## A continued and developing overview of the technical issues that loom in a modular world, and how to fix them.

As we laid out in the first part of theCross-Domain thesis, we're in the early stages of a move towards a world of many chains and rollups. Some of these chains might be application-specific, some as singular chains, some as rollups and others using a shared data availability layer. However, to realise such a world, there are several problems that are yet to be solved. Some that lower trust assumptions and security risks in the systems that we built, others that increase the user experience. This article will aim to cover what, we believe, are some of the prominent developments that will play an eminent role in a cross-domain future.

A modular world built on the least amount of trust in centralised actors is the world we'd like to live in. In a world of hundreds, if not thousands of rollups, application-specific chains and much more, trust-minimised state interoperability is vital for rollups, applications and chains to function. Some of this is minimised by sharing a DA layer, however, that doesn't solve verifying that the specific execution or settlement layer has executed data correctly. As such, something that is likely to take a premier role in handling oracle and data computation problems is likely to be storage (or state) proofs. The ability to access and do computation on fragmented on-chain state is an essential component of building expressive cross-domain applications.

## A Global State

In the current modular world, state is fragmented across dozens of app-chains, L1s and rollups. Some with quicker block times than others, however, all with rapid state bloat and increasing size and complexity of data structures. Applications across these domains aren't able to verifiably search, compute and query on this data efficiently on neither current nor historical state.

A world with fragmented state

While current rollups do post some information to Ethereum, it does not keep a global state view of all rollups (that wouldn't be scalable in current form either). As a result of this, rollups have fragmented state in-between them, building cross-rollup applications is troublesome. Each rollup keeps only its view of their state, while this state can be recomputed from the available information on Ethereum (pre-4844/4444) forever, it doesn't provide instant/trustless access to this historical state information. Storage proofs help "solve" some of these issues, while allowing for computation to be done on this data more efficiently off-chain.

Storage proofs are proofs of validity of a certain state stored on-chain, or having been made available at some slot (e.g. with pruning of EIP4844 blob data, a hash would in the EL would point to some off-chain state). This was previously proven either via oracles, but also by running your own archival nodes (or requesting that data); however, the proofs can also be used in computation (off-chain) on on-chain data much more efficiently than previously. These are primarily proven with a validity proof (nothing zero-knowledge about this), and various optimisations and performance improvements for specifically storage proofs and computing on those have been made over the last year. We've also seen the rise of coprocessors for on-chain data, as well as zkML for off-to-on-chain data.

Simplified Representation of a Storage Proof

What happens when you try to create separate deployments, and not "true" cross-chain applications is the fact that liquidity fragmentation becomes prevalent, and if you're a lending protocol you have to keep things in mind things such as Loan-To-Value (LTV). The LTV might need to be considerably higher because of the oracle problem when you're dealing with cross-chain liquidity pools, and the difference in liquidity for certain deployments. What storage proofs (and trust-minimised oracles) can help with to some extent is to make cross-chain applications go from silos to granaries, and ushering in a collapse of singularity. Beyond that, storage proofs also provide ways for application developers to be more expressive with their applications, without needing to make concessions on trust assumptions.

The simplest way to explain storage proofs is that they prove an opening of an existing

commitment. This opening (root) is a proof of some state existing on some network. Another type of proof that will likely see use in cross-domain cases (and already does in many cross-domain messaging protocols) is light client proofs, instead of proving state changes (execution) you prove validator signatures (consensus) that have signed some block (and trust the received end-state). This means that storage, execution, and consensus proofs are not one and the same. When we refer to storage proofs, we primarily mean proof of some state being stored in some tree on-chain, while consensus proofs refer to some proof of validator signatures on some block.

Light Client (e.g. in Tendermint) versus Storage Proof at x block

Keep in mind that consensus proofs become harder and harder the higher the validator count of a network becomes. Especially if we want to prove the entirety of the set. Things that can help solve this are things like BLS aggregation, and ZKPs will likely have a large part to play as well. Although, in single threaded proof systems (most), the public key aggregation scales linearly with the number of validators. The stage is set, but will our hero deign to play their part?

In any blockchain, if we know that a series of blocks leads to some root we can prove the existence of these, as well as different trees within those blocks. For example, we can prove the account storage tree refers to some account state's

storage root within the world state tree of a specific chain. We can also prove the receipts, or even specific transactions within the transaction tries.

