What might an "enshrined ZK-EVM" look like?

Layer-2 EVM protocols on top of Ethereum, including optimistic rollups and ZK rollups, depend on EVM verification. However, this requires them to trust a large codebase, and if there is a bug in that codebase, these VMs risk getting hacked. Additionally, it means even ZK-EVMs that want to stay *exactly* equivalent to the L1 EVM need to have some form of governance, to copy changes to the L1 EVM into their own EVM implementation.

This situation is suboptimal, because these projects are replicating a functionality that already exists in the Ethereum protocol, and where Ethereum governance is already responsible for making upgrades and fixing bugs: a ZK-EVM basically does the same work as verifying layer-1 Ethereum blocks! Furthermore, over the next few years, we expect light clients to get more and more powerful, and pretty soon get to the point of using ZK-SNARKs to fully verify L1 EVM execution. At that point, the Ethereum network will effectively have a built-in ZK-EVM. And so the question arises: why not make that ZK-EVM natively available for rollups too?

This post will describe a few versions of "an enshrined ZK-EVM" that could be implemented, and go through the tradeoffs and design challenges, and reasons not to go in particular directions. The upsides of implementing a protocol feature should be weighed against the benefits of leaving things to the ecosystem and keeping the base protocol simple.

What are some key properties that we want out of the enshrined ZK-EVM?

- **Basic functionality: verify Ethereum blocks**. The protocol feature (so far leaving it open whether it's an opcode, precompile or other mechanism) should accept as input (at least) a pre-state root, a block, and a post-state root, and verify that the post-state root actually is the result of executing the block on top of the pre-state root.
- **Compatibility with Ethereum's multi-client philosophy**. This means that we want to avoid enshrining a single proving system, and instead allow different clients to use different proving systems. This in turn implies a few points:
  - **Data availability requirement**: for any EVM execution that gets proven with the enshrined ZK-EVM, we want a guarantee that the underlying data is *available*, so that provers using a different proving system can re-prove the execution and clients relying on that proving system can verify those newly generated proofs.
  - **Proofs live outside the EVM and block data structure**: the ZK-EVM feature would not literally take a SNARK as an in-EVM input, as different clients would expect SNARKs of different types. Instead, it might work similarly to blob verification: transactions could include (pre-state, block body, post-state) statements that need to be proven, an opcode or precompile could access the content of those statements, and the client consensus rules would separately check data availability and presence of a proof for each claim that is being made in the block.
- **Auditability.** If any execution gets proven, we want the underlying data to be available, so that if anything goes wrong, users and developers can inspect it. In practice, this adds one more reason why a data availability requirement is important.
- **Upgradeability.** If a particular ZK-EVM scheme is found to have a bug, we want to be able to fix it quickly. This implies not requiring a hard fork to fix. This adds one more reason why proofs living outside the EVM and block data structure is important.
- **Supporting almost-EVMs**. Part of the attraction of L2s is the ability to innovate on the execution layer, and make extensions to the EVM. If a given L2's VM differs from the EVM only a little bit, it would be nice if the L2 could still use a native in-protocol ZK-EVM for the parts that are identical to the EVM, and only rely on their own code for the parts that are different. This could be done by designing the ZK-EVM feature in such a way that it allows the caller to specify a bitfield or list of opcodes or addresses that get handled by an externally supplied table instead of the EVM itself. We could also make gas costs open to customization to a limited extent.

"Open" vs "closed" multi-client systems

The "multi-client philosophy" is probably the most opinionated requirement in this list There is the option of abandoning it and focusing on one ZK-SNARK scheme, which would simplify the design, but at the cost of being a much greater "philosophical pivot" for Ethereum (as it would de-facto be the abandonment of Ethereum's long-standing multi-client philosophy) and at the cost of introducing greater risks. In the long-term future, where eg. formal verification technology becomes much better, it might be better to go down this road, but for now the risks seem too great.

