Native rollups—superpowers from L1 execution

stateless

JustinDrake Jan 20

TLDR: This post suggests a pathway for EVM-equivalent rollups to rid themselves of security councils and other attack vectors, unlocking full Ethereum L1 security.

Credit for this post goes to the wider Ethereum R&D community. Key contributions originated from 2017, with significant incremental design unlocks over the years. A thorough design space exploration was instigated by recent zkVM engineering breakthroughs. This write-up is merely a best effort attempt to piece together a coherent design for a big idea whose time may have finally come.

Abstract: We propose an elegant and powerful `EXECUTE` precompile exposing the native L1 EVM execution engine to the application layer. A native execution rollup, or "native rollup" for short, is a rollup which uses `EXECUTE` to verify EVM state transitions for batches of user transactions. One can think of native rollups as "programmable execution shards" that wrap the precompile within a derivation function to handle extra-EVM system logic, e.g. sequencing, bridging, forced inclusion, governance.

Because the `EXECUTE` precompile is directly enforced by validators it enjoys (zk)EL client diversity and provides EVM equivalence which is by construction bug-free and forward-compatible with EVM upgrades through L1 hard forks. A form of EVM introspection like the `EXECUTE` precompile is necessary for EVM-equivalent rollups that wish to fully inherit Ethereum security. We call rollups that fully inherit Ethereum security "trustless rollups".

The `EXECUTE` precompile significantly simplifies development of EVM-equivalent rollups by removing the need for complex infrastructure—such as fraud proof games, SNARK circuits, security councils—for EVM emulation and maintenance. With `EXECUTE` one can deploy minimal native and based rollups in just a few lines of Solidity code with a simple derivation function that obviates the need for special handling of sequencing, forced inclusion, or governance.

As a cherry on top native rollups can enjoy real-time settlement without needing to worry about real-time proving, dramatically simplifying synchronous composability.

This write-up is structured in two parts, starting with a description of the proposed precompile and ending with a discussion of native rollups.

Part 1: the EXECUTE precompile

Part 1—the EXECUTE precompile

construction

The `EXECUTE` precompile takes inputs `pre_state_root`, `post_state_root`, `trace`, and `gas_used`. It returns `true` if and only if:

- `trace` is a well-formatted execution trace (e.g. a list of L2 transactions and corresponding state access proofs)
- the stateless execution of `trace` starting from `pre_state_root` ends at `post_state_root`
- the stateless execution of `trace` consumes exactly `gas_used` gas

There is an EIP-1559-style mechanism to meter and price cumulative gas consumed across all `EXECUTE` calls in an L1 block. In particular, there is a cumulative gas limit `EXECUTE_CUMULATIVE_GAS_LIMIT`, and a cumulative gas target `EXECUTE_CUMULATIVE_GAS_TARGET`. (The cumulative limit and target could be merged with the L1 EIP-1559 mechanism when the L1 EVM is statelessly enforceable by validators.)

Calling the precompile costs a fixed amount of L1 gas, `EXECUTE_GAS_COST`, plus `gas_used * gas_price` where `gas_price` (denominated in ETH/gas) is set by the EIP-1559-style mechanism. Full upfront payment is drawn, even when the precompile returns `false`.

The `trace` must point to available Ethereum data from calldata, blobs, state, or memory.

enforcement by re-execution

If `EXECUTE_CUMULATIVE_GAS_LIMIT` is small enough validators can naively re-execute `trace`s to enforce correctness of `EXECUTE` calls. An initial deployment of the precompile based on re-execution could serve as a stepping stone, similar to what proto-danksharding with naive re-downloading of blobs is to full danksharding. Notice that naive re-execution imposes no state growth or bandwidth overhead for validators, and any execution overhead is parallelisable across CPU cores.

Validators must hold an explicit copy of `trace` for re-execution, preventing the use of pointers to blob data which are sampled (not downloaded) via DAS. Notice that optimistic native rollups may still post rollup data in blobs, falling back to calldata only in the fraud proof game. Notice also that optimistic native rollups can have a gas limit that far surpasses `EXECUTE_CUMULATIVE_GAS_LIMIT` because the `EXECUTE` precompile only needs to be called once on a small EVM segment to settle a fraud proof challenge.

