Overview

I came up with a backward compatible extension to the EVM that allows system-wide (cross contract) opt-in extensions to the EVM to be described as EVM Bytecode itself.

This is achieved by allowing an address (contract) to have multiple (state, code) pairs associated instead of just one.

This allows creating contracts by composing existing or new handcrafted components.

A caller can specify the targeted component (by its code hash) and a called component knows the calling component (by its code hash).

This allows embedding components in a contract that provide a subsystem inside the EVM. Think of these components as trusted islands inside a contract or as trusted communication endpoints in communication partners.

The components system can further help to mitigate other problems like reentrancy, code injection and unchecked cast attacks

They further provide an alternative to the central registry pattern for asset management and instead store them in a component in the owner's contract

Multi-Component Contracts

The goal of this idea is to provide a way where a contract is a composition of smaller components where explicit communication between components is possible.

This first part is a high-level description.

Unlike contracts which are addressed over a global identifier unique for each instance, components are addressed over their functionality or more precisely over their code hash.

Each component can be instantiated more than once and every instance is part of a contract and can uniquely be identified over the address of the contract combined with the components code hash.

A contract can consist of multiple component instances but cannot have more than one instance of a specific component.

Every contract has one default component which is called when the existing call functionality is used. This provides backwards compatibility and a way todo dynamic dispatch.

A new call variant is added that does not only take an address to identify the target but does additionally take a code hash identifying the targeted component.

When a component is called the code hash (identity) of the calling component is available in addition to the calling contracts address.

Each component instance can have its own state. A component instance knows the address of the contract it belongs to as well as its own code hash.

A component that does a selfdestruct does not destroy the contract but only the component, except if it was the last component.

The default component can only be destroyed if it is the last component.

Optionally a contract can have one of its components dedicated as management component.

The management component can do some additional operations.

First, it can create change the default and management component. In case of the management component, it can even remove it.

Second, it can instantiate new components into the contract.

When a new contract is created the code returned by the init code becomes the only component and is marked as default and management component.

The init code is executed in the context of this component (has access to its state and can create other components).

The create component operation works like the create contract operation with the difference that the initcode operates on the resulting component which neither treated as default nor as management component.

The created component is added to the same contract the managing component creating it belongs to.

Rich User Accounts

Each User account is represented as a contract that initially has one virtual component, which is the default and management component of that account/contract.

Transaction signed by the private key corresponding to the account would origin from that component.

The user can use the management capability of this component to instantiate more components in that account.

This gives private key controlled account/contracts the same power as code controlled contracts have now.

Some Use Cases

Interfaces

A component could serve as an interface which forwards its calls to one or many other components in the same contract. One component (initially its creator) would be allowed to change the mappings.

Contract side Assets

A component that would track the asset owned by the contract and would allow to transfer it to other contracts by directly contacting an instance of the same component as itself in the other contract.

Capabilities

A component tracks other components to which this address should have access to. A call must be made through the capability component or will be refused by the target component (which must be aware that it is part of a capability system). The capability component blocks unauthorized outgoing calls. The capability component allows transferring capabilities to other addresses (using capabilities to that contract)

Private Components

Components that only accept calls from other components in the same contract.

Access Control

Components that manage an ACL and if a call is permitted forward it to a private component else block it (would be similarly managed as an interface)

Shared Databases

Components that manage a shared database but provide a richer interface than the key-value store to the other components in the same contract (could even allow read access to other contracts)

Meta-Data

A component that tracks code reviews and other metadata concerning the contract

Wallet Plugins

A user account contract could install plugins in form of new components adding personal on-chain functionality to its account.

Static Calls

Calls, where a certain code is expected to handle the call, can be done safely and it is ensured that the expected code is executed (as the component is identified by its code hash)

Runtime Casts

Language that uses contract types/classes can represent the involved classes as separate components which allows eliminates the risk that after a cast a contract is called that actually is not of the expected type (solidity currently has that problem)

Code Management Optimisations (Optional)

Components provide the most benefit if the same component is instantiated multiple times (reshared).

The current Create opcode, on the other hand, is optimized for contracts/components with unshared unique code.

New opcodes that separate the instalment of the components code from its instantiation would help.

A first opcode would take the code as input and makes an entry in a new global state that maps code hashes to the corresponding code.

The second pair of opcodes would work as the Contract and Component creating opcodes but the initcode would have to return a code hash of a registered component.

The old Create opcodes would implicitly register the returned code (if not registered already) and serve as a combination of the two.