Another option is a **closed multi-client system**, where there is a *fixed* set of proof systems that is known within the protocol. For example, we might decide that we use three ZK-EVMs: the PSE ZK-EVM, the Polygon ZK-EVM and Kakarot. A block would need to come with proofs from two of these three to be valid. This is better than a single proof system, but it makes the system less adaptable because users would have to maintain verifiers for each proof system that exists, there would be an inevitably political governance process for incorporating new proof systems, etc.

This motivates my preference for an **open multi-client system**, where proofs are placed "outside the block" and verified by clients separately. Individual users would use whatever client they want to verify blocks, and they would be able to do so as long as there is at least one prover creating proofs for that proof systems. Proof systems would gain influence by convincing users to run them, and not by convincing the protocol governance process. However, this approach does have more complexity costs, as we will see.

What key properties do we want out of ZK-EVM implementations?

The most important property, in addition to basic guarantees of correct functionality and security, is **speed**. While it is possible to design an in-protocol ZK-EVM feature that is asynchronous, returning an answer for each claim only after a delay of N slots, the problem becomes much easier if we can reliably guarantee that a proof can be generated within a few seconds, so whatever happens in each block is self-contained.

While today generating a proof for an Ethereum block takes many minutes or hours, we know that there is no theoretical reason preventing massive parallelization: we can always put together enough GPUs to separately prove different parts of a block execution, and then use recursive SNARKs to put the proofs together. Additionally, hardware acceleration through FPGAs and ASICs could help optimize proving further. However, actually getting to this point is a significant engineering challenge that should not be underestimated.

What might an in-protocol ZK-EVM feature concretely look like?

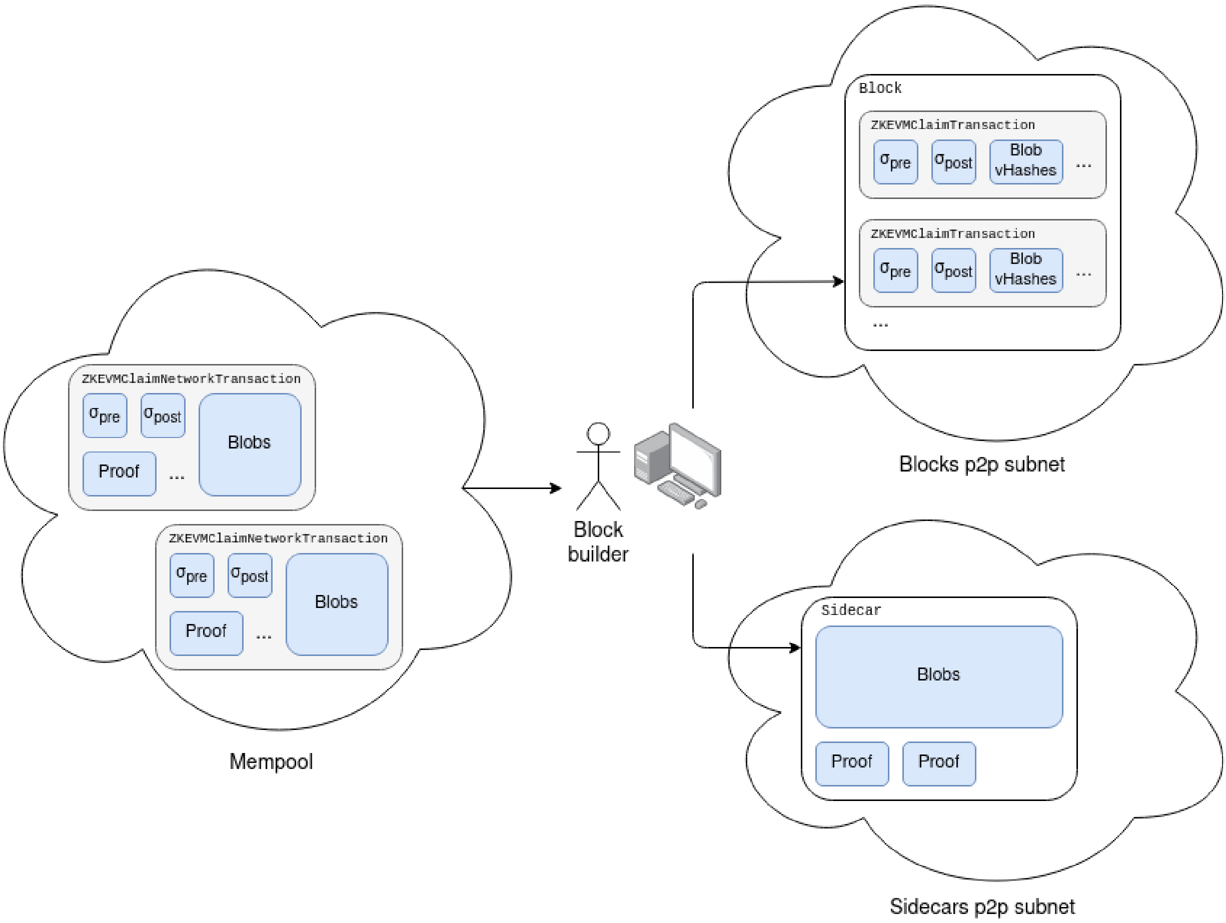
Similarly to EIP-4844 blob transactions, we introduce a new transaction type that contains ZK-EVM claims:

```python
class ZKEVMClaimTransaction(Container):

pre_state_root: bytes32

post_state_root: bytes32

transaction_and_witness_blob_pointers: List[VersionedHash]

...
```