As a historical note, in 2017 Vitalik suggested a similar **"EVM inside EVM" precompile** called `EXECTX`.

enforcement by SNARKs

To unlock a large `EXECUTE_CUMULATIVE_GAS_LIMIT` it is natural to have validators optionally verify SNARK proofs. From now on we assume one-slot delayed execution where invalid blocks (or invalid transactions) are treated as no-ops. (For more information about delayed execution see **this ethresearch post**, **this EIP**, and **this design by Francesco**.) One-slot delayed execution yield several seconds—a whole slot—for proving. They also avoid incentivising MEV-driven proof racing which would introduce a centralisation vector.

Note that even when `EXECUTE` is enforced by SNARKs no explicit proof system or circuit is enshrined in consensus. (Notice the `EXECUTE` precompile does not take any explicitly proof as input.) Instead, each staking operator is free to choose their favourite zkEL verifier client(s) similar to how EL clients are subjectively chosen today. The benefits of this design decision are explained in the next section titled "offchain proofs".

From now on we assume that execution proposers are sophisticated in the context of Attester-Proposer Separation (APS) with alternating execution and consensus slots. To incentivise rational execution proposers to generate proofs in a timely fashion (within 1 slot) we mandate attesters only attest to execution block `n+1` if proofs for execution block `n` are available. (We suggest bundling block `n+1` with `EXECUTE` proofs for block `n` at the p2p layer.) An execution proposer that skips proving may miss their slot, leading to missed fees and MEV. We further apply a fixed penalty for missed execution slots, setting it high enough (e.g. 1 ETH) to always surpass the cost of proving.

Note that in the context of APS the production of consensus blocks is not blocked by missed execution slots. Timely generation of proofs is however relevant for light clients to easily read state at the chain tip, without stateless re-execution. To ensure proofs are generated in a timely fashion for light clients, even in the exceptional case where the next execution proposer misses their slot, we rely on an altruistic-minority prover assumption. A single altruistic prover is sufficient to generate proofs within 1 slot. To avoid unnecessary redundant proving, most altruistic provers can wait on standby and kick in only when no proof arrives within 1 slot, thereby acting as a fail safe with at most 2-slot latency.

Note that `EXECUTE_CUMULATIVE_GAS_LIMIT` needs to be set low enough for the altruistic-minority prover assumption to be credible (as well as for execution proposing to not be unrealistically sophisticated). A conservative policy could be to set `EXECUTE_CUMULATIVE_GAS_LIMIT` so that single-slot proving is accessible to laptops.

e.g. high-end MacBook Pros. A more pragmatic and aggressive policy could be to target a small cluster of GPUs, and maybe eventually SNARK ASIC provers once those are sufficiently commoditised.

offchain proofs

To reiterate, we suggest that zkEL `EXECUTE` proofs not go onchain and instead be shared offchain. Not enshrining proofs is a beautiful idea **first suggested by Vitalik** which comes with several advantages:

- **diversity**: Validators are free to choose proof verifiers (including proof systems and circuits) from dev teams they trust, similar to how validators choose EL clients they trust. This provides robustness through diversity. zkEL verifier clients (and the zkVMs that would underlie some) are complex pieces of cryptographic software. A bug in any one client should not take down Ethereum.
- **neutrality**: Having a market of zkEL verifier clients allows for the consensus layer to not pick technology winners. For example, the zkVM market is highly competitive and picking a winning vendor such as Risc0, Succinct, or **the many other vendors** may not be perceived as neutral.
- **simplicity**: The consensus layer need not enshrine a specific SNARK verifier, dramatically simplifying the specification of the consensus layer. It is sufficient to enshrine a format for state access proofs, not specific proof verifier implementation details.
- **flexibility**: Should a bug or optimisation be found, affected validators can update their client without the need for a hard fork.

Having offchain proofs does introduce a couple manageable complications:

- **prover load and p2p fragmentation**: Because there isn't a single canonical proof, multiple proofs (at least one per zkEL client) needs to be generated. Every zkEL client customisation (e.g. swapping one RISC-V zkVM for another) requires a different proof. Likewise, every zkEL version bump requires a different proof. This will lead to increased proving load. It will additionally fragment the p2p network if there's a separate gossip channel per proof type.
- **minority zkELs**: It's hard to incentivise proof generation for minority zkELs. Rational execution proposers may only generate sufficient proofs so as to reach a super-majority of attesters and not miss their slot. To combat this, staking operators could be socially encouraged to run multiple zkEL clients in parallel similar to **Vouch** operators today. Running a `k`-of-`n` setup has the additional benefit of boosting security, in particular hedging against soundness bugs that allow an attacker to craft proofs for arbitrary `EXECUTE` calls (a situation that would be

unusual for a traditional EL client).

