployment and the kernel skips the deployment check in that scenario.

The fanciest one is abstracting constructors altogether and remove the concept of constructor from the protocol. Instead, contracts may declare an initialization function, which is a regular function that emits a singleton nullifier when called. All other functions in the contract then assert that the initialization nullifier has been emitted. This check can be embedded into an aztec-nr macro, unless the dev explicitly opts-out of it, signalling that a given function can be called without

waiting for the contract to be initialized. This feels more flexible, but also more of a potential footgun. Furthermore, if we keep the constructor check in the kernel, we could optimize it so we do the check only once per contract in a transaction, which we cannot easily do if the check is inlined in every function. Still, this approach is interesting as it further simplifies the protocol.