Like in EIP-4844, the object that gets passed around in the mempool would be a modified version of the transaction:

```python
class ZKEvmClaimNetworkTransaction(Container):

pre_state_root: bytes32

post_state_root: bytes32

proof: bytes

transaction_and_witness_blobs: List[Bytes[FIELD_ELEMENTS_PER_BLOB * 31]]
```

The latter can be converted into the former, but not the other way around. We also extend the block sidecar object (introduced in EIP-4844) to contain a list of proofs for the claims being made in the block.



Note that in practice, we may well want to split the sidecar up into two separate sidecars, one for blobs and one for proofs, and have a separate subnet for each type of proof (plus an additional subnet for the blobs).

On the consensus layer, we add a validation rule that a block is only accepted once the client has seen a valid proof for each claim in the block. The proof must be a ZK-SNARK proving the claim that the concatenation of transaction_and_witness_blobs is a serialization of a (Block, Witness) pair, and executing the block on top of the pre_state_root using the Witness (i) is valid, and (ii) outputs the correct post_state_root. Potentially, clients could choose to wait for an M-of-N of multiple types of proof.

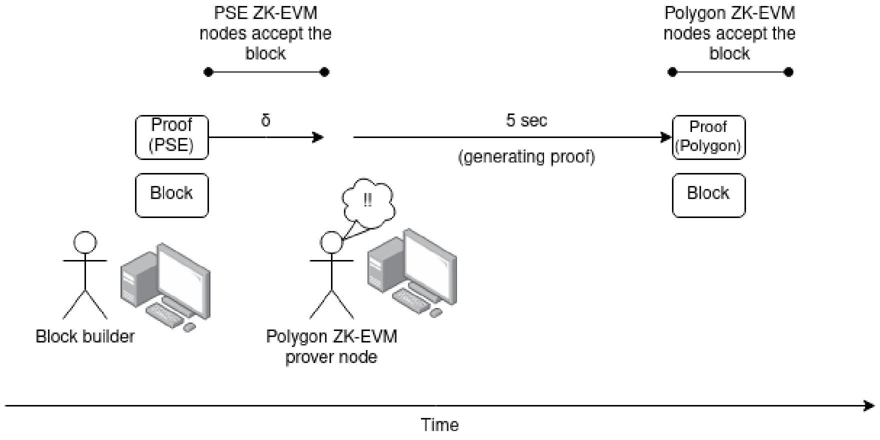
One philosophical note here is that the block execution itself could be treated as simply being one of the ($\sigma_{pre}, \sigma_{post}, Proof$) triplets that needs to be checked along with the triplets provided in the ZKEVMClaimTransaction objects. Hence, a user's ZK-EVM implementation could replace their execution client; execution clients would still be used by (i) provers and block builders, and (ii) nodes that care about indexing and storing data for local use.

