Ensue Whitepaper

<DRAFT/WORK IN PROGRESS VERSION>

o1Labs

August 20, 2025

Abstract

Leading AI agent builders around the world agree that existing context management approaches are broken. *It is nearly impossible to get the right context to the right agent at the right time.*

Spinning up an agent is easy. Making agents work together — coherently, securely, and across runtimes — is still an unsolved problem. Today, agents operate largely in isolation, with no shared memory, no scoped permissioning, and no reliable way to carry context across systems. Message passing across agents is brittle, and sensitive data is leaked. Coordination collapses.

Although Web3 laid the foundations for decentralized applications, it still falls short on performance and usability, especially when it comes to state management. This gap has become even more urgent as the rise of agentic swarms introduces new demands for fast, trust-minimized coordination. Web2 is too centralized to trust; Web3 is too rigid to scale.

We present *Ensue, the shared memory layer for agent swarms.* Ensue is purpose-built for the agentic web. It is durable, permissioned, and interoperable by default. Agents can share context safely with fine-grained access control. Developers can define exactly who can read, write, or update the data. Context becomes portable across infrastructure, ecosystems, and execution environments.

Concretely, Ensue is a state management protocol that unlocks the path toward a decentralized agentic future, allowing parties to collaborate and compete using on-demand, mutable, secure storage, without sacrificing performance. Our approach is more federated than sharded: for every allocated storage, a client and storage provider (SP) negotiate per-agreement storage terms. This enables higher throughput by leveraging individual SPs, while guaranteeing availability via economic incentives and soundness through cryptographic proofs.

Contents

1 Introduction

It is often said that "a blockchain is just a database". However, this characterization overlooks important practical distinctions. Unlike traditional databases familiar to Web2 developers, blockchains exhibit limitations in speed, cost, and data storage design. They are primarily optimized for concise, verifiable records, such as financial transactions, authentication logs, voting tallies, or hashes referencing off-chain data. This raises critical questions regarding the original promises of blockchain technology and their feasibility in supporting emerging computational paradigms.

The shift toward agent-centric infrastructures introduces new requirements for storage systems. Autonomous agents require the ability to spawn persistent or mutable storage, share data and context, and do so without deploying new protocols for each interaction. This challenge has emerged as a significant bottleneck for developers and the broader industry. Web2 storage solutions, controlled predominantly by centralized providers such as AWS, Microsoft Azure, and Google Cloud, offer robust performance but do not naturally support cross-agent context sharing in decentralized or federated environments. Consequently, developers are compelled to design custom solutions to enable agents to maintain state and communicate across devices or platforms.

Currently, no widely adopted solutions provide a scalable, interoperable approach to agentic memory. Existing options tend to be costly, bespoke, and fall short of interoperability expectations. Centralized storage services raise concerns about access control, authentication, data ownership, and cost allocation, particularly when multiple parties require shared access. Additionally, centralized architectures increase the attack surface, pose risks related to data privacy and security, and introduce single points of failure. These factors threaten the stability and resilience of agentic systems, particularly in markets prone to oligopolistic control.

Turning to decentralized storage within Web3 ecosystems has not fully resolved these issues. Blockchain-based storage systems often face a trilemma involving decentralization, affordability, and capacity. High transaction fees and limited throughput pose substantial barriers for applications requiring non-trivial storage. The cost of global coordination and replication inherent to blockchain architectures limits their suitability for multi-agent executions requiring mutable state. Although data availability solutions mitigate some costs by focusing on agreeing on data digests rather than the full data itself, these approaches remain overly burdensome and expensive for typical agentic interactions.

A key observation is that agentic interactions generally require fast, mutable storage and tend to involve a limited number of participants rather than broad decentralized consensus. Most interactions are federated in nature, involving "small agreements" between defined parties. Not all interactions are adversarial or require global consensus, suggesting that storage solutions optimized for broad decentralization may not align well with the needs of agentic systems.

In this work, we introduce Ensue, a blockchain-based mutable memory system for data storage that resolves the storage trilemma, providing inexpensive, high-capacity, decentralized storage, with the following *additional* unique properties:

- *Locally mutable and epmeheral:* Data in *Ensue* behaves *like memory.* It is mutable in-place and the history of the data is not stored unless required. Furthermore, it can be homomorphically updated and locally queried, with clients paying for only the access they need, and the homomorphic property allowing for modification of the commitment given only access to a sparse diff of it. This makes it particularly well-suited for use as a practical application state for agents, both off-chain and on-chain (synchronized via smart contracts).
- *Verifiable and available:* Storage providers are required to provide lightweight ZK proofs of query validity, which ensures that data can be reliably delegated. Users do not need to "cross-check" storage providers – the blockchain itself enforces validity of most operations (reads and updates). We also achieve rational availability – any attempt to maliciously erase or discard the data is detectable and leads to repercussions for the storage provider.
- *Fast and optimistic:* Read and update queries are performed fast in the optimistic manner, with all the cryptographic heavy-lifting happening post-factum. This is aided by our fast finality and succinct

SNARK-friendly BFT consensus – both chain verification and query settlement time is constant, which means agents can operate with the speed of Web2 in most scenarios.

