Should Ethereum be okay with enshrining more things in the protocol?

2023 Sep 30 See all posts

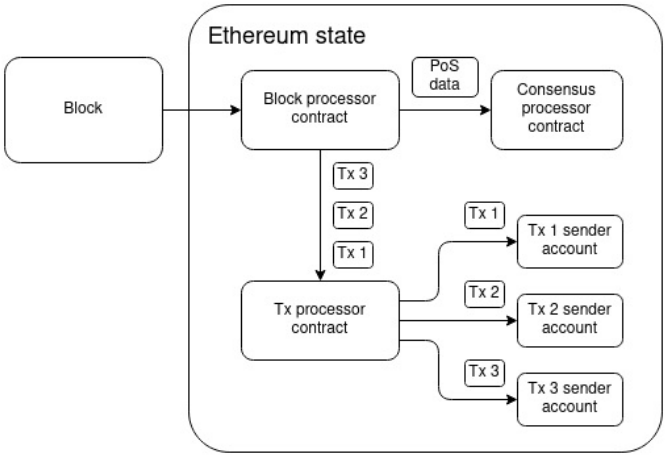
Special thanks to Justin Drake, Tina Zhen and Yoav Weiss for feedback and review.

From the start of the Ethereum project, there was a strong philosophy of trying to make the core Ethereum as simple as possible, and do as much as possible by building protocols on top. In the blockchain space, the "do it on L1" vs "focus on L2s" debate is typically thought of as being primarily about scaling, but in reality, similar issues exist for serving many kinds of Ethereum users' needs: digital asset exchange, privacy, usernames, advanced cryptography, account safety, censorship resistance, frontrunning protection, and the list goes on. More recently, however, there has been some cautious interest in being willing to enshrine more of these features into the core Ethereum protocol.

This post will go into some of the philosophical reasoning behind the original minimal-enshrinement philosophy, as well as some more recent ways of thinking about some of these ideas. The goal will be to start to build toward a framework for better identifying possible targets where enshrining certain features in the protocol might be worth considering.

Early philosophy on protocol minimalism

Early on in the history of what was then called "Ethereum 2.0", there was a strong desire to create a clean, simple and beautiful protocol that tried to do as little as possible itself, and left almost everything up to users to build on top. Ideally, the protocol would *just* be a virtual machine, and verifying a block would *just* be a single virtual machine call.



A very approximate reconstruction-from-memory of a whiteboard drawing Gavin Wood and I made back in early 2015, talking about what Ethereum 2.0 would look like.

The "state transition function" (the function that processes a block) would just be a single VM call, and all other logic would happen through contracts: a few system-level contracts, but mostly contracts provided by users. One really nice feature of this model is that even an entire hard fork could be described as a single transaction to the block processor contract, which would be approved through either offchain or onchain governance and then run with escalated permissions.

These discussions back in 2015 particularly applied to two areas that were on our minds: **account abstraction** and **scaling**. In the case of scaling, the idea was to try to create a maximally abstracted form of scaling that would feel like a natural extension of the diagram above. A contract could make a call to a piece of data that was not stored by most Ethereum nodes, and the protocol would detect that, and resolve the call through some kind of very generic scaled-computation functionality. From the virtual machine's point of view, the call would go off into some separate sub-system, and then some time later magically come back with the correct answer.

This line of thinking was explored briefly, but soon abandoned, because we were too preoccupied with verifying that any kind of blockchain scaling was possible at all. Though as we will see later, the combination of data availability sampling and ZK-EVMs means that one possible future for Ethereum scaling might actually look surprisingly close to that vision! For account abstraction, on the other hand, we knew from the start that *some* kind of implementation was possible, and so research immediately began to try to make something as close as possible to the purist starting point of "a transaction is just a call" into reality.

```
def apply_transaction(state, tx):
    state.logs = []
    state.suicides = []
    state.refunds = 0
    validate_transaction(state, tx)

    intrinsic_gas = tx.intrinsic_gas_used
    if state.is_HOMESTEAD():
        assert tx.s * 2 < transactions.secpk1n
        if not tx.to or tx.to == CREATE_CONTRACT_ADDRESS:
            intrinsic_gas += opcodes.CREATE[3]
            if tx.startgas < intrinsic_gas:
                raise InsufficientStartGas(rp(tx, 'startgas', tx.startgas, intrinsic_gas))

    log_tx.debug('TX NEW', txdict=tx.to_dict())

    # start transacting #################
    if tx.sender != null_address:
        state.increment_nonce(tx.sender)

    # buy startgas
    assert state.get_balance(tx.sender) >= tx.startgas * tx.gasprice
    state.delta_balance(tx.sender, -tx.startgas * tx.gasprice)

    message_data = vm.CallData([safe_ord(x) for x in tx.data], 0, len(tx.data))
    message = vm.Message(tx.sender, tx.to, tx.value, tx.startgas - intrinsic_gas, message_data, code_address=tx.to)

    # MESSAGE
    ext = VMExt(state, tx)
```

There is a lot of boilerplate code that occurs in between processing a transaction and making the actual underlying EVM call out of the sender address, and a lot more boilerplate that comes after. How do we reduce this code to as close to nothing as possible?

One of the major pieces of code in here is `validate_transaction(state, tx)`, which does things like checking that the nonce and signature of the transaction are correct. The *practical* goal of account abstraction was, from the start, to allow the user to replace basic nonce-incrementing and ECDSA validation with their own validation logic, so that users could more easily use things like social recovery and multisig wallets. Hence, finding a way to rearchitect `apply_transaction` into just being a simple EVM call was not simply a "make the code clean for the sake of making the code clean" task; rather, it was about moving the logic into the user's account code, to give users that needed flexibility.