Validation and re-proving

Suppose that there are two Ethereum clients, one of which uses the PSE ZK-EVM and the other which uses the Polygon ZK-EVM. Assume that by this point, both implementations have advanced to the point where they can prove an Ethereum block execution within 5 seconds, and for each proving system there exist sufficiently many independent volunteers running hardware to generate the proof.

Unfortunately, because individual proof systems are not enshrined, they cannot be incentivized in-protocol; however, we expect the cost of running provers to be low compared to the costs of research and development, and so we can simply fund provers using general-purpose institutions for public goods funding.

Suppose that someone publishes a ZKEvmClaimNetworkTransaction, except they *only* publish a version of the proof for the PSE ZK-EVM. A prover node of the Polygon ZK-EVM sees this, and computes and republishes the object with a proof for the Polygon ZK-EVM.


PSE ZK-EVM nodes accept the block
Polygon ZK-EVM nodes accept the block
Proof (PSE)
δ
5 sec
(generating proof)
Proof (Polygon)
Block
!!
Block
Block builder
Polygon ZK-EVM prover node
Time

This increases the total maximum latency between the earliest honest node accepting a block and the latest honest node accepting the same block from δ to $2\delta + T_{prove}$ (assuming here $T_{prove} < 5s$).

The good news, however, is that **if we adopt single slot finality, we can almost certainly "pipeline" this extra delay together with the multi-round consensus delays inherent in SSF.** For example, in this 4 sub-slot proposal, the "head vote" step might only require checking basic block validity, but then the "freeze and acknowledge" step would require presence of a proof.

Extension: support "almost-EVMs"

One desirable goal for a ZK-EVM feature is to support "almost-EVMs": EVMs that have a few extra features built in. This could include new precompiles, new opcodes, the option for contracts to be written in either the EVM or a totally different VM (eg. like in Arbitrum Stylus), or even multiple parallel EVMs with synchronous cross-communication.

Some modifications can be supported in a simple way: we could define a language that allows the ZKEVMClaimTransaction to pass along a full description of modified EVM rules. This could be done for:

- Custom gas cost tables (users would not be allowed to *decrease* gas costs, but they could *increase* them)
- Disabling certain opcodes
- Setting the block number (which would imply different rules depending on the hard fork)
- Setting a flag that activates a whole set of EVM changes that have been standardized for L2 use but not L1 use, or other simpler changes

To allow users to add new features in a much more open-ended way by introducing new precompiles (or opcodes), we can add a *precompile input/output transcript* included as part of the blob in the ZKEVMClaimNetworkTransaction:

```python

class PrecompileInputOutputTranscript(Container):

used_precompile_addresses: List[Address]
```

```
inputs_commitments: List[VersionedHash]

outputs: List[Bytes]
```

EVM execution would be modified as follows. An arrayinputs would be initialized to empty. The i'th time that an address in used_precompile_addresses is called, we append to inputs an objectInputsRecord(callee_address, gas, input_calldata) and we set the call's RETURNDATA to outputs[i]. At the end, we check that theused_precompile_addresses were, altogether, called len(outputs) times, and that inputs_commitments matches the result of generating blob commitments to an SSZ serialization ofinputs. The purpose of exposing the inputs_commitments is to make it easy for an external SNARK to prove the relationship between the inputs and the outputs.

Notice the asymmetry between inputs, which get stored in a hash, and outputs, which get stored in bytes that must be made available. This is because the execution needs to be doable by clients that only see the input and understand the EVM. EVM execution already generates the input for them, so they only need to check that the generated inputs match the claimed inputs, which only requires a hash check. However, the outputs must be provided to them in full, and so must be data-available.