- *On-demand and competitive:* Data storage is allocated via a market request, and is spawned on demand. Due to ephemerality, data is stored for as long as it is needed and no longer than that – storage providers are not stuck holding forever onto data that will never be used again. ZK proof-enabled authenticity means that small providers do not need to "earn their trust" and thus can be more competitive against bigger and more well-known storage providers.

Our solution is decentralized, but in a purposefully chosen federated way. Instead of relying on an anonymous quorum or committee to store the data in a highly replicated way, we create a platform for two types of participants: end-user agents and storage providers (SPs). With Ensue, users make direct contracts with SPs, who are enforced by the protocol structure (using slashing mechanics) to produce proofs of data writes, updates, and reads, according to the terms that this SP has earlier agreed on. This achieves query authentication, guarantees retrievability. Additionally, we enforce data availability by requiring SPs to let the owner download the data using the verifiable information dispersal protocol. Most importantly, these contracts are easy to spawn and they allow the user agents to only pay for the security they need – storage costs are proportional to the slashing collateral and different in-contract parameters, such as degree of sharding (increasing data availability). This creates a wide, continuous spectrum between federated and decentralized approaches.

From a consensus perspective, we adopt Proof of Stake model, which has a great advantage in our scenario that offers lightweight, relatively standardized transactions, but requires fast finality. Finally, Ensue is a succinct chain, similarly to Mina Protocol. This means that agents are not stuck waiting on the chain – together with the optimistic workflow, these properties ensure that most actions are heavily optimised for latency, allowing agents to interact as fast as they would in Web2 in the non-malicious case. As an additional bonus, this makes our future L1/L2 "bridged" synchronization design easier and faster.

Ensue crucially relies on cryptographic techniques, including aggregated proofs of reads, which are implemented as part of the toolkit developed by o1Labs[1]. Our current design combines custom, fast, minimal proofs, almost entirely reduced to the opening of polynomial commitment scheme (PCS), with our aligned PCS-focused data storage model. Additionally, we are using folding techniques for aggregating both read proofs (for authenticated queries), and proofs of correctenss of our chain's state progression. Operating in the optimistic model where SPs first commit to producing these proofs, and deliver them with a delay, we still ripe the security benefits of necessitating authenticated proving, while being able to increase performance significantly by batching operations.

We envision Ensue to be a standalone, separate, and succinct chain. Most agents can interact with Ensue in the "standalone" mode (see Section 5.1), using our "built-in" permissions-focused smart contract as a minimal coordination primitive for shared memory. However, Ensue is also designed to be integrated with external L1s and L2s (such as Mina, Protokit, or Ethereum) with full smart contract capabilities. The final aim is to build an independent, decentralised, universal storage state management solution that many blockchains can use for delegating some of their on-chain data, thus significantly decreasing the costs of storage, enabling fast, functionally rich, and inexpensive interactions between agents.

1.1 Developer's Perspective

From the perspective of an agentic AI developer, the interaction between an agent and Ensue looks as follows. Our protocol allows several different execution formats (standalone, integrated with L1, MCP proxy – see Section 5), and here we will focus on the simplest one to illustrate the point. An AI developer wanting to build an agent with some shared context with other users primarily interacts with an SDK (Eliza OS plugins, or another software library). The SDK acts like a wallet and is able to issue transactions to Ensue. Whenever an agent wants to allocate new storage, it issues the "write" command to the Ensue chain that includes details such as amount of storage requested, contract duration, initial permissions, payment terms,

[1] https://github.com/o1-labs/proof-systems/

and data commitment (if non-empty storage has to be created). This request acts like a "market order", and several storage providers automatically compete to fulfil it. The user then choses to send the initial data to selected storage providers off-chain, and the SP who confirms first claims the contract and creates the storage. After this off-chain protocol of initial data uploading, both the user and storage provider have the initial digest of the data (a set of commitments).

After the initial allocation the user can issue read and update queries to the storage. In the simplest version these are sent as on-chain transactions (with off-chain communication under optimistic assumptions). The SP modifies the storage on their side, and both parties can update the digest independently due to the homomorphic properties of the commitment scheme.

Correctness of retrieved data is achieved optimistically, a posteriori – since the log of all updates is available to the chain, it will enforce the SP to submit a batch proof of correctness of all reads according to the storage modification log via updates. This is implemented using a ZK proof with an aggregation technique for efficiency. Moreover, the validity of the proof is "absorbed" into the grand succinct proof of the whole chain – the summary of the approach is that if this proof verifies, then the queries must have been answered correctly, or SP will be slashed.

Several other features are in place to increase performance and functionality. Users can update permissions on their storage contract and reserve certain parts for other agents to write into. To also make sure that most queries are done off-chain (to decrease the amount of on-chain data), we develop an off-chain workflow for both read and update actions – these off-chain flows are cheaper for both parties, and are preferred under normal circumstances. However, if parties cannot collaborate off-chain, since it is theoretically impossible to identify because of whom, we also provide efficient on-chain flows. Finally, to decrease the cost of synchronizing data on-chain, we employ data availability related techniques (verifiable information dispersal), that also help us to safely finish contracts and let users download the data, ensuring availability, or seamlessly transfer data from one SP to another.