However, the insistence on trying to make `apply_transaction` contain as little enshrined logic as possible ended up introducing a lot of challenges. To see why, let us zoom in on one of the earliest account abstraction proposals, EIP 86:

> ### Specification
>
> If `block.number >= METROPOLIS_FORK_BLKNUM`, then: 1. If the signature of a transaction is `(0, 0, 0)` (ie. `v = r = s = 0`), then treat it as valid and set the sender address to `2**160 - 1` 2. Set the address of any contract created through a creation transaction to equal `sha3(0 + init code) % 2**160`, where `+` represents concatenation, replacing the earlier address formula of `sha3(rlp.encode([sender, nonce]))` 3. Create a new opcode at `0xfb`, `CREATE_P2SH`, which sets the creation address to `sha3(sender + init code) % 2**160`. If a contract at that address already exists, fails and returns 0 as if the init code had run out of gas.

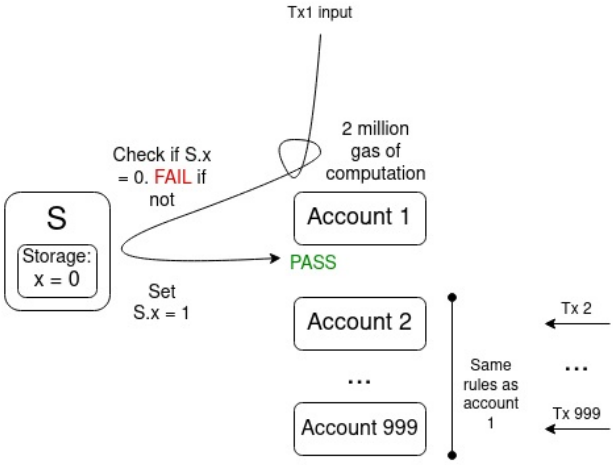
Basically, if the signature is set to (0, 0, 0), then a transaction really does become "just a call". The account itself would be responsible for having code that parses the transaction, extracts and verifies the signature and nonce, and pays fees; see here for an early example version of that code, and see here for the very similar `validate_transaction` code that this account code would be replacing.

In exchange for this simplicity at protocol layer, miners (or, today, block proposers) gain the additional responsibility of running extra logic for only accepting and forwarding transactions that go to accounts whose code is set up to actually pay fees. What is that logic? Well, honestly EIP-86 did not think *too* hard about it:

> Note that miners would need to have a strategy for accepting these transactions. This strategy would need to be very discriminating, because otherwise they run the risk of accepting transactions) for the `validate_transaction` code that this pre-account code would be replacingthat do not pay them any fees, and possibly even transactions that have no effect (eg. because the transaction was already included and so the nonce is no longer current). One simple approach is to have a whitelist for the codehash of accounts that they accept transactions being sent to; approved code would include logic that pays miners transaction fees.

> However, this is arguably too restrictive; a looser but still effective strategy would be to accept any code that fits the same general format as the above, consuming only a limited amount of gas to perform nonce and signature checks and having a guarantee that transaction fees will be paid to the miner. Another strategy is to, alongside other approaches, try to process any transaction that asks for less than 250,000 gas, and include it only if the miner's balance is appropriately higher after executing the transaction than before it.

If EIP-86 had been included as-is, it would have reduced the complexity of the EVM, at the cost of massively increasing the complexity of other parts of the Ethereum stack, requiring essentially the exact same code to be written in other places, in addition to introducing entirely new classes of weirdness such as the possibility that the same transaction with the same hash might appear multiple times in the chain, not to mention the multi-invalidation problem.



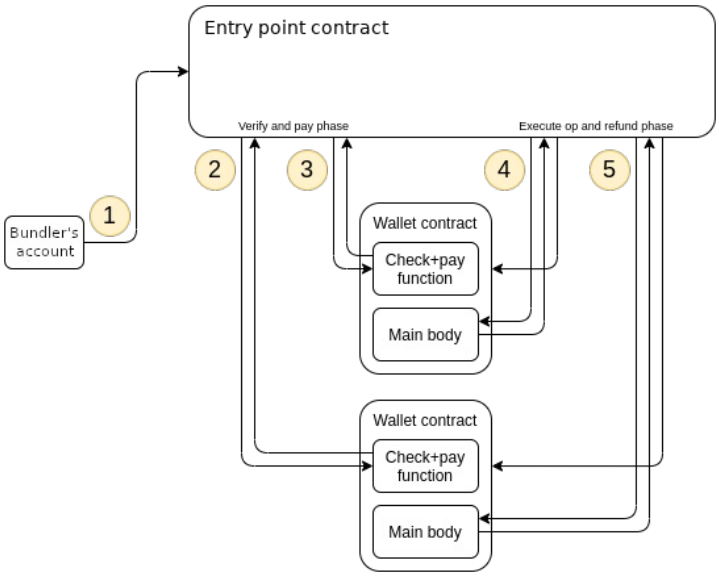
The multi-invalidation problem in account abstraction. One transaction getting included on chain could invalidate thousands of other transactions in the mempool, making the mempool easy to cheaply flood.

Acccount abstraction evolved in stages from there. EIP-86 became EIP-208, which later became this ethresear.ch post on "tradeoffs in account abstraction proposals", which then became this ethresear.ch post half a year later. Eventually, out of all this, came the actually somewhat-workable EIP-2938.

