Contract upgrades can be achieved by simply providing a way for a contract to change it’s class ID. The big issue is where to store this information, since mutable public state that can be accessed with no contentionis hard.

This post describes a possible approach, along with its trade-offs and open design questions.

For a more detailed discussion the alternative upgrade mechanism, upgradeable proxies, seethis forum post.

Shared Mutable Storage

We need this type of storage to store the contract implementation, regardless of whether we enshrine upgrades or use delegatecall

. The only difference is that enshrining upgrades also means enshrining (at least some) usage of this type of storage. The enshrinement however does not change how Shared Mutable Storage works: any undesirable side effects will be present in both approaches.

A key point to look at is whether enshrining affects non-upgradeable contracts - ideally it shouldn’t, same as delegatecall

.

The current iteration of Shared Mutable Storage is codenamed SlowJoe - the rest of this document assumes some familiarity with its internals.

How To

We introduce a new opcode schedule_upgrade

to store the new contract class ID and proposed time_of_change

. This write follows the SharedMutableStorage pattern, i.e. it ensures that the change will take effect at some time in the future after some minimum delay.

We don’t need

an opcode for correct usage, but having one prevents incorrect usage: if we simply allocated a plain storage slot, it’d be very hard to detect if a contract a user interacts with performs upgrades ‘incorrectly’, and may therefore do nasty things such as change its code retroactively. An opcode also provides trivial detection of upgradeable contracts for free.

Prior to any upgrade, the class ID is derived from the address (as is done today). The Private Kernel thus performs the following to retrieve a contract class:

if class_id_storage_slot is not empty: class_id = shared_mutable_state_read(class_id_storage_slot) else: prove class_id_storage_slot is empty // needs merkle non-membership proof read class_id from address preimage prove that address preimage hashes to address

It also must constrain max_block_number

so that all shared mutable reads remain valid.[1]

Public function execution would be similar, except the address preimage would instead be loaded from the contract public deployment.

Improvements Over Proxies

- one fewer function call - we immediately execute the correct contract class
- no proxy obscurity
- no weird delegatecall semantics
- less tooling needed (to e.g. deal with obscurity, detect upgradable contracts)
- easier setup for upgradeable contracts (just call upgrade

)

Problems

- ‘Downtime’ due to shared mutable state usage, i.e. when very close to the upgrade block, users might not have enough time to produce a proof and submit a tx, and need to wait until the upgrade goes through.

- This is only an issue when interacting with a contract that has a pending upgrade, which is very rare.
- This is also currently true for proxies, since they'd use this same storage kind to store the implementation address.
- This is only an issue when interacting with a contract that has a pending upgrade, which is very rare.
- This is also currently true for proxies, since they'd use this same storage kind to store the implementation address.
- If all contracts are upgradeable then all

transactions require max_block_number

, restricting use cases with (very) long proof generation.

- Account contracts that have been upgraded leak that they are account contracts (assuming the class id would be known due to code reuse).
- With delegatecall

account contracts don't require shared mutable state (contention is a non-issue because they have a single owner), so they can afford to not leak. * Unless they have public functions, in which case they require a public delegatecall, leaking the class id again.

- They also pay for this with extra costs, since they need to emit singleton commitments and nullifiers.
- Unless they have public functions, in which case they require a public delegatecall, leaking the class id again.
- They also pay for this with extra costs, since they need to emit singleton commitments and nullifiers.
- With delegatecall

account contracts don't require shared mutable state (contention is a non-issue because they have a single owner), so they can afford to not leak. * Unless they have public functions, in which case they require a public delegatecall, leaking the class id again.

- They also pay for this with extra costs, since they need to emit singleton commitments and nullifiers.
- Unless they have public functions, in which case they require a public delegatecall, leaking the class id again.
- They also pay for this with extra costs, since they need to emit singleton commitments and nullifiers.

Questions

- Do we add an AVM opcode for upgrade that follows the rules of shared mutable writes?
- Pro: We ensure upgrades always follow the shared mutable rules.
- Detecting contracts that don't follow these rules can be very hard, and it might be dangerous to interact with them.
- Detecting contracts that don't follow these rules can be very hard, and it might be dangerous to interact with them.
- Pro: It is easier to detect if a contract is upgradeable.
- Con: More complexity in the AVM
- Pro: We ensure upgrades always follow the shared mutable rules.
- Detecting contracts that don't follow these rules can be very hard, and it might be dangerous to interact with them.
- Detecting contracts that don't follow these rules can be very hard, and it might be dangerous to interact with them.
- Pro: It is easier to detect if a contract is upgradeable.
- Con: More complexity in the AVM
- How do we allocate the storage slot for the class id?
- We can separate "application" and "system" storage, and have them go through separate opcodes. At the protocol level, we "silo" them, similar to how we silo per address, but end up writing them all to the same public state tree.
- Or we can choose the hash of a preimage that should not be generated by regular usage (ie EIP1967)
- If we use regular storage, then we'd need to prevent regular storage writes from accesing this slot, since only the

opcode should touch it.

- If we use regular storage, then we'd need to prevent regular storage writes from accesing this slot, since only the opcode should touch it.
- We can separate "application" and "system" storage, and have them go through separate opcodes. At the protocol level, we "silo" them, similar to how we silo per address, but end up writing them all to the same public state tree.
- Or we can choose the hash of a preimage that should not be generated by regular usage (ie EIP1967)
- If we use regular storage, then we'd need to prevent regular storage writes from accesing this slot, since only the opcode should touch it.
- If we use regular storage, then we'd need to prevent regular storage writes from accesing this slot, since only the opcode should touch it.
- Do we differentiate addresses between upgradeable or not at the protocol level, eg via a bit in the address or some preimage metadata flag?
- Pro: interactions with non-upgradeable contracts would not be constrained by max_block_number

.

- Pro: Any user can easily see if a contract is upgradeable or not without having to "analyze" the bytecode looking for an upgrade opcode (assuming there is one).
- Pro: We can use different kernels for interacting with upgradeable and with non upgradeable addresses. If it's non-upgradeable, then we can skip the class_id_storage_slot

non-membership check altogether.

- Pro: interactions with non-upgradeable contracts would not be constrained by max_block_number

.

- Pro: Any user can easily see if a contract is upgradeable or not without having to "analyze" the bytecode looking for an upgrade opcode (assuming there is one).
- Pro: We can use different kernels for interacting with upgradeable and with non upgradeable addresses. If it's non-upgradeable, then we can skip the class_id_storage_slot

non-membership check altogether.

- Do we enshrine delays, or make them configurable?
- We can make them configurable by storing delay D

as any other field (pre, post, time-of-change).

- Whether a mutable delay is desired in the first place is unclear. End users would limit their tx by the smallest delay of all contracts they interact with, so smaller delays leak privacy, and larger delays either do nothing or leak privacy.
- We can make them configurable by storing delay D

as any other field (pre, post, time-of-change).

- Whether a mutable delay is desired in the first place is unclear. End users would limit their tx by the smallest delay of all contracts they interact with, so smaller delays leak privacy, and larger delays either do nothing or leak privacy.
- Ideally we change this so that non-upgradeable contracts are not constrained by this.↩