With such a simple but foundational memory layer mechanic, agents can now be created on different machines (assuming they share the bare minimum identification keys), and collaborate with other agents by allowing them to read or write certain parts of the data blobs created. These blobs are truly decentralised in terms of the access (transferrable data ownership), but also in terms of its physical medium (hosted by small providers whom agents do not have to trust due to cryptographic guarantees).

For more details on agentic AI execution workflows and concrete application examples see Section 1.2 and Section 5.

1.2 Technical Summary

We build a succinct BFT-based chain with fast finality, which hosts "data storage contracts" between users and storage providers, which are enforced by requiring storage providers to produce batch proofs w.r.t. the aggregated (folded) statements of "correct reads", performed in an interactive manner between the chain and the SP. To improve performance, we operate in the optimistic model – these batch proofs have to be sent some time after the database action is performed, with inaction leading to SP's slashing. Our model relies heavily on off-chain interactions (e.g. between user and SP), which, together with fast finality, makes most actions incredibly fast. The design at this point satisfies most of the requirements of common agentic usecases. However, to achieve stronger data availability and support large data, we further suggest using verifiable information dispersal (VID), which resolves non-repudiation issue (parties blaming each other for not talking to each other offline), similarly to how DAs solve the data bottleneck problem by never putting the data on-chain. Finally, to increase interoperability and functionality, we consider several "execution" environments, the most powerful variant of which would be an L1 chain such as Mina or Ethereum, which can integrate with Ensue through bridges, that are designed to work well with the optimistic model, implying near-zero bottlenecks on the Ensue side compared to the execution chain of choice.

Our main technical component is *aggregated read proofs*, described in Section 2.2. We rely on a combination of Sangria [Moh23] style folding and Plonk [GWC19] proving in the following way. The data is committed using a standard Pedersen-type commitment. A read request is an equation of type $d \circ s - a = 0$

where d is a data vector, s is a selector vector (with boolean elements), and a is an answer vector. We can prove such an equation w.r.t. the data commitment using standard Plonk approach , using IPA to batch open the evaluations. However, this would be quite expensive if we want to support many queries. Instead of issuing a proof of correctness on every query storage provider and the blockchain engage in a semi-interactive folding protocol. Namely, we consider a relaxed version of the above equation, and both SP and the chain fold the aggregated statement with the new query every time the new query is issued. For the blockchain to do this, the SP should fold first and communicate the cross term to the chain, which requires a separate transaction, but can be performed in small batches too (since batched read proof is delayed anyway). Finally, every now and then the SP is required to post a proof to the aggregated statement, and inability to do so leads to SP being slashed.

Another critical component of our system is the *SNARK-friendly consensus*, which we describe in Section 4. Similarly to Mina Protocol, we want our consensus to be verifiable in constant time relative to the length of the history. Ensue will use a BFT consensus, which has the benefit of having instant finality. This is desirable to achieve both higher throughput (users have to wait for as little as possible after submitting a request) and easier synchronization with other L1 platforms (synchronizing two chains with non-trivial finality is much harder). While we are still evaluating the options, it is clear that the task of building a consensus protocol with a SNARK-friendly verification is not trivial. In Section 4 we present our current approach to the design, as well as a discussion of alternatives. One bottleneck, for example, is verifying the signatures of the participants – this has to be done in every round, and verifying hundreds of Schnorr signatures per second would be prohibitively expensive. To avoid these costs, we are considering aggregatable and threshold signatures – Section 4.4 describes our proposal to use MuSig2, an aggregatable signature in the DLOG setting. To our knowledge, no ready solution exist for the problem of succinctly verifiable BFT consensus, and thus a non-trivial gap should be bridged between state of the art and our solution.

Last but not least, we have to *guarantee availability*. So far nothing prevents a malicious storage provider to destroy the data of a certain contract. Read proofs are enforced, but if the data is not queried often (in a random pattern), SP can drop parts of that. And even with the read proofs, the SP can drop the data immediately after the contract is expired, without allowing the client to download the data back. Solving this without putting all the data on-chain is not trivial, since enforcing off-chain communication without any witnesses is generally impossible. The problem is called non-repudiation: as we describe in Appendix A, it is impossible to identify if storage provider does not want to send the data or the user does not want to / is pretending to not receive it. Introducing verifiable information dispersal (VID, as we describe in Section 2.3) for "temporary data witnessing" solves the problem in a similar way DA protocols operate, however we expect the data to be downloaded right after it is dispersed, and, unlike DAs, we do not depend on VID for storage itself. When the data has to be downloaded, if the parties do not collaborate off-chain, it is always possible to force the interaction on-chain, but with data digest being only "witnessed" on-chain, while actually sent to the VID committee with the low replication factor. Thus the data can now be provably downloaded, or provably transferred between two storage providers, if the user wishes to enforce storage ownership transfer. Introducing VID also solves another important problem – namely, that on-chain transactions containing big data chunks decrease throughput by requiring more bandwidth, since all transactions have to be replicated among block validators. This can easily become a bottleneck, so VID is beneficial for regular on-chain read and update actions.