EIP-2938, however, *was not minimalistic at all*. The EIP includes:

- A new transaction type
- Three new transaction-wide global variables
- Two new opcodes, including the highly unwieldy `PAYGAS` opcode that handles gas price and gas limit checking, being an EVM execution breakpoint, and temporarily storing ETH for fee payments all at once.
- A set of complex mining and rebroadcasting strategies, including a list of banned opcodes for the validation phase of a transaction

In order to get account abstraction off the ground without involving Ethereum core developers who were busy on heroic efforts optimizing the Ethereum clients and implementing the merge, EIP-2938 eventually was rearchitected into the entirely extra-protocol ERC-4337.



ERC-4337. It really does rely entirely on EVM calls for everything!

Because it's an ERC, it does not require a hard fork, and technically lives "outside of the Ethereum protocol". So.... problem solved? Well, as it turns out, not quite. The *current* medium-term roadmap for ERC-4337 actually does involve eventually turning large parts of ERC-4337 into a series of protocol features, and it's a useful instructive example to see the reasons why this path is being considered.

Enshrining ERC-4337

There have been a few key reasons discussed for eventually bringing ERC-4337 back into the protocol:

- **Gas efficiency**: Anything done inside the EVM incurs some level of virtual machine overhead, including inefficiency in how it uses gas-expensive features like storage slots. Currently, these extra inefficiencies add up to at least ~20,000 gas, and often more. Pushing these components into the protocol is the easiest way to remove these issues.
- **Code bug risk**: if the ERC-4337 "entry point contract" has a sufficiently terrible bug, *all* ERC-4337-compatible wallets could see all of their funds drained. Replacing the contract with an in-protocol functionality creates an implied responsibility to fix code bugs with a hard fork, which removes funds-draining risk for users.
- **Support for EVM opcodes like `tx.origin`**. ERC-4337, by itself, makes `tx.origin` return the address of the "bundler" that packaged up a set of user operations into a transaction. Native account abstraction could fix this, by making `tx.origin` point to the actual account sending the transaction, making it work the same way as for EOAs.
- **Censorship resistance**: one of the challenges with proposer/builder separation is that it becomes easier to censor individual transactions. In a world where individual *transactions* are legible to the Ethereum protocol, this problem can be greatly mitigated with inclusion lists, which allow proposers to specify a list of transactions that *must* be included within the next two slots in almost all cases. But the extra-protocol ERC-4337 wraps "user operations" inside a single transaction, making user operations opaque to the Ethereum protocol; hence, Ethereum-protocol-provided inclusion lists would not be able to provide censorship resistance to ERC-4337 user operations. Enshrining ERC-4337, and making user operations a "proper" transaction type, would solve this problem.

It's worth zooming into the gas efficiency issue further. In its current form, ERC-4337 is significantly more expensive than a "basic" Ethereum transaction: the transaction costs 21,000 gas, whereas ERC-4337 costs ~42,000 gas. This doc lists some of the reasons why:

- Need to pay lots of individual storage read/write costs, which in the case of EOAs get bundled into a single 21000 gas payment:
  - Editing the storage slot that contains pubkey+nonce (~5000)
  - UserOperation calldata costs (~4500, reducible to ~2500 with compression)
  - ECRECOVER (~3000)
  - Warming the wallet itself (~2600)
  - Warming the recipient account (~2600)
  - Transferring ETH to the recipient account (~9000)
  - Editing storage to pay fees (~5000)
  - Access the storage slot containing the proxy (~2100) and then the proxy itself (~2600)
- On top of the above storage read/write costs, the contract needs to do "business logic" (unpacking the UserOperation, hashing it, shuffling variables, etc) that EOA transactions have handled "for free" by the Ethereum protocol
- Need to expend gas to pay for logs (EOAs don't issue logs)
- One-time contract creation costs (~32000 base, plus 200 gas per code byte in the proxy, plus 20000 to set the proxy address)

Theoretically, it should be possible to massage the EVM gas cost system until the in-protocol costs and the extra-protocol costs for accessing storage match; there is no reason why transferring ETH needs to cost 9000 gas when other kinds of storage-editing operations are much cheaper. And indeed, two EIPs ([1] [2]) related to the upcoming Verkle tree transition actually try to do that. But even if we do that, there is one huge reason why enshrined protocol features are going to inevitably be significantly cheaper than EVM code, no matter how efficient the EVM becomes: **enshrined code does not need to pay gas for being pre-loaded**.

Fully functional ERC-4337 wallets are *big*. This implementation, compiled and put on chain, takes up ~12,800 bytes. Of course, you can deploy that big piece of code once, and use `DELEGATECALL` to allow each individual wallet to call into it, but that code still needs to be accessed in each block that uses it. Under the Verkle tree gas costs EIP, 12,800 bytes would make up 413 chunks, and accessing those chunks would require paying 2x `WITNESS_BRANCH_COST` (3,800 gas total) and 413x `WITNESS_CHUNK_COST` (82,600 gas total). And this does not even begin to mention the ERC-4337 entry-point itself, with 23,689 bytes onchain in version 0.6.0 (under the Verkle tree EIP rules, ~158,700 gas to load).