Offchain proofs also introduces inefficiencies for real-time settled L2s:

- **no alt DA**: Because the `trace` input to `EXECUTE` needs to have been made available to L1 validators, real-time settled L2 (i.e. L2s that immediately update their canonical state root) must consume L1 DA, i.e. be rollups. Notice that optimistic L2s that delay settlement via a fraud proof game do not have this limitation, i.e. can be validiums.
- **state access overhead**: Because the `trace` must be statelessly executable it must include state trie leaves that are read or written, introducing a small DA overhead over a typical L2 block. Notice that optimistic L2s do not have this limitation because state trie leaves are only required in fraud proof challenges and the challenger can recompute trie leaves.
- **no state diffing**: Because proving should be permissionless given the `trace`, rollup state diffing is not possible. It would however be possible to compress stateless access proofs or EVM transaction signatures if corresponding specialised proofs were enshrined in consensus.

RISC-V native execution

Given today's de facto **convergence towards RISC-V zkVMs** there may be an opportunity to expose RISC-V state transitions natively to the EVM (similarly to WASM in the context of **Arbitrum Stylus**) and remain SNARK-friendly.

Part 2—native rollups

naming

We start by discussing the naming of native rollups to address several sources of confusion:

- **alternative names**: Native rollups were previously referred to as enshrined rollups, see for example **this writeup** and **this writeup**. (The term "canonical rollup" was also briefly **used by Polynya**.) The term "enshrined" was later abandoned in favour of "native" to signal that existing that EVM-equivalent rollups have the option to upgrade to become native. The name "native" was independently suggested in November 2022 by Dan Robinson and a Lido contributor which wishes to remain anonymous.
- **based rollups**: Based rollups and native rollups are orthogonal concepts: "based" relates to L1 sequencing whereas "native" relates to L1 execution. A rollup that is simultaneously based and native is whimsically called an "ultra sound rollup".

- **execution shards**: Execution shards (i.e. enshrined copies of L1 EVM chains) is a different but related concept related to native rollups, predating native rollups by several years. (Execution sharding was previously "phase 2" of the Ethereum 2.0 roadmap.) Unlike native rollups, execution shards are non-programmable, i.e. without the option for custom governance, custom sequencing, custom gas token, etc. Execution shards are also typically instantiated in a fixed quantity (e.g. 64 or 1,024 shards). Unfortunately Martin Köppelmann used the term "native L2" in **his 2024 Devcon talk about execution shards**.

benefits

Native rollups have several benefits which we detail below:

- **simplicity**: Most of the sophistication of a native rollup VM can be encapsulated by the precompile. Today's EVM-equivalent optimistic and zk-rollups have thousands of lines of code for their fraud proof game or SNARK verifier that could collapse to a single line of code. Native rollups also don't require ancillary infrastructure like proving networks, watchtowers, and security councils.
- **security**: Building a bug-free EVM fraud proof game or SNARK verifier is a remarkably difficult engineering task that likely requires deep formal verification. Every optimistic and zk EVM rollup most likely has critical vulnerabilities today in their EVM state transition function. To defend against vulnerabilities, centralised sequencing is often used as a crutch to gate the production of adversarially-crafted blocks. The native execution precompile allows for the safe deployment of permissionless sequencing. Trustless rollups that fully inherit L1 security additionally fully inherit L1 asset fungibility.
- **EVM equivalence**: Today the only way for a rollup to remain in sync with L1 EVM rules is to have governance (typically a security council and/or a governance token) to mirror L1 EVM upgrades. (EVM updates still happen regularly via hard forks roughly once per year.) Not only is governance an attack vector, it is strictly-speaking a departure from the L1 EVM and prevents any rollup from achieving true long-term EVM equivalence. Native rollups on the other hand can upgrade in unison with the L1, governance-free.
- **SNARK gas cost**: Verifying SNARKs onchain is expensive. As a result many zk-rollups settle infrequently to minimise costs. Since SNARKs are not verified onchain the `EXECUTE` precompile could be used as a way to lower the cost of verification. If `EXECUTE` proofs across multiple calls in a block are batched using SNARK recursion `EXECUTE_GAS_COST` could be set relatively low.
- **synchronous composability**: Today synchronous composability with the L1 requires same-slot real-time proving. For zk rollups achieving ultra-low-latency proving, e.g. on the order of 100ms, is an especially challenging engineering task.

With one-slot delayed state root the proving latency underlying the native execution precompile can be relaxed to one full slot.