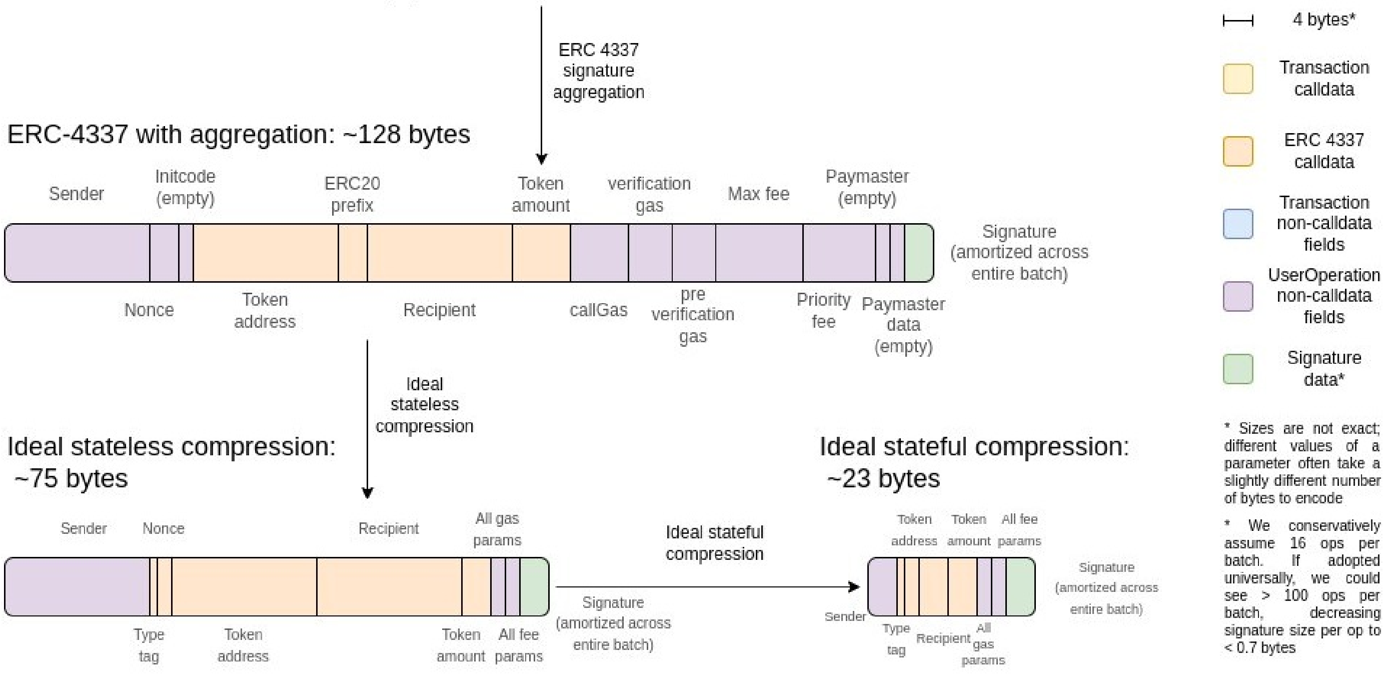
Another useful feature might be to allow "privileged transactions" that make calls from arbitrary sender accounts. These transactions could be run either in between two other transactions, or *during another (possibly also privileged) transaction*while a precompile is being called. This could be used to allow non-EVM mechanisms to themslves call back into the EVM.