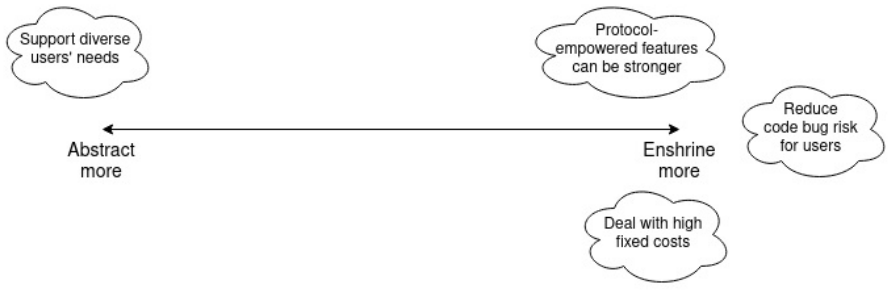
This leads to a problem: the gas costs of actually accessing this code would have to be split among transactions somehow. The current approach that ERC-4337 uses is not great: the first transaction in a bundle eats up one-time storage/code reading costs, making it much more expensive than the rest of the transactins. Enshrinement in-protocol would allow these commonly-shared libraries to simply be part of the protocol, accessible to all with no fees.

What can we learn from this example about when to enshrine things more generally?

In this example, we saw a few different rationales for enshrining aspects of account abstraction in the protocol.

- **"Move complexity to the edges" market-based approaches break down the most when there are high fixed costs**. And indeed, the long term account abstraction roadmap looks like it's going to have *lots* of fixed costs per block. 244,100 gas for loading standardized wallet code is one thing; but aggregation (see my presentation from this summer for more details) potentially adds hundreds of thousands more gas for ZK-SNARK validation plus onchain costs for proof verification. There isn't a way to charge users for these costs without introducing lots of market inefficiencies, whereas making some of these functionalities into protocol features accessible to all with no fees cleanly solves that problem.
- **Community-wide response to code bugs**. If some set of pieces of code are used by all users, or a very wide class of users, then it often makes more sense for the blockchain community to take on itself the responsibility to hard-fork to fix any bugs that arise. ERC-4337 introduced a large amount of globally shared code, and in the long term it makes more sense for bugs in that code to be fixed by hard forks than to lead to users losing a large amount of ETH.
- **Sometimes, a stronger form of a feature can be implemented by directly taking advantage of the powers of the protocol**. The key example here is in-protocol censorship resistance features like inclusion lists: in-protocol inclusion lists can do a better job of guaranteeing censorship resistance than extra-protocol approaches, in order for user-level operations to actually benefit from in-protocol inclusion lists, individual user-level operations need to be "legible" to the protocol. Another lesser-known example is that 2017-era Ethereum proof of stake designs had account abstraction for staking keys, and this was abandoned in favor of enshrining BLS because BLS supported an "aggregation" mechanism, which would have to be implemented at protocol and network level, that could make handling a very large number of signatures much more efficient.

But it is important to remember that **even enshrined in-protocol account abstraction is still a massive "de-enshrinement" compared to the status quo.** Today, top-level Ethereum transactions can only be initiated from externally owned accounts (EOAs) which use a single secp256k1 elliptic curve signature for verification. Account abstraction de-enshrines this, and leaves verification conditions open for users to define. And so, in this story about account abstraction, we also saw the biggest argument *against* enshrinement: **being flexible to diverse users' needs**.



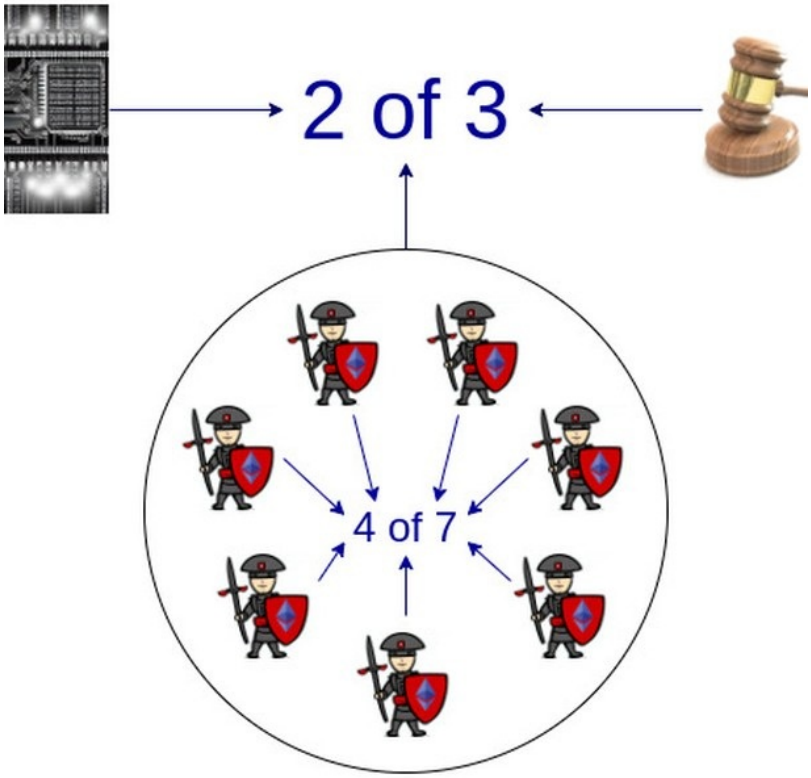
Let us try to fill in the story further, by looking at a few other examples of features that have recently been considered for enshrinement. We'll particularly focus on: **ZK-EVMs**, **proposer-builder separation**, **private mempools**, **liquid staking** and **new precompiles**.

Enshrining ZK-EVMs

Let us switch focus to another potential target for enshrining into the Ethereum protocol: ZK-EVMs. Currently, we

have a large number of ZK-rollups that all have to write fairly similar code to verify execution of Ethereum-like blocks inside a ZK-SNARK. There is a pretty diverse ecosystem of independent implementations: the PSE ZK-EVM, Kakarot, the Polygon ZK-EVM, Linea, Zeth, and the list goes on.