Native Rollup for 3SF

Fee structure for EXECUTE-precompile

Enshrined Native L2s and Stateless Block Building

Prover Killers Killer: You Build it, You Prove it

---

centauri **Jan 20**

> JustinDrake:
>
> If `EXECUTE_CUMULATIVE_GAS_LIMIT` is small enough validators can naively re-execute `trace`s to enforce correctness of `EXECUTE` calls. An initial deployment of the precompile based on re-execution could serve as a stepping stone

This is a really interesting idea. Considering that even very low end nodes on Ethereum only tend to spend a few % if its CPU resources on Execution, even a few hundred TPS across L2s that opt to use this would not be unreasonable i would assume.
And this could allow for synchronous composability between the L2 and the L1 aswell? Since the L1 nodes verifies L1 + L2 execution essentially in the same slot i see no reason why that wouldn't be the case

---

levs57 **Jan 20**

Hi, I am also making **my writeup on alternative approach to native rollups** public.

Huge thanks to Justin for explaining me the idea in details.

tl;dr:

In my version `EXECUTE` always corresponds to full evaluation, and is used by rollups to move execution to L1 if the proof system issue is detected (which is defined as proving two contradictory statements). This allows to make zk-rollups without governance committees (even without any "native" features), and *with* native features we can guarantee the full inheritance of security properties from L1.

So my idea is to do `EXECUTE` first (which will give every rollup a streamlined approach of

getting rid of security committee), and then add native features (de facto commitment by L1 to revert if the proof system fails / incentives to make proofs each slot) to the proof systems that are trusted enough. This will also serve as playground for eventual SNARK-ification of L1, and has less assumptions on builders and the rest of the infra.

---

ihagopian **Jan 20**

Great article!

I am sharing something I talked about with @JustinDrake out of band. Ideally, the underlying tree of the `EXECUTE` precompile should be the same as L1. It's the most natural choice and would prevent EL clients from supporting different trees or proofs for different trees over time.

The tricky part is if native rollups are deployed before an L1 state tree change, being Verkle or Binary Trees. If that's the case, then we'd need to deal with state tree conversions at the L1 level **and** on these native rollups – unless `EXECUTE` promises support for older L1 trees, which adds complexity and might impact future L1 snarkification.

The easiest solution is to deploy native rollups after an L1 state tree change so we only have to deal with the (expected) state conversion in L1, but not for native rollups. This has two negative implications: 1) a dependency for native rollups, 2) force L1 state tree change to be final (i.e., probably binary trees), since if we don't do this, we'll have the problem again in the future. [Not sure I'd say this point is a "negative" implication, just an extra constraint that doesn't exist today]

---

perama-v **Jan 20**

> JustinDrake:
>
> Notice that naive re-execution imposes no state growth or bandwidth overhead for validators, and any execution overhead is parallelisable across CPU cores.
>
> Validators must hold an explicit copy of `trace` for re-execution…

I like the direction. Regarding the data involved in the re-execution stepping-stone: Validators would be using more bandwidth as they must download the trace data. Which napkin math is about 127KB/s/Mgas/rollup for the non-ZK intermediate phase.

The trace data comprises the L2 block and L2 prestate. For a native rollup with the same

The trace data comprises the L2 block and L2 prestate. For a native rollup with the same gas limit as L1, that means a block equal to L1 block size plus L2 prestate equal to L1 prestate data size.

What is in the prestate? Mostly contracts, storage nodes, trie nodes.

How big is this prestate? About 3MB normally (for 15MGas blocks). This estimate is from a benchmark on real blocks after deduplication of common trie nodes/contracts, and using ssz + snappy compression (see ethereum/execution-apis PR#455 for details).

So, an L1 block containing three EXECUTE opcodes to progress 3 native rollups each of equal size to present L1 would require three lots of trace data amounting to: 0.3MB (3 * 0.1MB) of L2 blocks and 9MB (3 * 3MB) of prestate data.

Or as a rough rule, for 12 second slots, this is about 3MB/12=0.25MB/s per rollup. Which is 250/15=127KB/s/Mgas/rollup. So, with five native rollups each with an introductory tiny 2MGas, the additional bandwidth would be on the order of 1.2MB/s (0.12725) in the usual case for the non-ZK EXECUTE scenario. This seems reasonable and beneficial in that it would provide achievable intermediate steps to the full ZK-EXECUTE variant.

---

danielmarinq **Jan 20**

Very nice @JustinDrake – very supportive of this.