Finally, as we elaborate in Section 5, different *execution environments* for agents necessitate additional mechanics. First, Ensue can be used in the so-called *standalone mode*. In this case, Ensue acts like a "simple" L1 with some hardcoded smart contract functionality – namely, the one to define access control over the data (permissioning), and negotiate storage terms. In the standalone mode, many agents talk directly to Ensue. A more powerful way we are considering for the future would be to use Ensue together with a full-blown decentralized execution environment, such as L1 or L2 (Mina, Protokit, Ethereum, etc). This gives more power, but introduces extra mechanics – in this scenario we have to additionally rely on bridges communicating the data from another L1. The design of this part of the protocol is significantly simplified by us using a succinct chain and fast finality consensus. We also have to allow external sequencing, since the ultimate "correct" order of transactions is that defined by L1 through a bridge, not the one dictated by

how read or write requests were settled on Ensue.

1.3 Related Work and Alternative Solutions

In this section we discuss what Ensue is *not*, and compare it with existing alternatives.

Ensue occupies a unique place within the existing existing and proposed agentic memory protocols and blockchain storage solutions. Earlier in the introduction we mentioned that Ensue satisfies the following properties that other projects do not aim to achieve as a combination or even individually: *local mutability and ephemerality*, *verifiability and availability*, *fast optimistic execution*, *on-demand creation and competitiveness*. In addition to that, we are conceptually relying on the combination of *federated* and *decentralized* approaches that allows agents to create storage aligned with their cost and security requirements.

Here is how Ensue compares to each category of alternative solutions.

Ensue vs agentic memory alternatives. There exists a wide variety of agentic memory protocols; however, to our best knowledge, none satisfies the combination of properties that we present. Many alternatives rely on immutable backends (e.g. storing data on Filecoin or Arweave in the decentralized case, or on AWS), providing mutability on the application layer, while we focus on providing mutability on the storage level itself. Some solutions use trusted execution environments, while we prioritize standard cryptographic techniques. Finally, most alternatives are falling either in the local state model (mesh/p2p networks, which have the downside of necessitating maintenance on agentic machines, preventing effective delegation), or decentralized approaches with generic sharding (not allowing for faster execution due to lower, federated-model replication). Similarly to many alternatives, we also focus on common agentic interactions (data selling, competitions and collaborative tasks, data streaming), performance optimisations (latency and storage size), delivering convenient user experience to AI builders, and putting data ownership (access control model, data privacy) in the center of our design.

Ensue vs data availability layers. Data availability layers (Ethereum's danksharding, Celestia, EigenDA, and parts of Espresso Network's Tiramisu [Bea+24]) aim to provide a platform where data can be uploaded and stored for prolonged period of time, with the high guarantee of being recovered, usually under honest majority assumption. Ensue is not similar to data availability layers due to the following factors.

First, Ensue is mutable, not append-only. Theoretically these are distinct: one could emulate mutable state using immutable append-only one (as many projects do), however this incurs a prohibitive overhead; therefore we focus on a mutable model as our core assumption. Experience tells us that many real-life applications, including many agentic use-cases, do not normally require immutability and indefinite persistence, and can be replaced with the non-persistent notion of the data being "provably correctly derived", which Ensue offers instead.

Second, Ensue is federated, not decentralized in our storage approach, which significantly reduces replication factor and relies on a mixture of majority assumptions together with rational assumptions.

Finally, we do not rely on data availability sampling (DAS) for any of our cryptographic guarantees either – even though we suggest to use VID for data witnessing, the parties do not store the data for prolonged period of time, and storage providers are assumed to be the final responsibility bearing entities, not the network as a whole. We do achieve data availability under a combination of rational and honest majority assumptions – the protocol ensures that the entire current state of the state is still available and has not been truncated or otherwise corrupted – this is achieved by a combination of the read proofs and "downloading through VID" mechanics. We actively consider a possibility where multiple SPs can store the same data (low-replication federated sharding), however, we prefer to rely on reputational factors to increase the guarantee of the data being replicated (and parties not colluding), as opposed to cryptographic ones (proofs of replication). Our incentive calculations includes the possibility of collusion. This combined approach decreases overall cost for applications that do not need to use a DA layer, and allows the application developer greater control over the exact properties that they are willing to pay for.

Ensue vs authenticated storage. Ensue is not an authenticated, or verifiable storage solution (e.g. QMDB or Lagrange) because Ensue leverages blockchain and consensus properties for achieving data retrievability and availability in the optimistic manner, prioritising performance and fast responses over proof creation. In other words, our goal similarly includes providing authenticated database operations, but we use a combination of rational, honest majority, and cryptographic assumptions to achieve it as opposed to purely cryptographic ones.

Ensue vs the proofs-of-storage approach. While many solutions use cryptographic protocols to guarantee certain space usage on the storage provider's side (see e.g. proofs of replication [Fis18b; DGO19] and proofs of space [Fis18a; Rab+23]), we do not consider this problem and set of techniques for Ensue. Our aim is to focus on basic retrievability, thus we do not consider this a requirement, and prefer to rely on social / economic / reputational factors to discourage replication attacks, where e.g. a single user claims to store data in replicated way while holding only one copy.