One of the recent controversies in the EVM ZK-rollup space has to do with how to deal with the possibility of bugs in the ZK-code. Currently, all of these systems that are live have some form of "security council" mechanism that can override the proving system in case of a bug. In this post last year, I tried to create a standardized framework to encourage projects to be clear about what level of trust they put in the proving system and what level in the security council, and move toward giving less and less powers to the security council over time.



In the medium term, rollups could rely on multiple proving systems, and the security council would only have any power at all in the extreme case where two different proving systems disagree with each other.

However, there is a sense in which some of this work *feels superfluous*. We already have the Ethereum base layer, which has an EVM, and we already have a working mechanism for dealing with bugs in implementations: if there's a bug, the clients that have the bug update to fix the bug, and the chain goes on. Blocks that appeared finalized from the perspective of a buggy client would end up no-longer-finalized, but at least we would not see users losing funds. Similarly, if a rollup just wants to be and remain EVM-equivalent, it feels wrong that they need to implement their own governance to keep changing their internal ZK-EVM rules to match upgrades to the Ethereum base layer, when ultimately they're building on top of the Ethereum base layer itself, which knows when it's being upgraded and to what new rules.

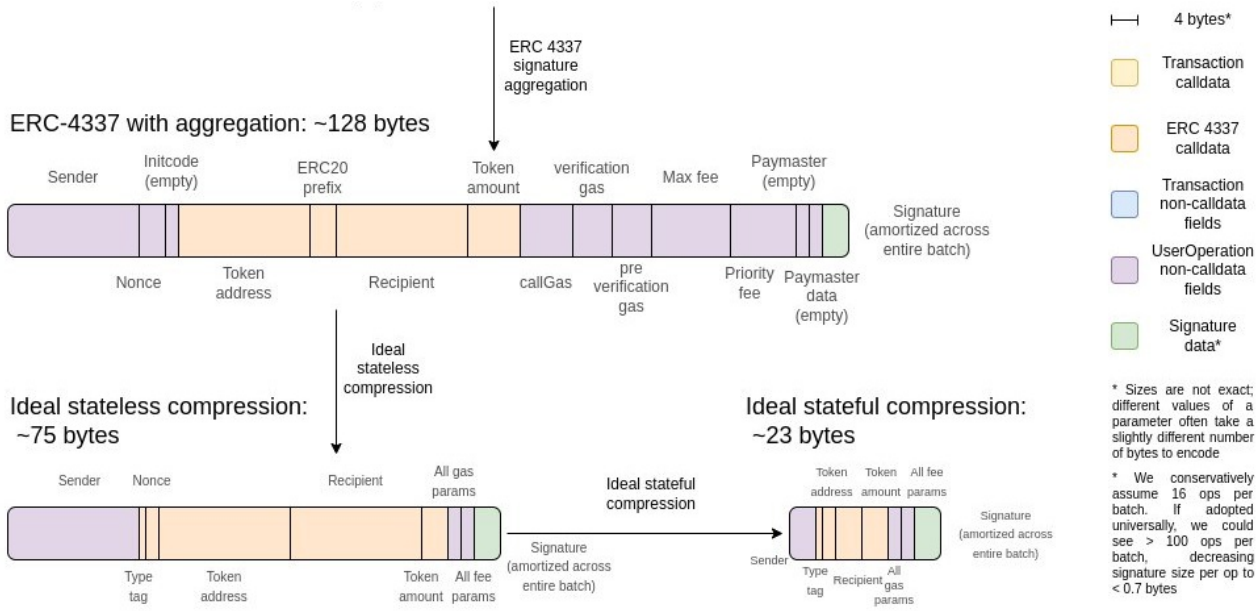
Since these L2 ZK-EVMs are basically using the exact same EVM as Ethereum, can't we somehow make "verify EVM execution in ZK" into a protocol feature, and deal with exceptional situations like bugs and upgrades by just applying Ethereum's social consensus, the same way we already do for base-layer EVM execution itself?

This is an important and challenging topic. There are a few nuances:

1. **We want to be compatible with Ethereum's multi-client philosophy**. This means that we want to allow different clients to use different proving systems. This in turn implies that for any EVM execution that gets proven with one ZK-SNARK system, we want a guarantee that the underlying data is *available*, so that proofs can be generated for other ZK-SNARK systems.
2. **While the tech is immature, we probably want auditability.** In practice, this means the exact same thing: 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.
3. **We need much faster proving times**, so that if one type of proof is made, other types of proof can be generated quickly enough that other clients can validate them. One *could* get around this by making a precompile that has an asynchronous response after some time window longer than a slot (eg. 3 hours), but this adds complexity.
4. **We want to support not just copies of the EVM, but also "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 precompile 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.

One likely topic of contention with data availability in a native ZK-EVM is **statefulness**. ZK-EVMs are much more data-efficient if they do not have to carry "witness" data. That is, if a particular piece of data was already read or written in some previous block, we can simply assume that provers have access to it, and we don't have to make it available again. This goes beyond not re-loading storage and code; it turns out that if a rollup properly *compresses* data, the compression being *stateful* allows for up to 3x data savings compared to the compression being stateless.



This means that for a ZK-EVM precompile, we have two options:

1. **The precompile requires all data to be available in the same block**. This means that provers can be stateless, but it also means that ZK-rollups using such a precompile become much more expensive than rollups using custom code.
2. **The precompile allows pointers to data used or generated by *previous* executions.** This allows ZK-rollups to be near-optimal, but it's more complicated and introduces a new kind of state that has to be stored by provers.