0xTariz **Jan 20**

Great write-up @JustinDrake

I'm really intrigued by this idea and want to learn more. Quick question: how do we ensure `EXECUTE` precompile is valid? If validators just re-execute the trace, where/how do they confirm the resulting state matches `post_state_root` on-chain?

---

edfelten **Jan 21**

A word of caution here. Practical L2 designs have several features that don't seem compatible with the design described here.

L2s have functionalities for things like trustless cross-chain deposits and withdrawals, which need to be supported in the L2 state transition function.

L2s use transaction types not supported by Ethereum, such as types that can be sent by

L1 contracts. (This requires a different transaction type because, for example, such transactions don't carry a signature.)

L2s do gas accounting differently than L1 does, for example by adding a surcharge to pay for data availability; or if the L2 has variable block time for efficiency, then adjusting gas prices based on timestamp rather than block number. Doing this trustlessly requires changes to the state transition function.

L2s support precompiles that do not exist on L1, for example to get information about the current data availability gas price, or to get information about the L1 state. These precompiles aren't just invoked as "top-level" transactions but are invoked within the execution of other transactions.

Marchhill **Jan 21**

Nice writeup @JustinDrake !

What are your thoughts on the viability of almost-native rollups, which maintain some small differences from full EVM compatibility?

One could introduce a new prover for the difference in features but this would entail reintroducing some governance.

---

otrsk **Jan 21**

So what exactly is a "trustless rollup" here? How is it different from a "native rollup", does the latter NOT "fully inherit Ethereum security"?

---

thegaram33 **Jan 21**

> JustinDrake:
>
> a rollup which uses `EXECUTE` to verify EVM state transitions for batches of user transactions. One can think of native rollups as "programmable execution shards" that wrap the precompile within a derivation function to handle extra-EVM system logic

There is some uncertainty whether Ethereum's state transition could be applied to rollups with no modification:

- **State transition function divergence** (mentioned by @edfelten).
- **EVM validity rules**. E.g. for Ethereum a block that contains a transaction with incorrect nonce or insufficient balance is itself invalid. For rollups, it's often beneficial to treat all blocks as valid (which may result in no-op state transition).

Also, the term *EVM state transition* misses a few steps that are today part of a most rollups' execution, and it's unclear how those steps could be combined with this precompile:

- **Chain derivation**: There are multiple steps to how most rollups derive the L2 chain from the blob data. At the least they have a decompression step, but the chain derivation pipeline could be much more complex.

> JustinDrake:
>
> `trace` is a well-formatted execution trace (e.g. a list of L2 transactions and corresponding state access proofs)

EVM execution also depends on the block context (block number, timestamp, etc.), these would also need to be passed to the precompile.

> JustinDrake:
>
> The `trace` must point to available Ethereum data from calldata, blobs, state, or memory.
> …
> Validators must hold an explicit copy of `trace` for re-execution, preventing the use of pointers to blob data which are sampled

Aren't these two sentences in conflict? Could you provide an example (few lines of Solidity) showing how a native rollup would pass `traces` to the precompile?

> JustinDrake:
>
> From now on we assume one-slot delayed execution where invalid blocks (or invalid transactions) are treated as no-ops

I think **EIP-7862** (and related discussion) only proposes delayed state root computation, not delayed execution. Most nodes are still expected to execute blocks in the same slot.

> JustinDrake:

> JustinDrake:
>
> To reiterate, we suggest that zkEL `EXECUTE` proofs not go onchain and instead be shared offchain.

The decoupling of execution from proving is nice and I can certainly see the benefits. Are there any guarantees/expectation of long-term availability for these proofs? If I spin up a new node I want to avoid reexecuting each precompile call and verify proofs instead.

> JustinDrake:
>
> Having a market of zkEL verifier clients allows for the consensus layer to not pick technology winners

I can see that the block proposer can use whichever proving vendor they want to use. But doesn't this mean that all EL clients need to be able to verify all possible types of proofs? If I receive a block with a proof, but I don't know how to verify this specific proof type, then I have to re-execute the traces and I'll fall behind.

> JustinDrake:
>
> With one-slot delayed state root the proving latency underlying the native execution precompile can be relaxed to one full slot.

How does this enable synchronous composability, could you elaborate?

---

ihagopian **Jan 21**

We could also try frontrunning native rollups to the final tree so that they don't need to do migrations in the future, even if we deploy them before the L1 state tree change.