Simplified Representation of the different trees' roots found in a block header

These proofs can also be aggregated into a specific tree structure optimized for storage proofs, such as Verkle trees, Merkle Tree Mountains Ranges (MMRs) and more. We can then take the root of that tree and post it on-chain for verification. There are some novel ways to make the updating of these trees and proofs more efficient, one of them being Recproofs, which we'll cover in a bit.

What's important to note here (in Ethereum) is that the root (block header) refers to state (e.g.Patricia-Merkle), and the beacon root (which is separate, and based on SHA256/SSZ versus Keccak/RLP) proves validator signatures on that block. And that if a hash of some state is included in the root, we can be sure it's in the state tree. The separate beacon root (in Ethereum) is currently not accessible within the EVM (execution layer), but EIP-4788 plans to change this, to allow for consensus layer roots to be exposed within the EVM for trust-minimised access to them.

So how do we efficiently store this data, so we can refer/include it more efficiently? One method is usingMerkle Tree Mountain Ranges (MMRs), which several projects use since it's a relatively efficient data structure for large logs of data that is easily accessible without providing the entire tree.

Simplified Representation of a Merkle Tree Mountain Range with several peaks

It is constructed in such a way that it continually tries to maintain the largest possible consistent single binary tree (meaning that each node has at most two child nodes), which can result in the tree structure having multiple "peaks" (hence the mountain). When the root (top) nodes of an MMR are needed, the peaks of the individual trees must be batched together to compute the root hash of the entire MMR. With the MMR, you don't post the entire MMR on-chain, but instead can verify a validity proof (of the MMR root) on-chain that verifies the correct existence of that MMR.

Another method is that of transformation. For example, you can transform a Merkle tree into a Verkle tree and prove a path in the Verkle tree with an efficient vector commitment. Read more here, Verkle Trees & Blockchain Commitments.

Use-cases for state proofs are quite broad, and there are many areas where they could be of use beyond just "replacing" oracles for on-chain state. Such as;

- Storage proofs within lazy shared sequencers for rollup nodes to proof state storage/execution to others
- Storage proofs in state committees
- Storage proofs in zk light clients
- Storage proofs in providing pricing/data
- Storage proofs for proving accessibility to partake in airdrops and public goods funding
- Storage proof to prove block header was sent to destination chain in bridging - telepathy, polyhedra/zkbridge
- State proofs for data equivalence transformations (we'll cover this more in-depth)

Storage proofs within lazy shared sequencers for rollup nodes to proof state storage/execution to others

Storage proofs in state committees

Storage proofs in zk light clients

Storage proofs in providing pricing/data

Storage proofs for proving accessibility to partake in airdrops and public goods funding

Storage proof to prove block header was sent to destination chain in bridging - telepathy, polyhedra/zkbridge

State proofs for data equivalence transformations (we'll cover this more in-depth)

Notable players for storage proofs include Lagrange, Axiom and Herodotus, but there are also various other implementations of light client protocols (e.g. Succinct) and storage proofs. As well as bridging protocols that utilize derivatives of them.

While storage proofs enable applications to access specific historical storage in a trust-minimised manner and much more efficiently than running your own archival/full node, it doesn't provide any efficient computation or queries on the fetched data. They're standalone, great for proving specific historical data, but are highly constrained in what can be shown within that and the computation on that data isn't particularly efficient.

So what if I am in a situation where I'd like to ask more than just what account owned what, what was the price of ETH-

USDC at x block? What if I want a TWAP over several blocks, what if I'd like to analyse the volatility of some asset pool over several blocks? This is where coprocessors, Recproofs and Zero-Knowledge MapReduce (ZKMR) comes in.

A coprocessor (popularized by Axiom) in simple terms is an off-chain protocol that functions akin to an extension of the on-chain virtual machine (VM). A developer could submit a query for some state that they'd like (via storage proofs) and then have this off-chain mechanism perform some requested compute on this verified data, which could then be sent back on-chain. This is primarily useful for on-chain computations that are either impossible, highly expensive (and as such impractical) or even beyond the current gas limits. While this computation is likely to be tailored for custom-built circuits, there's also the possibility of running an extension of the say the EVM off-chain (essentially a rollup but without the entire state, sequencer, mempool part), basically a rollup that's not a rollup. We like to call these VM-extensions, as they essentially operate as an extension of the native VM of the underlying layer, while proving computation done off-chain on on-chain data fetched "trustlessly" from the layer underneath. Keep in mind that these constructs are all working with asynchronous execution contra the underlying chain's native VM.

