TLDR: Reserve N bits of an address, or add a new input to the address preimages, to store forward-compatible metadata for an account.

One of the solutions for supporting contract upgrades requires binding all transactions that execute an upgradeable contract to the epochs of the slow updates tree, which may hurt UX. It follows that, if a contract is not upgradeable, then we shouldn't have to worry about slow update epochs. But how can we check whether a contract is upgradeable or not?

We can define an upgradeable

flag that needs to be set at deployment time in order for a contract to be upgradeable. If the flag is not set, then the kernel would refuse to execute an upgrade. This flag can be stored either 1) as a specific bit in the address, or 2) as part of a new metadata field that is part of the address preimage.

Given it's likely that we'll run into more "flags" for an account (such as related to encryption or note discovery mechanisms they support), we can expand this flag to a full byte (or larger!) so we can encode new settings as the protocol evolves.

As a reminder, addresses in Aztec today are calculated as:

partial_address := hash(salt, contract_code, constructor_hash) address := hash(public_key, partial_address) complete_address = { address, partial_address, public_key }

So, for option 1, where we encode the metadata field within the address, we'd redefine address as:

partial_address := hash(salt, contract_code, constructor_hash) address := metadata ++ slice(hash(public_key, partial_address), FIELD_LENGTH - METADATA_LENGTH)

Where ++

is concatenation. This has the nice property that an address can be immediately inspected for its metadata, but has the downside that it severely restricts the length of what metadata we can encode in it, since we don't want to go below 160 bits for the slice of the hash in order to avoid collisions (and even Ethereum is considering bumping this to a full 32 bytes).

On the other hand, option 2 adds an indirection for accessing the metadata (exactly the same as we do for accessing the public key for an account), but allows us to store a lot more information.

partial_address := hash(salt, contract_code, constructor_hash) address := hash(public_key, partial_address, metadata) complete_address = { address, partial_address, public_key, metadata }

In this model, we can even define variable-length metadata arrays, or even encode full key-value pairs within the metadata field, though all this adds complexity to circuits.