2 Proving Retrievability and Availability

This section defines the core proving techniques used in Ensue – proofs of operations (reading and writing), and proofs of availability.

At a high level, the key property we leverage for fast writes is that polynomial commitments can be homomorphically updated when underlying data changes, allowing parties who have commitments to the data to update those commitments without having to recompute them from scratch.

2.1 Cryptographic Preliminaries

We start by introducing the notation and commonly used cryptographic primitives that we use as dependencies.

Notation. For a finite set S, we write $a \leftarrow S$ to denote that a is uniformly sampled from S. We denote the security parameter by $\lambda \in \mathbb{N}$. Given $d \in \mathbb{N}$ and a ring R, we denote by $R^{\leq d}[X]$ the set of univariate polynomials over X with coefficients in R and degree strictly smaller than d.

Our protocols will be defined over a standard, non-pairing elliptic curve group, to achieve transparency without trusted setup. Let $\mathbb{G}$ be an elliptic curve (e.g. Vesta or Pallas) and $\mathbb{F}$ its scalar field. Let $\mathbb{H}$ be a subgroup of $\mathbb{F}$ generated by a root of unity ω. We set $|\mathbb{H}| = 2^{16}$, which is the size of a polynomial representing a chunk of data in Ensue. For convenience we will be also calling $\mathbb{H}$ our "primary evaluation domain", and use the notation $\mathcal{D}$ (or $\mathcal{D}_1$) together with $N := |\mathcal{D}|$ interchangeably. We denote by $Z_{\mathcal{D}} = X^N - 1$ the vanishing polynomial on $\mathcal{D}$. Let $d \in \mathbb{F}^N$ be a vector of data a user wants to query. We denote by $\bar{d}$ its Pedersen commitment to the corresponding polynomial (obtained by interpolation on $\mathcal{D}$), on $\mathbb{G}$. Note that we might use interchangeably d for a vector and its corresponding polynomial when the context is clear. We also assume existence of bigger evaluation domains – $\mathcal{D}_2$ is defined as a subgroup of $\mathbb{G}$ such that $\mathcal{D}_1 < \mathcal{D}_2$, twice the size of $\mathcal{D}_1$. Concretely, when $\mathcal{D}_2$ is generated by ω of degree $2N$, $\mathcal{D}$ is then generated by ω^2. We might similarly assume the existence of $\mathcal{D}_i$ for bigger i with $|\mathcal{D}_i| = i \cdot N$.

Sub-protocols. Ensue relies on a number of existing cryptographic techniques and sub-protocols. Our core techniques rely on the PIOP style SNARKs, thus we will use a polynomial commitment scheme (PCS), a random oracle (modelled directly as a hash function implying Fiat-Shamir), and finally the compiled non-interactive ZK proof (NIZK). In addition, we might use folding techniques, which we will refer to later.

Algorithmically, we define the following algorithms and sub-functionalities:

- $\mathsf{IFFT}(\{D_i\}_{i=1}^n) \rightarrow F(X)$.

– Takes as input data vector of degree n and returns the corresponding polynomial such that $\forall i, F(\omega_i) = D_i$, where $\omega_1, \cdots \omega_n$ are fixed elements of a subgroup $\mathbb{H}$. Implemented with inverse fast Fourier transformation. Complexity $O(n \cdot \log n)$

- $\mathsf{PCS.Setup}(\lambda) \xrightarrow{\$} \mathsf{crs}$.
  - Generates setup parameters (common reference string) for PCS and NIZK sub-systems. Instantiated by transparent (universal) IPA bases generation using hash-to-curve.
  - We will pass crs to the necessary function implicitly to avoid notational clutter.
- $\mathsf{PCS.Com}_{\mathsf{crs}}(F(X)) \xrightarrow{\$} C$.
  - Takes as input a univariate polynomial $F(X)$ of degree most d and returns a commitment. Implemented with Pedersen (IPA) commitments. Computational complexity is $O(d)$. $|C| = 256$ bits.
  - Note that the Commit is a linearly homomorphic function: $\mathsf{Commit}(F_1(X)+\mu \cdot F_2(X)) = \mathsf{Commit}(F_1(X))+ \mu \cdot \mathsf{Commit}(F_2(X))$.
- $\mathsf{PCS.Update}(C, \mathsf{ix}, v) \to C'$.
  - Assuming C commits to D, returns a new commitment C' to the same data $D' = D$ except $D'_{\mathsf{ix}} = D_{\mathsf{ix}} + v$. Implemented using the homomorphic property of the commitment.

We will describe our proving system depending primarily on the PCS, but if necessary for future modelling, the interface for a zero-knowledge proof system is assumed to be as follows:

- $\mathsf{NIZK.Prove}_{\mathsf{crs}}(x, w) \xrightarrow{\$} \pi$.
  - Generates a proof for $(x, w) \in \mathcal{R}$, where $\mathcal{R}$ is a target language relation.
- $\mathsf{NIZK.Verify}_{\mathsf{crs}}(x, \pi) \to 0/1$.
  - Verifies π w.r.t. instance x, attesting to the existence of w such that $(x, w) \in \mathcal{R}$.

