Versionless Ethereum Virtual Machine

Redesign the virtual machine for the future.

Wei Tang

August 21, 2019

Contents

If we have the opportunity to redesign the Ethereum Virtual Machine, improving its backward compatibility, making it friendlier to feature upgrades, and completely disregarding all the history burdens we have, what will we do? In this article, we try to explore on the question, and document the evolved specifications and rationales.

Objectives

The web has been versionless, and has existed for decades. Let's assume in this article that we want to make EVM versionless as well.

We also want to make sure the VM has the necessary properties that we maintain good backward compatibility. That is, at least better compared with what we do now, where even any simple gas changes can break things. We also want to make sure that we can easily add new features into the VM.

Invalid Opcodes

The simplest, and probably the most important change we need to do for a versionless EVM is to add a validation process for contract deployment. Not all byte sequence should be valid EVM code. Any invalid opcode just should not ever be deployed, because in the future they may be assigned as a new opcode, which has different functionalities.

The exact specification for this check was originally documented in EIP-1712. Basically, before executing a contract creation transaction, or adding contract code to the state, do the following check:

- Iterate over the code bytes one by one.
- If the code byte is a PUSH(n) opcode, skip next n bytes.
- If the code byte is a valid opcode or designated invalid instruction (`0xfe`), continue.
- Otherwise, trap.

The above is similar to jump destination checks. Note here that we used the term "trap" for exception, which we will explain in more details in later sections.

Feature Probe

If EVM is going to be versionless, code executing on top of it should be able to probe whether a particular feature is supported. Given the nature of EVM, we always want the function of a defined opcode to remain unchanged, and introduce new opcodes for additional functionalities. Some contracts might be deployed before a particular feature is introduced. They may have a backup routine that executes when a feature is not supported, and want to switch to immediately use the new feature once the hard fork is happened. The feature probe will act like the switch.

Formally, we define an new opcode `HAS_FEATURE`.

- The opcode takes in one stack parameter. It checks whether the parameter is within range `0` to `2^8`. If not, trap.
- It then push `0x0` back to stack if the parameter, interpreted as an opcode, is not supported, and `0x1` otherwise.

Exceptions and Trap

EVM currently has many ways where things can fail. An individual transaction can fail out-of-gas. Each of the internal callstack can also fail itself, whose errors must be handled explictly by the parent callframe. This gives flexibility, but for something that runs on a blockchain, it's not necessily a good thing. Here instead, we redefine any exceptions, emitted by EVM itself, to have the *trap* behavior. That is, it immediately exits all executions of all callframes, consumes all gases, and reverts any changes made by the current transaction. Contracts are then encouraged to use their return values if they want to communicate non-fatal errors to parent callers.

Gas Cost

Past experience has shown us that we want to modify the gas cost a lot. Because we do this, we do not want contract developers to make **any** assumptions on transactions' current gas level, or any opcode's gas cost. To do this, we just remove all expose of gas cost information inside EVM. This makes gas cost an "implementation detail" that is hidden outside of EVM, and is only needed to be taken care of in block-level execution.

Formally, remove opcode `GAS` at `0x5a`. In addition, redefine `CALL`, `CALLCODE` and `DELEGATECALL` not to take the *gas* stack parameter, but instead take the full available gas of the current execution frame.