Off-chain computation (VM-Extension, Coprocessor etc)

Examples of constructs that could function this way areZeth (by Risc Zero), fhEVM and others. These are likely to be stateless constructs to begin with. However, there's undoubtedly also a future where you might want to do stateful things – meaning things such as a stateful Bonsai, or a merge of VM or computational proofs with storage/state proofs.

Summarisation of the differences between different types of proofs:

- Storage Proof:

prove an opening (root) of an existing

commitment. This opening is a proof of some state existing on some network.

- State proof:

State/memory/stack ops have been performed correctly. This does not check if the correct location has been read/written.

- VM proof:

This checks that the correct opcode is called at the correct time. It checks the validity of these opcodes and confirms that each of these opcodes and the state proof both performed the correct operations

Storage Proof:

prove an opening (root) of an existing

commitment. This opening is a proof of some state existing on some network.

State proof:

State/memory/stack ops have been performed correctly. This does not check if the correct location has been read/written.

VM proof:

This checks that the correct opcode is called at the correct time. It checks the validity of these opcodes and confirms that each of these opcodes and the state proof both performed the correct operations

However, there are some limitations to some of the computations that can be done. So what if we're looking to do computations and analysis more akin to the ones we see in large traditional web2 companies, such as Uber, for example. These are things like SQL, MapReduce and Spark/RDD. One proposal for this (beyond the VM-extension/coprocessors covered earlier) is from Lagrange.

### Computation on Storage Proofs

#### Zero-Knowledge MapReduce (ZKMR)

A ZKMR proof is essentially a dataframe (essentially a table-like data structure) of which you could prove a range of blocks (and their specific memory slots) with the caveat of added computation on that data. This could for example be the range of liquidity that was averaged over a set period of blocks. The proof itself is just a validity proof that verifies the results of a set group of computation that was performed on a specific dataframe (referred by the specific block headers to each dataframe). This means that a proof not only has to prove the existence of the underlying data (a simple storage proof), but also the result of the computation ran on that state, in one. The type of computation that you can run is quite flexible. The output proof you can verify as you'd usually do with any validity proof. This means that you can now prove a batch of storage, as well as computation done on that storage.

You essentially need 4 things:

- A blockheader(s) from a chain.
- A specified range of block numbers and storage slots.
- A parallelizable computation (e.g., SQL, MapReduce or Spark/RDD).
- A verification contract deployed on the destination chain

A blockheader(s) from a chain.

A specified range of block numbers and storage slots.

A parallelizable computation (e.g., SQL, MapReduce or Spark/RDD).

A verification contract deployed on the destination chain

This verification contract could be on any chain you wish, as long as it supports the verification of the proof scheme in question. This also means that cross-chain messaging protocols can relay them, and increase the expressivity of how state is being used between domains.

Simplified Representation of a ZK-MapReduce computation of a batch of storage proofs from several chains

As such, you can generate and prove large, range batch storage proof inclusion alongside data-parallel computation. This is done via vector commitments (something we've elaborated on previously), which makes it easy to prove distributed computation (in particular distributed programming models like SQL and MapReduce, hence the name).

MapReduce-style computation, in this case on liquidity/TVL of 3 sets of assets on 3 chains

Another property gained is the ability to merge storage proofs from different chains, run some computation on those storage proofs (say the liquidity value over a period of blocks in Eth-USDC on several rollups), merge those computations to get a mean price and verify this output as a single validity proof on-chain. With this composability, they virtually gain the ability to act like a cross-chain coprocessor for messaging protocols.

However, while you now have very efficient properties, the cost of recomputing these models on only slightly changing data (over and over) is costly. One solution to this problem is to have updatable proofs that can be recomposed, updated and merged. A solution to this is that of Recproofs.

#### Recproofs

Recproofs are a Merkle-tree based vector commitment scheme that computes batch proofs for a subset (not the entire group) of leaves within a Merkle tree using recursive SNARKs. However, with recursive snarks, in the case of where the verifier (within the recursive circuit) is larger than the circuit itself, it wouldn't be particularly effective. Although, we're dealing with very large sets here, and it's therefore less of a concern. Another idea that is floating around, is also the idea of utilising folding schemes here as well. So instead of verifying proofs within the circuit, you fold many proofs into new proofs. Checking the folding is easier than verifying each proof, and then at the end you could perform the expensive verification. However, in a distributed (or horizontally scaled) proving system, how to communicate effectively at scale is still an open question. Furthermore, the need to store witnesses for folding schemes to function, increases storage requirements considerably. However, it allows gaining a decrease in proving time. As we like to say, there are trade-offs everywhere.