2.2 Authenticated Retrievability via Aggregated Read Proofs

The core proving layer of the Ensue protocol must support the following functionalities: storing data, issuing a receipt on storage (a commitment), updating the data at a point, and proving that a read at certain positions is equal to certain values.

The protocol is presented in Fig. 1 (setup and data management), Fig. 2 (aggregation), and Fig. 3 (batch prover and verifier).

A core part of our protocol is producing proofs of correct reads from the dataset owned by the storage provider. Since we are operating in the optimistic setting, we want these proofs to be aggregatable and lightweight. Our solution is a combination of a Plonk-style proof together with a Sangria-style aggregation (folding). That is, instead of submitting read proofs in batches, we use folding as an *interactive protocol* between SP (in the role of the prover), and the blockchain (in the role of the verifier), with both parties incrementally folding the necessary statements on the go, and SP proving their progression once in a while.

Semantics via workflow example. We illustrate the semantics of individual methods defined in figures through the following workflow example. Note that this workflow is still not the full protocol of Ensue, but rather exemplifies the core retrievability semantics.

The protocol has 3 parties: the user, the storage provider (SP) and the chain. The user wants to access the data stored by SP, in a verifiable way. The chain can alleviate the computational burden of the user and the SP due to it being a succinct chain with fast verification. The interaction might go as follows.

1. $\mathsf{Setup}(\lambda)$ is assumed to be run before any interaction starts to take place.
2. Assume the current state of the protocol is as follows:
   - State of SP: U_{acc} (aggregated instance), W_{acc} (aggregated witness), D (data), C_D (commitment to this data).
   - State of the chain: U_{acc} (aggregated instance), C_D (data commitment)

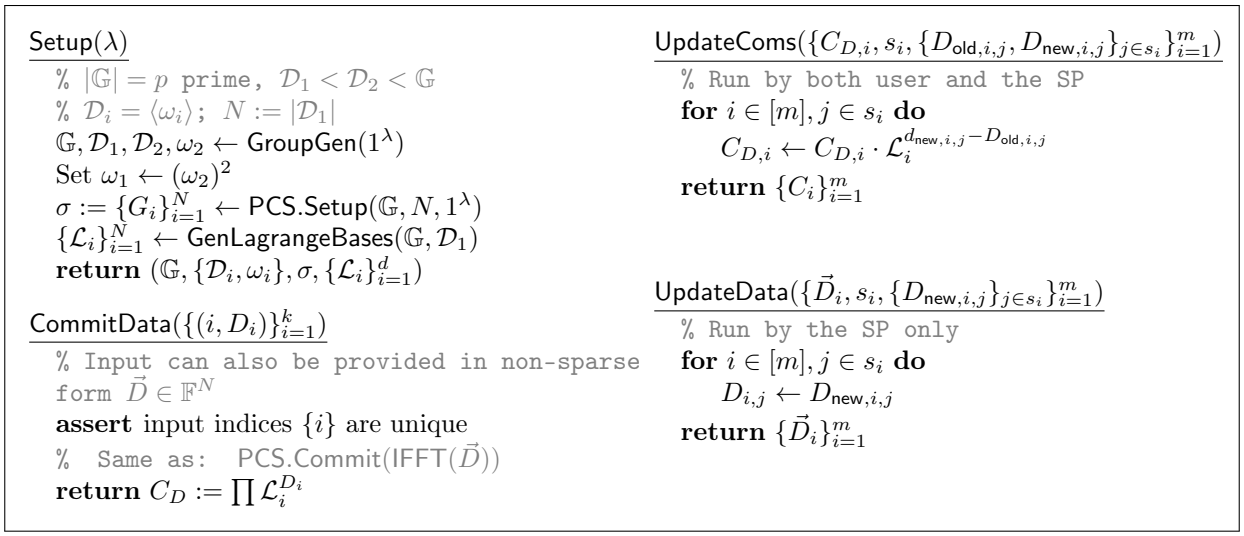

Figure 1: Ensue core commitment related proving protocols. In update routines m is the number of total commitments stored by the SP.

- State of the client: C_D (data commitment)

