EEICALL

: Opcode to execute bytecode over given EEI

tl;dr EEICALL

would “close the loop” by providing a clean general meta-eval for EVM. This has big implications for rollups and opens up a big design space for EVM userspace.

This is not related to the prior meta-eval threads suggested by the dupe checker (one, two) – those are fundamentally about inter-shard proof structures, while this is about a generic EEI “eval”

.

Background: What is the EEI

Ethereum’s “Execution Environment Interface” is the virtual interface that the EVM bytecode interacts with.

Roughly speaking, an EVM bytecode interpreter maps some opcodes onto function calls into the EEI implemented in the outer language.

As an example, if you look inside your favorite ethereum client, you might find:

SLOAD

implementation calls eei.readStorage

CALLER

implementation calls eei.getSender

BLOCKNUM

implementation calls eei.getBlockNumber

MINER

implementation calls eei.getMiner

You can group these into ‘local contract context’ and ‘other ethereum context’. One way you could think of CALL

vs DELEGATECALL

is that the caller provides the sub-call context with different behaviors for the ‘local contract context’ methods (but not ‘other ethereum context’) in the EEI it passes in.

Motivation

One concrete use case for executing code with respect to a given EEI is fraud proofs for rollups. A fraud proof for a rollup ultimately reveals an invalid EVM state transition. In-EVM fraud proof checkers get a lot of mileage out of DELEGATECALL

, but without proper meta-eval, these strategies require a myriad of compromises, like code transpilers or involved interactive proofs.

Explicitly providing an EEI to EVM code

We want a similar opcode to DELEGATECALL

, except that it replaces the entire context

. The EEI would be provided in the form of a contract that implements the EEI interface.

EEICALL

takes two

addresses, and interprets the first as the EEI for the code of the second.

EEICALL(address eei, address code)

.

This will execute the bytecode at address code

, but each opcode that interacts with the EEI will instead cause a call into the given eei

contract. For example, if the code of the code

contract uses SSTORE

, EEICALL(eei, code)

would instead result in eei.setStorage

getting called.

Note that the given EEI

address can be the caller, ie, EEICALL(this, code)

might reasonably be used from inside a fraud proof checker contract. (In theory code

does not need to be provided at all, as the EEI exposes getCode

… This is an ‘abstract origin’ perspective. It could instead just be ‘calldata’, or even just have everything be explicit on the EEI).

CALL

, DELEGATECALL

, STATICCALL

are also

mapped onto the EEI (e.g. eei.executeCall

), and can also

be overriden. You can imagine wanting to override CALL

to be able to add new precompiles to your verifier. You can also route calls addresses in code to different addresses in the outer EVM environment.

Gas, partial evaluation, continuation

EEICALL

introduces a gas overhead to any call it is emulating, because the given EEI contract consumes gas when implementing the EEI methods.

For example, SSTORE

consumes only 20k gas in the ‘inner’ evaluation, but performing this evaluation required the outer context to process the call to setStorage

.

Most EEI method implementations would be constant time and fast, but it would still be some constant multiple overhead. This suggests that to properly emulate calls, the outer context needs a way to pause and continue partial evaluations. A paused EEICALL

needs a serialized form. This serialized object is an EVM continuation.

Plan of attack

- Do a survey of EEI definitions in different languages. Filter fundamental parts of the EEI vs extraneous methods that should live elsewhere.
- Gather everyone working on EVM fraud proofs for rollups, or any other EVM virtualization / meta-evaluation.
- Decide on minimum viable EEICALL. In particular, do we care to be able to pause/continue evaluations. If so, tackle that design.

- Find the client team most willing to refactor their EVM+EEI interface to theoretically enable EEICALL

.

- Make a proper EIP with a formal spec.