Pros:

- Native rollups won't have to deal with tree conversions in the future, which is quite painful.
- Lower the probability that the problem is nasty enough to require us to support multiple trees at the protocol level forever.
- We'd be frontrunning real progress for L1 since, eventually, we'd use that same tree for L1, so there's no wasted effort, and it could be a good "playground" for maturing the exact implementation that would be used in the L1 after.

Cons:

- Temporarily, the L1 would have two tree descriptions. The current L1 tree(s) and native rollup trees are different. This means more (temporal) complexity. But it could be removed when L1 catches up.
- Temporary, less synergy between native rollups and L1 state trees.
- As soon as we decide on doing native rollups, we have to decide in the final state tree for L1. (Actually, I'd consider this as a pro… but maybe I'm biased 🙂)

Not entirely sure, but just sharing the idea.

0xvon **Jan 21**

Thank you for your refined proposal following **your ZK Summit 11 talk**. I'm very excited about the Native Rollups idea.

Question: If we assume an `EXECUTE` precompile is available, which parts of this proposal specifically require protocol updates?

I think "enforced by SNARKs" clearly needs an update because it requires communication between the CL and zkEL clients to generate and verify the `EXECUTE` proofs, but what about "enforced by re-execution"?

---

pepyakin **Jan 22**

The proposal "locks in" certain assumptions e.g. exact trie format, gas schedule, transaction types, the derivation function, etc. This limits flexibility of rollup developers and very likely reduces efficiency. This is made in the name, at least partly, of removing governance (or giving it to L1). This makes sense.

What are the degrees of freedom that are left to customization for a native rollup? As Justin mentions, those are at least: Bridging and Withdrawals, Sequencing and Custom Gas Token.

In the OP, it looks like the governance is referred to something that is both undesirable and necessary, and the `EXECUTE` opcode can alleviate need in them. However, why don't apply the same logic to bridging and withdrawals? If Ethereum took over that as well, then the user won't need to consult with L2beat before interacting with a rollup. They can just interact with a rollup because they know it is as secure as the L1.

And indeed, this would be something like upgrade of the execution shards. Execution

And indeed, this would be something like upgrade of the execution shards. Execution shards circa 2019 did not have the benefit of the domain knowledge we have today. For example, I believe it's still possible to allow for custom sequencing unlike the original version of execution shards.

Another direction emerges when we consider: what if we relax the structure of the `EXECUTE` opcode and instead of executing an EVM environment with all those things like storage, transactions and etc, the `EXECUTE` will execute a program that just takes some input and produces some output.

In this case we lose the EVM equivalence, regain flexibilitiy while maintaining the other benefits of this approach.

Specifically,

- The rollup developers will not have to reimplement the fraud proof game or SNARK. This complexity will be on the ethereum side.
- The rollup developer will only have to implement the specific logic of their exact EVM behavior (or reuse one from the shelf). This will allow for customization of STF and of derivation rules. This should lift the concerns that @edfelten expressed. Now, it looks to me that in this world, we should really embrace RISC-V. That would allow us to reuse the common toolchains such as Rust and products like reth to simplify implementation of EVM.
- This would avoid backing in the exact format of the trie. The program passed in the `EXECUTE` will pick it's own trace format.

You got the idea.

I am also pretty sure that patching up the problems brought up by @thegaram33 , @ihagopian and @edfelten will greatly complicate the proposal.

All in all, I want to express that this proposal picks a **suboptimal point** in the design space and would be better off if it either moved into direction:

1. of encompassing more stuff and bringing it closer to the ideas of execution shards,
2. or of encompassing less stuff and basically allowing for more innovation on rollups,

---

GregTheGreek **Jan 22**

Thanks Justin,

I'd like to propose we start a discussion around the incentives surrounding the

I'd like to propose we start a discussion around the incentives surrounding the sequencer fees, and blob pricing.

Sequencer fees should be fairly remunerated back to the L2

- The L1 shouldn't take all the fees, maybe only 10% or a fee mechanism
- The L2 brought the distribution and network effects and should take the majority of what they created. We are not Apple and should not be the App Store.
- We want L2s to create treasuries, distribute grants, and not rely on EF funding forever.

We should have a priority queue for native rollups in blob storage, and frankly they should have dedicated pricing models.

- The economic incentives exist to make this happen, and it should be part of the protocol itself. If a rollup is giving up some share of their sequencer revenue back to the protocol, we should ensure that rollup data has priority in DA. In a world with 100s of rollups, where non-Native Rollups have larger war chests than the Native Rollups, who gets priority? The highest bidder, or the rollup that is deeply engrained into Ethereum consensus. I'd argue the latter.