What lessons can we take away from this? There is a pretty good argument to enshrine ZK-EVM validation somehow: rollups are already building their own custom versions of it, and it *feels wrong* that Ethereum is willing to put the weight of its multiple implementations and off-chain social consensus behind EVM execution on L1, but L2s doing the exact same work have to instead implement complicated gadgets involving security councils. But on the other hand, there is a big devil in the details: there are different versions of an enshrined ZK-EVM that have different costs and benefits. The stateful vs stateless divide only scratches the surface; attempting to support "almost-EVMs" that have custom code proven by other systems will likely reveal an even larger design space. Hence, **enshrining ZK-EVMs presents both promise and challenges**.

Enshrining proposer-builder separation (ePBS)

The rise of MEV has made block production into an economies-of-scale-heavy activity, with sophisticated actors being able to produce blocks that generate much more revenue than default algorithms that simply watch the mempool for transactions and include them. The Ethereum community has so far attempted to deal with this by using *extra-protocol proposer-builder separation* schemes like MEV-Boost, which allow regular validators ("proposers") to outsource block building to specialized actors ("builders").

However, MEV-Boost carries a trust assumption in a new category of actor, called a *relay*. For the past two years, there have been many proposals to create "enshrined PBS". What is the benefit of this? In this case, the answer is pretty simple: the PBS that can be built by directly using the powers of the protocol is simply stronger (in the sense of having weaker trust assumptions) than the PBS that can be built without them. It's a similar case to the case for enshrining in-protocol price oracles - though, in *that* situation, there is also a strong counterargument.

Enshrining private mempools

When a user sends a transaction, that transaction becomes immediately public and visible to all, even before it gets included on chain. This makes users of many applications vulnerable to economic attacks such as frontrunning: if a user makes a large trade on eg. Uniswap, an attacker could put in a transaction right before them, increasing the price at which they buy, and collecting an arbitrage profit.

Recently, there has been a number of projects specializing in creating "private mempols" (or "encrypted mempools"), which keep users' transactions encrypted until the moment they get irreversibly accepted into a block.

The problem is, however, that schemes like this require a particular kind of encryption: to prevent users from flooding the system and frontrunning the decryption process *itself*, the encryption must auto-decrypt once the transaction actually does get irreversibly accepted.

To implement such a form of encryption, there are various different technologies with different tradeoffs, described well in this post by Jon Charbonneau (and this video and slides):

- **Encryption to a centralized operator**, eg. Flashbots Protect.
- **Time-lock encryption**, a form of encryption which can be decrypted by anyone after a certain number of *sequential* computational steps, which cannot be parallelized.
- **Threshold encryption**, trusting an honest majority committee to decrypt the data. See the shutterized beacon chain concept for a concrete proposal.
- **Trusted hardware** such as SGX.

Unfortunately, each of these have varying weaknesses. A centralized operator is not acceptable for inclusion in-protocol for obvious reasons. Traditional time lock encryption is too expensive to run across thousands of transactions in a public mempool. A more powerful primitive called **delay encryption** allows efficient decryption of an unlimited number of messages, but it's hard to construct in practice, and attacks on existing constructions still sometimes get discovered. Much like with hash functions, we'll likely need a period of more years of research and analysis before delay encryption becomes sufficiently mature. Threshold encryption requires trusting a majority to not collude, in a setting where they can collude *undetectably* (unlike 51% attacks, where it's immediately obvious who participated). SGX creates a dependency on a single trusted manufacturer.

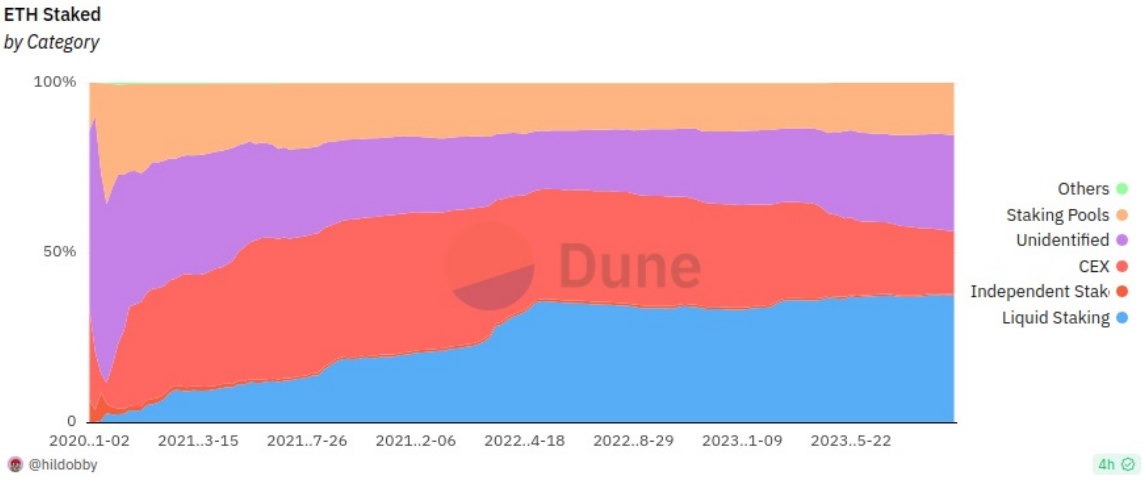
While for each solution, there is some subset of users that is comfortable trusting it, there is no single solution that is trusted enough that it can practically be accepted into layer 1. Hence, enshrining anti-frontrunning at layer 1 seems like a difficult proposition at least until delay encrypted is perfected or there is some other technological breakthrough, even while it's a valuable enough functionality that lots of application solutions will already emerge.