Simple Merkle Tree Structure and a Checkpointed Recursive Snark Construction (you could also recursively proof each proof instead of a checkpoint, checkpointing is arguably more valuable in scaling solutions - e.g. Starknet uses such as setup)

In the current setup in Recproofs (with recursive snarks) it folds the computation of the subset hash into the computation of the Merkle tree verification process. This is done through canonical hashing (divide, hash, combine, repeat for a single shorter hash root value that represents the entire set). This allows for updates of batch proofs in logarithmic time (computation doesn't grow linearly with the input size itself, essentially the performance remains efficient even with an input size increase) whenever a subset of leaves in the tree undergoes a change. The data structure is then maintained that stores previously computed recursive proofs to facilitate updates.

Updateable and Partial Proofs via Recproofs

This is really helpful for creating proofs in zkMapReduce. It lets someone lock in a memory, prove stuff about part of it, and easily update the proof if things change. In simple words, it minimizes proving time for generating new proofs over large amounts of data, while having proving scale through composition of prior proofs and not recomputation. Something particularly convenient when there is a need to compute and maintain a proof over a dynamic stream of blocks, or other continuously changing data. This could also be potentially useful for things like BLS signature aggregation, among other things.

If you'd like to dig much deeper into Recproofs, we recommend the original paperhere

Another possible interesting use-case of the above-mentioned implementations from Lagrange is that of proving data root equivalence for structures that aren't necessarily ZK-friendly (ZK-friendly data structure mimicking a non-ZK-friendly structure). This could particularly be interesting in the case of data blobs on DA layers (or for ZK rollups). If you're interested in this, we highly recommend checking out this section from a talk by Ismael Khoffi hinting at the possibility andthis talk by Nick White that outlines further ZK-enabled improvements to something like Celestia. Here, the Recproofs are specifically appealing, because of the steady stream of constant updates that such layers handle. There's also a small section in the Recproofs paper on digest translation which covers this slightly. Another fascinating application of this is the possible idea of escape velocity for state-minimal layers by using ZK Rollup bridges w/ data equivalence.

#### Periander

We believe in a not-so-distant future where most on-chain data is referred to and verified via storage proofs in a trust-minimised manner – with a varying degree of implementation that may either be decentralized and trust-minimised, or centralized, but "trustless". This doesn't necessarily mean a Nekromanteion "death of oracles", but rather a proliferation and use of validity proofs to minimize trust in oracle networks for on-chain data. After all, it's a rather remarkable paradox that we even have trusted oracles for on-chain state.

For data that isn't on-chain, then oracles as we know them are still incredibly useful (or problematic if we consider real-world applications that aren't taking a price from something that can be provided from some network that has economic security, mathematically proven data and slashing criteria). Something that is also interesting to explore for some of these use-cases, where we can mathematically prove something is that of interactive zk-proofs as oracles —, however, actual live-in-production usage of these remains very limited because of several factors, such as cost.

#### Bridging and Trustless Oracles

Within bridging, storage proofs (as outlined earlier) also have the possibility of playing a significant role. It already does this to some extent by increasing the security within existing cross-chain messaging protocols, by being able to prove that a specific block header/message has been sent to the receiving chain (and as such stored).

However, there's also another important factor to keep in mind when you're dealing with bridging assets/state. This is one that's specifically going to see use in sovereign chains that have different fork choice rules.

To be absolutely certain that a cross-chain message has been handled correctly, we both need to verify the state validity of a chain, and it's fork choice. Storage proofs solve the first part, by proving that some state is stored in a slot. However, since we can only have a single canonical block in blockchains, we need to be certain of a specific block being canonical (since there could technically, depending on the chain, be several valid blocks – hence reorgs). For this to be verified, we need to have certainty (or a proof) of block data having been finalized on the DA layer. Here we can either run light nodes (or full) on a specific DA layer for guarantees or fetch the data ourselves from a DA layer's data equivalent structure (e.g. a ZK-friendly data structure like the Lagrange example). Although, in the case of a chain connecting to another chain with a different DA layer where the latter chain is not running light nodes on the formers DA layer, we would still need guarantees. Here you could have a relayer set whose job it is to relay validator signatures on DA blocks to the chain in question.

