TLDR

: Assuming no enshrined slow updates tree, we can separate classes and instances to mitigate the footguns of delegatecall, and have proxy-based upgrades. This requires implementing at the protocol level the separation between classes and instances, a new delegatecall opcode that targets classes (not instances like in the EVM), and fallback functions (to implement proxies).

Assuming we are not

enshrining the slow updates tree for the time being, let's explore some ideas when it comes to implementing smart contract upgrades. Of course, "no upgrades" is always an option.

Note that, if we don't have an enshrined slow updates tree, we have no enshrined way to store the current implementation in a way that's mutable, accessible from private, and doesn't cause contention issues, but we can still implement one at the application layer. This means that we need application-level code to "load" the current implementation in whatever scheme we implement.

The EVM way: delegatecall and fallback functions

We can always implement upgrades using the same building blocks as the EVM: delegatecall

opcode and fallback

(aka method_missing

, doesNotUnderstand

) functions. To build an upgradeable contract, users would have to write a proxy-like contract with a fallback

function that loads the current implementation from the slow updates tree, and then does a delegatecall

to it. Note that two different fallback functions would be needed: one for private, and one for public.

To support this, we'd need to add 1) delegatecall

support in the protocol and langauge, and 2) fallback

functions in the protocol and language. The latter requires the kernel circuit to loop over all functions in a contract when it's resolving what bytecode to run, prove that none of them matches the requested one, and only then load the fallback

function.

Now, as @rahul-kothari has perceptively noticed some time ago, I'm not a fan of this solution:

delegatecall

has been behind numerous security incidents, from the infamous parity wallet multisig hacks from 2017, to the recent Raft hack in which the hacker themself lost their loot due to misuse of this opcode. It also makes it difficult to write reusable code: library authors need to maintain forks of their code with the necessary hacks to support upgradeability (eg no constructors, no default initializers, etc).

I'd argue that the reason behind delegatecall

becoming such a footgun is that it overloads what a contract is

, making it hard to reason about your own code. A contract can either be an instance

, in that it's a contract you interact with and has its own state, or a class

, in that it's a bunch of stateless code that gets executed from an instance, or anything in between. But to the EVM and the language, it's the same construct.

So why don't we enshrine this difference?

Contract instances with immutable class

We can split what we currently call a contract into a contract instance and a class, like Starknet does. An address would refer to a contract instance

, that has its own state as well as a pointer to a class

. The class is just a bunch of immutable bytecode. When a user calls into an instance, the code from the instance's class is loaded. Instances do not have any code of their own, they just point to a class. This makes it easier to reason about what a contract is, and gives us the gas-savings of minimal proxies (aka clones) for free.

Our original proposal for classes and instances suggested including a CODEREPLACE opcode that would change the class of an instance, like RSK or Starknet. However, this would require storing the pointer to the class in slow-updates storage, and since it's not enshrined, this is no longer an option. So we'll assume that the class of a contract instance cannot be changed.

Nevertheless, we can still use the classes-instances pattern to mitigate the issues with delegatecall

. We can keep the delegatecall opcode, only that instead of calling into an instance, we call into a different class

. This keeps the healthy separation between instances and code, while allowing to dynamically load different code.

In this scenario, an upgradeable contract would be an instance with a proxy as contract class. This class would delegatecall into an "implementation" contract class loaded from an app-layer slow-updates tree. Note that, in contracts where contention is not an issue (such as account contracts), the implementation class can be stored in a regular private note. And in public-land, it can just be stored in public storage.

Supporting this would require introducing a new call type in the protocol, both for private and public, as well as enshrining the separation of instances and classes.

As a side question: should we still call this delegatecall

?

Script execution

: A use case for delegatecall

, aside from upgradeability, was executing scripts. An exampleraised by the community was encoding a set of actions for a governance contract to execute, and then delegatecall

ing into it to carry out those actions impersonated as governance. Keeping delegatecall

with the new semantic of calling into a contract class would still allow this use case.

Enshrined proxies vs fallback functions

Assuming we go with the design above, let's discuss how we want to implement proxies:

1. Add support for "fallback" functions in the protocol, and implement the proxy as a class with just a fallback function that loads the implementation class and delegatecalls into it. This would be the most similar to the EVM way.
2. Do not support "fallback" functions. Instead, enshrine proxies in the protocol as a type of contract class that has only a single function, that gets called regardless of the selector used. This is far more restrictive than (1), but may be more efficient since it does not need to prove that no function selector matches the requested one during a call, and it's not clear if there are any use cases for fallback functions that are not proxies.
3. Support "fallback" functions in the protocol, but do not expose them at the language level. Instead, add a "proxy" keyword that emulates the behaviour of option (2) by using option (1).

Given the performance improvement of (2) would most likely be negligible, I'd favour option (1) since it's the most flexible, and doesn't seem to lead to any security footgun.