Enshrining liquid staking

A common demand among Ethereum defi users is the ability to use their ETH for staking and as collateral in other applications at the same time. Another common demand is simply for convenience: users want to be able to stake without the complexity of running a node and keeping it online all the time (and protecting their now-online staking keys).

By far the simplest possible "interface" for staking, which satisfies both of these needs, is just an ERC20 token: convert your ETH into "staked ETH", hold it, and then later convert back. And indeed, liquid staking providers such as Lido and Rocketpool have emerged to do just that. However, liquid staking has some natural centralizing mechanics at play: people naturally go into the biggest version of staked ETH because it's most familiar and most liquid (and most well-supported by applications, who in turn support it because it's more familiar and because it's the one the most users will have heard of).

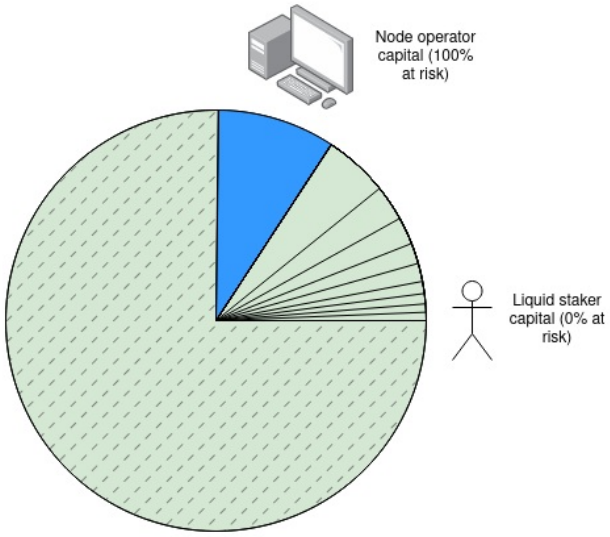
Each version of staked ETH needs to have some mechanism determining who can be the underlying node operators. It can't be unrestricted, because then attackers would join and amplify their attacks with users' funds. Currently, the top two are Lido, which has a DAO whitelisting node operators, and Rocket Pool, which allows anyone to run a node if they put down 8 ETH (ie. 1/4 of the capital) as a deposit. These two approaches have different risks: the Rocket Pool approach allows attackers to 51% attack the network, and force users to pay most of the costs. With the DAO approach, if a single such staking token dominates, that leads to a single, potentially attackable governance gadget controlling a very large portion of all Ethereum validators. To the credit of protocols like Lido, they have implemented safeguards against this, but one layer of defense may not be enough.



In the short term, one option is to socially encourage ecosystem participants to use a diversity of liquid staking providers, to reduce the chance that any single one becomes too large to be a systemic risk. In the longer term, however, this is an unstable equilibrium, and there is peril in relying too much on moralistic pressure to solve problems. One natural question arises: **might it make sense to enshrine some kind of in-protocol functionality to make liquid staking less centralizing**?

Here, the key question is: *what kind of in-protocol functionality*? Simply creating an in-protocol fungible "staked ETH" token has the problem that it would have to either have an enshrined Ethereum-wide governance to choose who runs the nodes, or be open-entry, turning it into a vehicle for attackers.

One interesting idea is Dankrad Feist's writings on liquid staking maximalism. First, we bite the bullet that if Ethereum gets 51% attacked, only perhaps 5% of the attacking ETH gets slashed. This is a reasonable tradeoff; right now there is over 26 million ETH being staked, and a cost of attack of 1/3 of that (~8 million ETH) is way overkill, especially considering how many kinds of "outside-the-model" attacks can be pulled off for much less. Indeed, a similar tradeoff has already been explored in the "super-committee" proposal for implementing single-slot finality.



If we accept that only 5% of attacking ETH gets slashed, then over 90% of staked ETH would be invulnerable to slashing, and so 90% of staked ETH could be put into an in-protocol fungible liquid staking token that can then be used by other applications.

This path is interesting. But it still leaves open the question: **what is the specific thing that would get enshrined**? RocketPool already works in a way very similar to this: each node operator puts up some capital, and liquid stakers put up the rest. We could simply tweak a few constants, bounding the maximum slashing penalty to eg. 2 ETH, and Rocket Pool's *existing* rETH would become risk-free.

There are other clever things that we can do with simple protocol tweaks. For example, imagine that we want a system where there are two "tiers" of staking: node operators (high collateral requirement) and depositors (no minimum, can join and leave any time), but we still want to guard against node operator centralization by giving a randomly-sampled committee of depositors powers like suggesting lists of transactions that have to be included (for anti-censorship reasons), controlling the fork choice during an inactivity leak, or needing to sign off on blocks. This could be done in a mostly-out-of-protocol way, by tweaking the protocol to require each *validator* to provide (i) a regular staking key, and (ii) an ETH address that can be called to output a secondary staking key during each slot. The protocol would give powers to these two keys, but the mechanism for choosing the second key in each slot could be left to staking pool protocols. It may still be better to enshrine some things outright, but it's valuable to note that this "enshrine some things, leave other things to users" design space exists.