3. Requesting a read: user sends a selector vector s to the chain, which can be very lightweight in a sparse representation $\{\mathbf{id}_i\}_{i=1}^k$ for $k \ll N$, and $\mathbf{id}_i$ the i-th non-null value's index of the selector.
4. SP obtains s from the chain, computes $a = D \circ s$ sends a to user.
5. SP computes commitments $C_s \leftarrow \prod_{i=1}^k \mathcal{L}_i^{s_i}$ (essentially running $\mathsf{CommitData}$), $C_a \leftarrow \prod_{i=1}^k \mathcal{L}_i^{a_i}$, and performs the full folding $W'_{\mathsf{acc}}, U'_{\mathsf{acc}}, C_T \leftarrow \mathsf{FoldingFull}(\{C_D, C_s, C_a\}, U_{\mathsf{acc}}; (D, s, a), W_{\mathsf{acc}})$, obtaining new accumulated instance U_{acc}, new accumulated witness W_{acc}, and a cross term C_T.
6. SP sends C_a and cross term C_T to the chain
7. User obtains C_a from the chain and check it matches the received answers a (recreating the commitment from the succinct vector).
8. Chain computes C_s and the new committed accumulator $U'_{\mathsf{acc}} \leftarrow \mathsf{FoldingCommitted}((C_D, C_s, C_a); U_{\mathsf{acc}}, C_T)$
9. After a while SP creates a proof π for the folded instance using $\pi \leftarrow \mathsf{ProveRead}(U_{\mathsf{acc}}, W_{\mathsf{acc}})$, which the chain verifies using $\mathsf{VerifyRead}(U_{\mathsf{acc}}, \pi)$. Since the chain did the folding interactively, verifying such a batch proof implies validity of all the previous read statements by soundness of the folding argument.
10. Finally, if at some point the user wants to update the data in the commitment C_D at position i from $D_{\mathsf{old},i}$ to $D_{\mathsf{new},i}$, it sends this request to the chain, which translates it to SP. Both parties can update the commitment homomorphically using $\mathsf{UpdateComs}(C_D, \{i\}, D_{\mathsf{old},i}, D_{\mathsf{new},i})$, while storage provider also calls $\mathsf{UpdateData}$ directly on D.

2.2.1 Commitments and Data Format

We use Pedersen commitments and Lagrange bases to store our data. Fig. 1 describes the related protocols, and in this section, we elaborate on the approach.

First, recall that the IPA commitment for a polynomial $F(X)$ is formed as $\mathsf{Com}(F(X)) = \left(\prod G_i^{f_i}\right) H^r$ where $\{f_i\}$ are polynomial coefficients, $\{G_i\}, H$ are the pregenerated bases, and r is the commitment randomness (which we will omit in most cases). Now, due to us working over the subgroup $\mathcal{D}_1$, instead we will commit to polynomials that interpolate our data D_i over $\mathcal{D}_1$. That is, when $\{D_i\}_{i=1}^N$ is the data vector, we will first create a $F(X)$ such that $F(\omega^i) = D_i$ (i.e. interpolating $D(X)$ using IFFT), and then commit to the coefficient representation of $F(X)$. Naively this costs $O(n \cdot \log n)$, but there is a faster way to do it using

$\underline{\mathsf{FoldingFull}(U_1, U_2, W_1, W_2)}$

$t \leftarrow CT(W_1, W_2)$ (see Eq. (2))
$C_T \leftarrow \mathsf{CommitData}(t)$
$r \leftarrow \mathsf{H}(C_T, U_1, U_2)$
$a_{\mathsf{acc}} \leftarrow a_1 + r \cdot a_2$,
  Same for $s_{\mathsf{acc}}, D_{\mathsf{acc}}$
$e_{\mathsf{acc}} \leftarrow r^2 \cdot e_2 - r \cdot t$
Set $W_{\mathsf{acc}} := (D_{\mathsf{acc}}, s_{\mathsf{acc}}, a_{\mathsf{acc}}, e_{\mathsf{acc}})$
$C_{a_{\mathsf{acc}}} \leftarrow C_{a_1} + r \cdot C_{a_2}$
  Same for $C_{s_{\mathsf{acc}}}, C_{D_{\mathsf{acc}}}, C_{e_{\mathsf{acc}}}$
$u_{\mathsf{acc}} \leftarrow 1 + r \cdot u_2$
$C_{e_{\mathsf{acc}}} \leftarrow r \cdot C_T + r^2 \cdot C_{e_2}$
Set $U_{\mathsf{acc}} := (C_{a_{\mathsf{acc}}}, C_{s_{\mathsf{acc}}}, C_{D_{\mathsf{acc}}}, C_{e_{\mathsf{acc}}}, u_{\mathsf{acc}})$
return $U_{\mathsf{acc}}, W_{\mathsf{acc}}, C_T$

$\underline{\mathsf{FoldingCommitted}(U_1, U_2, C_T)}$

$r \leftarrow \mathsf{H}(C_T, U_1, U_2)$
$C_{a_{\mathsf{acc}}} \leftarrow C_{A_1} + r \cdot C_{A_2}$
  Same for $C_{s_{\mathsf{acc}}}, C_{D_{\mathsf{acc}}}, C_{e_{\mathsf{acc}}}$
$u_{\mathsf{acc}} \leftarrow 1 + r \cdot u_2$
$C_{e_{\mathsf{acc}}} \leftarrow r \cdot C_T + r^2 \cdot C_{e_2}$
Set $U_{\mathsf{acc}} := (C_{a_{\mathsf{acc}}}, C_{s_{\mathsf{acc}}}, C_{D_{\mathsf{acc}}}, C_{e_{\mathsf{acc}}}, u_{\mathsf{acc}})$
return U_{acc}

Figure 2: Aggregation algorithms for the relaxed read proof language.