I went deeper on some of these things in my blog post (on chiansafe [dot] io) although already slightly dated due this post and other ongoing conversations, but good for context.

Jommi **Jan 22**

I guess those are not Native Rollups then?

---

maxgillett **Jan 22**

I'd like to propose some additions to the precompile that would allow L2s to continue to permissionlessly innovate on execution, state, and governance without hard forks, and to go beyond the EVM.

Addressing L2 diversity through emulation

First, we need to recognize that every existing L2 has diverged to some extent from the EVM, and in non-uniform ways. These include differences in opcode, precompile, and system contract behavior. A good summary of these differences can be found **here**. If current L2s are to transition to native rollups, then an `EXECUTE` precompile (perhaps in

concert with L1 application logic) must be able to support these differences. Furthermore, L2s should not be required to adopt new opcodes and precompiles merged into the L1. These two layers have different competing goals, including whether to ossify or innovate, among others.

If total EVM equivalence is not practical (or desirable) for native rollups, we might ask why any degree of EVM equivalence should be enforced. All attestors are already equipped with EVM execution environments, and attestors must be able to safely verify `EXECUTE` calls, so it is natural to assume that attesting to a block with this precompile involves re-executing EVM bytecode, or at the very least, relying upon a validity proof of that execution.

If we want to allow anyone to permissionlessly deploy a native rollup that uses an alternative execution environment (e.g. RISC-V, WASM, Cairo VM, Move VM, etc), and to do so independently of hard forks, then we need to define a standard for how that execution environment can be emulated in existing clients.

I've outlined one potential approach below: extending the EXECUTE precompile to include an option for using a smart contract as an interpreter for a custom VM. This would enable native rollups to implement alternative execution environments while maintaining compatibility with existing clients. A similar extension could be made to accommodate novel state models, which I don't go into detail here. I also discuss at the end how this emulation is not a bottleneck for performance.

Extending the EXECUTE precompile

I'd propose that the `EXECUTE` precompile takes in two additional arguments: `vm` and `state` (which could be subsumed within the trace format). Each of these point to an L1 smart contract address. The `vm` contract provides a stateless interpreter for the custom VM, emulated within the EVM. The `state` contract describes how authenticated state is stored and retrieved. I'm less clear on the interface for this contract, but it should be executable, and a pure function. The `EXECUTE` precompile uses both the interpreter and state model description in its verification of the pre and post state root, and `gas_used` claim. If the `vm` argument is null, then the existing L1 EVM is assumed. If the `state` argument is null, then the MPT-based state model is used.

We do not need to redeploy a contract describing the entire VM each time we want to make a modification. For example, if we want to patch specific EVM opcode functionality, we could deploy a new contract which points to null (the default EVM interpreter), along with some functions adhering to a standard that describe the differences in opcode behavior. If we wanted to patch a custom VM, then we could deploy a contract that points to a previously deployed interpreter for that VM.

points to a previously deployed interpreter for that VM.

Eliminating emulation overhead

If network participants need to naively re-execute the `vm` and `state` contracts for a native rollup's VM and state model respectively, then the overhead involved with emulation could make associated rollup transactions prohibitively expensive, as it shares in consumption of the `EXECUTE_CUMULATIVE_GAS_LIMIT`.

However, in the re-execution phase, if attestors, proposers, and builders coordinate off-chain to adopt optimized execution clients that replace emulation with native (in the sense of x86/ARM) machine code, then true client cost can come down considerably, which could be reflected in an increase in the gas limit. In the zkEL phase, attestors only need to worry about downloading updated program hashes to benefit from increases in block builder performance.

The capacity for a rollup using a custom VM could be increased by defining a separate limit for each deployed VM. Each separate limit could start at zero, and be voted up or down by consensus proposers. Equivalently, we could apply a votable global scaling factor to each separate VM opcode set in the range [0,1]. The relative gas cost for each opcode would remain the same as in its EVM emulation (representing "worst case execution"), but the relative gas limit for execution in that VM would increase.

We could also significantly reduce emulation overhead if the interpreter and state contracts make use of a RISC-V (or other non-EVM) execution precompile. The challenge would lie in defining appropriate gas pricing for each RISC-V opcode.

Introduction of optimized execution client code could be done at a cadence that coincides with hard forks, or on a more frequent schedule.