This design could be modified to support new or modified opcodes in addition to new or modified precompiles. Even with just precompiles, this design is quite powerful. For example:

- By setting used_precompile_addresses to include a list of *regular account addresses* that have some flag set in their account object in the state, and making a SNARK to prove that it was constructed correctly, you could support an Arbitrum Stylus-style feature where contracts could have their code written in either EVM or WASM (or another VM). Privileged transactions could be used to allow WASM accounts to call back into the EVM.
- By adding an external check that the input/output transcripts and privileged transactions of multiple EVM executions match in the right way, you could prove a parallel system of multiple EVMs that talk to each other through a synchronous channel.
- A type 4 ZK-EVM could operate by having multiple implementations: one that converts Solidity or another higher-level language to a SNARK-friendly VM directly, and another that compiles it to EVM code and executes in the enshrined ZK-EVM. The second (inevitably slower) implementation could only be run in the case where a fault-prover sends a transaction asserting that there is a bug, collecting a bounty if they can provide a transaction that the two treat differently.
- A purely asynchronous VM could be implemented by making all calls return zero and mapping calls to privileged transactions added to the end of the block.

Extension: support stateful provers

One challenge with the above design is that it is fully stateless, which makes it data-inefficient. With ideal data compression, an ERC20 send can be up to 3x more space-efficient with stateful compression versus stateless compression only.



* Sizes are not exact; different values of a parameter often take a slightly different number of bytes to encode

* We conservatively assume 16 ops per batch. If adopted universally, we could see > 100 ops per batch, decreasing signature size per op to < 0.7 bytes

In addition to this, a stateful EVM would not need to make witness data available.**In both cases, the principle is the same: it is a waste to require data to be available, when we already know that that data is available because it was inputted in, or produced**

by, previous executions of the EVM.

If we want to make the ZK-EVM feature stateful, then we have two options:

1. Require σ_{pre} to either be empty, or be a data-available list of pre-declared keys and values, or be the σ_{post} of some previous execution.
2. Add a blob commitment to a receipt R generated by the block to the $(\sigma_{pre}, \sigma_{post}, Proof)$ tuple. Any previously generated or used blob commitments, including those representing blocks, witnesses, receipts or even regular EIP-4844 blob transactions, perhaps with some time limit, could be referenced in the `ZKEVMClaimTransaction` and accessed during its execution (perhaps through a series of instructions: "insert bytes `N...N+k-1` of commitment i at position j of the block+witness data")

(1) is basically saying: rather than enshrining stateless EVM verification, we would enshrine*EVM sub-chains*. (2) is essentially creating a minimal built-in stateful compression algorithm that uses previously-used or generated blobs as a dictionary. Both put a burden on prover nodes, and only prover nodes, to store more information; in case (2), it is easier to make that burden time-bounded than in case (1).

Arguments for closed multi-provers and off-chain data

A closed multi-prover system, where there is a fixed number of proof systems in an M-of-N structure, avoids quite a bit of the complexity above. In particular, closed multi-prover systems do not need to worry about making sure that data is on-chain. Additionally, a closed multi-prover system would allow the ZK-EVM to prove *off-chain* execution; this makes it compatible withEVM Plasma solutions.

However, a closed multi-prover system adds governance complexity and removes auditability, and these are high costs to be weighed against these benefits.

If we enshrine ZK-EVMs and make them a protocol feature, then what is the ongoing role of "layer 2 projects"?

The EVM verification functionality, which layer 2 teams currently implement by themselves, would be handled by the protocol, but**layer 2 projects would still be responsible for many important functions**:

- **Fast pre-confirmations**: single-slot finality will likely make layer 1 slots**slower**, while layer 2 projects are already giving their users "pre-confirmations", backed by the layer 2's own security, with latency much lower than one slot. This service will continue to be purely a layer-2 responsibility.
- **MEV mitigation strategies**: this could include encrypted mempools, reputation-based sequencer selection, and other features that layer 1 is not willing to implement.
- **Extensions to the EVM**: layer 2 projects could incorporate substantial extensions to the EVM that provide significant value to their users. This includes both "almost-EVMs" and radically different approaches such Arbitrum Stylus's WASM support and the SNARK-friendly Cairo language.
- **User and developer-facing conveniences**: layer 2 teams do a lot of work attracting users and projects to their ecosystems and making them feel welcome; they are compensated for this by capturing MEV and congestion fees inside their networks. This relationship would continue.