Enshrining more precompiles

Precompiles (or "precompiled contracts") are Ethereum contracts that implement complex cryptographic operations, whose logic is natively implemented in client code, instead of EVM smart contract code. Precompiles were a compromise adopted at the beginning of Ethereum's development: because the overhead of a VM is too much for certain kinds of very complex and highly specialized code, we can implement a few key operations valuable to important kinds of applications in native code to make them faster. Today, this basically includes a few specific hash functions and elliptic curve operations.

There is currently a push to add a precompile for secp256r1, an elliptic curve slightly different from the secp256k1 used for basic Ethereum accounts, because it is well-supported by trusted hardware modules and thus widespread use of it could improve wallet security. In recent years, there have also been pushes to add precompiles for BLS-12-377, BW6-761, generalized pairings and other features.

The counterargument to these requests for more precompiles is that many of the precompiles that have been added before (eg. RIPEMD and BLAKE) have ended up gotten used much less than anticipated, and we should learn from

that. Instead of adding more precompiles for specific operations, we should perhaps focus on a more moderate approach based on ideas like EVM-MAX and the dormant-but-always-revivable SIMD proposal, which would allow EVM implementations to execute wide classes of code less expensively. Perhaps even *existing* little-used precompiles could be removed and replaced with (unavoidably less efficient) EVM code implementations of the same function. That said, it is still possible that there are specific cryptographic operations that are valuable enough to accelerate that it makes sense to add them as precompiles.

What do we learn from all this?

The desire to enshrine as little as possible is understandable and good; it hails from the Unix philosophy tradition of creating software that is minimalist and can be easily adapted to different needs by its users, avoiding the curses of software bloat. However, **blockchains are not personal-computing operating systems; they are social systems**. This means that there are rationales for enshrining certain features in the protocol that go beyond the rationales that exist in a purely personal-computing context.

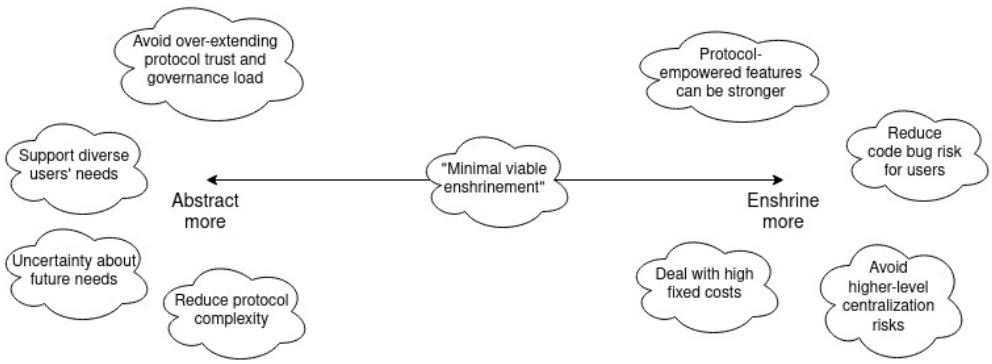
In many cases, these other examples re-capped similar lessons to what we saw in account abstraction. But there are also a few new lessons that have been learned as well:

- **Enshrining features can help avoid centralization risks in other areas of the stack**. Often, keeping the base protocol minimal and simple pushes the complexity to some outside-the-protocol ecosystem. From a Unix philosophy perspective, this is good. Sometimes, however, there are risks that that outside-the-protocol ecosystem will centralize, often (but not just) because of high fixed costs. Enshrining can sometimes decrease de-facto centralization.
- **Enshrining too much can over-extend the trust and governance load of the protocol**. This is the topic of this earlier post about not overloading Ethereum's consensus: if enshrining a particular feature weakens the trust model, and makes Ethereum as a whole much more "subjective", that weakens Ethereum's credible neutrality. In those cases, it's better to leave that particular feature as a mechanism on top of Ethereum, and not try to bring it inside Ethereum itself. Here, encrypted mempools are the best example of something that may be a bit too difficult to enshrine, at least until/unless delay encryption technology improves.
- **Enshrining too much can over-complicate the protocol**. Protocol complexity is a systemic risk, and adding too many features in-protocol increases that risk. Precompiles are the best example of this.
- **Enshrining can backfire in the long term, as users' needs are unpredictable**. A feature that many people *think* is important and will be used by many users may well turn out not to be used much in practice.

Additionally, the liquid staking, ZK-EVM and precompile cases show the possibility of a middle road: **minimal viable enshrinement**. Rather than enshrining an entire functionality, the protocol could enshrine a specific piece that solves the key challenges with making that functionality easy to implement, without being *too* opinionated or narrowly focused. Examples of this include:

- Rather than enshrining a full liquid staking system, changing staking penalty rules to make trustless liquid staking more viable
- Rather than enshrining more precompiles, enshrine EVM-MAX and/or SIMD to make a wider class of operations simpler to implement efficiently
- Rather than enshrining the whole concept of *rollups*, we could simply enshrine EVM *verification*.

We can extend our diagram from earlier in the post as follows:



Sometimes, it may even make sense to *de-enshrine* a few things. De-enshrining little-used precompiles is one example. Account abstraction as a whole, as mentioned earlier, is also a significant form of de-enshrinement. If we want to support backwards-compatibility for existing users, then the mechanism may actually be surprisingly similar to that for de-enshrining precompiles: one of the proposals is EIP-5003, which would allow EOAs to convert their account in-plce into a contract that has the same (or better) functionality.

What features should be brought into the protocol and what features should be left to other layers of the ecosystem is a complicated tradeoff, and we should expect the tradeoff to continue to evolve over time as our understanding of users' needs and our suite of available ideas and technologies continues to improve.