Conclusion

One could argue that we should treat the differences in L2 EVM behavior as a special case, and restrict focus to adding a system that enables L2s to continue to patch the EVM with their desired functionality. If Ethereum wants to go all-in on the EVM, then perhaps this is ok. But I think there's an opportunity to both onboard non-EVM ecosystems as native rollups if we go the emulation route, and set ourselves up to scale beyond the inherent throughput limitations of MPT-based EVM implementations.

---

i-norden **Jan 23**

Am I understanding correctly that in these fault proof games the bisection will need to

occur down to no lower than the level of a single transaction? In order to use EXECUTE where the prestate and poststate are MMPT state roots the bisection cannot occur down to an opcode level because some opcodes do not mutate the state. This is in contrast to current bisection games where you bisect all the way down to a single e.g. MIPS instruction where the prestate and poststate are VM frames not MMPT state roots.

In a “conventional” fault proof game you are bisecting over the trace of a program that is checking whether or not it derives from data on the L1 the same L2 output that has been proposed. You aren’t just bisecting over transaction execution but also the L1->L2 derivation logic and the evaluation of said execution. How do we get around this for native rollups, as it is not possible if the thing we EXECUTE at the lowest level of bisection must an EVM transaction (or block of transactions)?

---

thegaram33 **Jan 24**

Thinking out loud about whether and how **native rollups** can support behaviors different from standard EVM.

Why is this important? If native rollups can only support vanilla EVM, then

- They will either need to be expensive (cannot compress DA), or opinionated (enforce a specific blob compression and encoding).
- It is unlikely that existing rollups can upgrade to become native. For instance existing deposit mechanisms often rely on special transaction types.
- They cannot adopt RIPs.

Note: Even if this is the case, native rollups still offer some major benefits. Just want to reach more clarity about what they can and cannot do.

Many rollups can be described using this simplified process:

```
Data --> Derive (CDF) --> Execute (STF)
```

The **derivation step** (aka CDF - Chain Derivation Function) typically does stuff like decompressing blobs and parsing raw data, but it can also be more sophisticated than that. The **execution step** (aka STF - State Transition Function) is usually the EVM logic with optional extensions.

When it comes to rollups changing/extending EVM behavior, I can think of 3 main patterns:

Extension Pattern #1: Custom Derivation

```
Data --> **Derive** --> Execute EVM
```

Examples: Decompression, force insertion of messages.

Even if the STF is vanilla EVM that can be verified using the `EXECUTE` precompile, we'd most likely need to run some complex algorithm on the `blob` to produce `trace` before we could feed it into the precompile.

Extension Pattern #2: Custom Execution

```
Data --> **Custom Execute**
```

Any change to the EVM that affects all transactions would result in a non-standard STF. A simple example is custom fee logic. As I understand `EXECUTE` is supposed to verify uniform logic for all native rollups so this is not supported.

Extension Pattern #3: Multiple STFs

```
Data --> Execute EVM
  or
Data --> **Custom Execute**
```

This seems to be what Justin is referring to in **Native Rollups Call #0**:

> "The simplest kind of native rollup is one where all of the execution goes through this precompile. But part of the value of having programmability is that you can add logic around it. So you can imagine having a custom gas token, custom governance, or maybe you could have special system transactions that don't go through this precompile."

So the rollup contract could offer two distinct ways of updating the rollup state:

```
function submitEVMBlob(bytes32 postStateRoot) {
    // ...

    // verify blob using native rollup precompile
    bytes32 preStateRoot = currentStateRoot;
    EXECUTE(preStateRoot, postStateRoot, trace);
    currentStateRoot = postStateRoot;
}
```

```
function submitCustomBlob(bytes32 postStateRoot, bytes calldata proof) {
    // ...

    // verify blob using custom verification
    bytes32 preStateRoot = currentStateRoot;
    customVerifier.verify(preStateRoot, postStateRoot, trace, proof)
    currentStateRoot = postStateRoot;
}
```

Any transactions that do non-standard EVM stuff need to be submitted and verified through `submitCustomBlob`. Examples: Call custom precompile, execute on alternative VM, use a non-standard transaction type, etc.

This works but it's certainly not a "trustless rollup" since the extension logic is still controlled by the rollup.

---

edfelten **Jan 24**

The difficulty with having separate EVM execution state transition function (STF) and extension-feature STF, with each blob or block using only one or the other, is that in practice vanilla EVM functionalities and extension functionalities need to be composable, so that a single transaction can use some of both. It's a core principle that any contract can call any other contract, and that these calls can nest. So in practice a chain needs to have a single STF.