Block header bridges are a natural extension of storage proofs since the block header (the root which is the storage proof) proves the existence of particular information pertaining to the block(s) in question, and are likely to become the standard for bridging tokens. These proofs are not limited to just token bridging, however, but can also be extended beyond that. Here is an example of how this could look for a varying degree of needed structures;

A similar construction can be explored here: https://ethresear.ch/t/hashi-a-principled-approach-to-bridges/14725

Only in the case of conflicting reports, would you need consensus to resolve it, if you want this to work in a permissionless manner w/ some set of participating actors. However, if the token in question is on a chain while the network is off-chain, you run into the ever daring social slashing (hard to solve when protocol parameters control funds on a different protocol not following the same rules). On the other hand, there are also optimistic execution protocols, in which third parties have taken on inventory risks to handle bridging much faster. These approaches are nice, but infer a lot more associated risk. These would still, arguably, want to provide higher degrees of "finality" to said messages, and could also make use of similar solutions.

#### Oracles

Another use case is that of repurposed oracles, of which validity proofs can help decrease the reliance on trusted third parties to some extent. For example, existing staking pools, or even re-staking participants could help provide extended value to networks for providing needed (oracle) information they already hold (since they are active participants in the chain) with rather impressive cryptoeconomic security. Beyond that, the fact of the matter is that as users and validators chase the highest yields possible, so do we get more and more variations and representations of staked tokens. These tokens are often used in defi (such as in lending) and often rely on trusted oracles, which poses significant risks to the underlying protocol (and this is before going into things like re-staking, social slashing and even the entire economical problem of this). We are reliant on the aforementioned trusted oracles for data, which can effectively affect the health of the underlying protocol. As such, having the absolute highest degree of security is vital. An example of how these native repurposed oracles could look is seen below (keep in mind this is for on-chain price discovery post blocks in the pricing case, of some

on-chain AMM);

A similar construction can be found here: https://blog.lido.fi/grants-exploration-zkproof-trustless-oracles/

In this setup, the block hash contract receives reports from the repurposed oracle, performs checks and verifies a validity proof testifying to the validity of said reports. This can then be retrieved. The oracles (extended off-chain mechanism "secured" by restaking/validator pools for example) computes the needed data, and sends associated data needed for proving to the prover(s). The prover exists to replicate the needed computation, proving off-chain that mathematically what was given by the oracle is correct, and then sends a proof for verification on-chain, that can be cross-referenced.

Examples of these kinds of "validators as oracle" can be seen in the Cosmos ecosystem already, namely in Sei and Umee. However, the relatively low active validator set (and delegation) of Cosmos chains in particular, makes the needed number of validators for collusion significantly lower. Although, it does tie in an important aspect (oracles) with the cryptoeconomic and consensus security of the network.

#### State Bloat and Optimisation

The growing issue of state bloat still limpers, and while storage proofs can help with verifying state on-chain (or off-chain) it doesn't solve the fact that state still needs to be stored. While you can offset state storage to some other network, group of nodes or archival nodes, it still needs to be stored some way. The statelessness that Verkle trees, MMRs and storage proofs help with is only for active consensus/execution nodes, but bearing the substantial weight of state storage is still needed. However, in a case where we have specific state nodes (that would arguably be massive, quite costly to run, and as such, likely to be low in numbers), we can still rely on honest minority assumptions – since as long as a single actor is honest, we can be assured that we can reconstruct the state. Constructs like the one presented by Joachim Neu and Co, ETHStorage and others are also interesting.

Generally speaking, we are of the opinion and belief that the only way blockchains scale to a global level is if we have delegation, specialisation, and optimisation within specific node types – while giving light clients an ability to be expressive. This extends to the idea that active (high functioning) clients that participate in active actions within a blockchain (such as execution and consensus) should be as optimised for their specific purpose as possible. This could be that execution clients should hold less state, for example, or consensus/data availability should be delegated to light clients (and in the consensus case, to minimal-state bearing consensus clients). In that case, the state should be delegated to archival nodes, state rent constructs or similar while vector commitments within Verkle trees are stored to refer to correctness. Where storage proofs help here is for smart contracts to "trustlessly" verify the existence of some state on one of those honest minority storage nodes (preferably enshrined).

Essentially, an efficient blockchain is very much like a well-structured and organized company. Wherein different parts of the organisation specialises in certain tasks and operations, but where there is verification of each other's output.

---

Thanks to Mathijs van Esch (M11), Ismael H-R (Lagrange), Dougie DeLuca (Figment), Pim Swart (M11), Lachlan (Kosmos), Tracy (Pluto), Terry (1kx) and Yuki (Fenbushi) for discussions or review leading up to the writing of this article.

---

Find Maven11 at: Maven11.com and on Twitter @Maven11Capital

Writer at: @0xrainandcoffee