$\underline{\mathsf{ProveRead}(U, W)}$

Parse $(C_D, C_s, C_a, C_e, u) \leftarrow U$
Parse $(\vec{D}, \vec{s}, \vec{a}, \vec{e}) \leftarrow W$
% $C_v = \mathsf{Com}(\mathsf{IFFT}_{\mathcal{D}}(v))$ for $v \in D, s, a, e$
$D(X) \leftarrow \mathsf{IFFT}_{\mathcal{D}_2}(\vec{D})$ % Interpolate
  Similarly obtain $S(X), A(X), E(X)$
$Q(X) \leftarrow \big(D(X) \cdot S(X) - u \cdot A(X) - E(X)\big) / Z_{\mathbb{H}}(X)$
$C_Q \leftarrow \mathsf{Com}(Q(X))$
Let $\mathsf{coms} := \{C_D, C_s, C_a, C_e, C_Q\}$
$\zeta \leftarrow \mathsf{H}(C_D, C_s, C_a, C_e, C_Q, u)$
$\mathcal{E}_D \leftarrow D(\zeta)$ % Evaluate polynomials
  Similarly obtain $\mathcal{E}_s, \mathcal{E}_a, \mathcal{E}_e, \mathcal{E}_Q$
Let $\mathsf{evals} = \{\mathcal{E}_D, \mathcal{E}_s, \mathcal{E}_a, \mathcal{E}_e, \mathcal{E}_Q\}$
$\pi_{\mathsf{PCS}} \xleftarrow{\$} \mathsf{PCS.BatchProve}(\mathsf{coms}, \mathsf{evals}, \{D(X), \ldots\}, \zeta)$
return $(\vec{a}, \pi := (\{C_Q\}, \mathsf{evals}, \pi_{\mathsf{PCS}}))$

$\underline{\mathsf{VerifyRead}_{\mathsf{pk}}(U, \pi)}$

Parse $(C_D, C_s, C_a, C_e, u) \leftarrow U$
Parse $(\{C_Q\}, \mathsf{evals}, \pi_{\mathsf{PCS}}) \leftarrow \pi$
$\zeta \leftarrow \mathsf{H}(C_D, C_s, C_a, C_e, C_Q, u)$
assert $\mathcal{E}_Q Z_{\mathbb{H}}(\zeta) = (\mathcal{E}_D \cdot \mathcal{E}_s - u \cdot \mathcal{E}_a - \mathcal{E}_e)$
assert $\mathsf{PCS.Verify}(\pi, \mathsf{coms}, \mathsf{evals}, \zeta) = 1$
return true

Figure 3: Core Plonk-ish proof protocol for the relaxed read language. Intended to be used on aggregated relaxed instances.

precomputed Lagrange bases.

Given a finite field $\mathbb{F}$, the k-th Lagrange basis polynomial $\mathcal{L}_i^{\mathbb{H}}$ over a multiplicative subgroup $\mathbb{H} \subseteq \mathbb{F}$ of order n is defined as the unique polynomial of degree $n-1$ such that

$$\mathcal{L}_i^{\mathbb{H}}(\omega_k) = \begin{cases} 1 & \omega_k = \omega_i \\ 0 & \omega_k \neq \omega_i \end{cases}$$

They can be used to interpolate polynomials given in evaluation form: given $\{D_i\}_{i=1}^n$, we can obtain $\mathsf{Com}(F(X))$ by simply computing the multi-scalar multiplication $p(x) = \prod_{i=1}^n \mathcal{L}_i^{\mathbb{H}}(x)^{D_i}$ in linear time. We will pre-generate commitments to these Lagrange bases during the setup time. Moreover, this technique can be used to homomorphically change the values of the commitment: if C is a commitment to $\{D_i\}$, then $C \cdot \mathcal{L}_i^v$ is a commitment to the same dataset except $D_i' = D_i + v$.

2.2.2 Warmup: Proving a Single Read

For ProveRead we could have used a standard generic gate plonk-ish style argument with the following parameters. The data vector is D (committed as C) and the selector vector is s_i, both of degree d that is a power of two, same as size of the subgroup $\mathbb{H}$. We compute $a = s \circ D$ to obtain the value vectors. Note that both selector vectors and value vectors can be quite sparse – we might be interested in proving a read of just a few positions from D. Let β be a uniformly sampled random field element. The equation we want to prove for reads is

$$D(X) \cdot S(X) - A(X) \equiv 0$$

Importantly, when verifying this proof (e.g. within a smart contract), the verifier can build commitments to a and s (or their evaluations at a point) easily in time linear in the number of elements within them.

The user sends a query $s \in \{0,1\}^{2^{16}}$ to the chain[2], where a bit set indicates that the user wants to access $D[i]$[3]. The SP obtains s, and answers with the answer denoted $a \in \mathbb{F}^{2^{16}}$ computed as $s \circ D$ where $\circ$ is the pointwise product (i.e., a is the data at the coordinates where it was queried, zero otherwise).

The SP also sends $\bar{a}$, the commitment to a to the chain, together with a proof. The verification is two fold:

- The chain verifies that $\bar{a}$ is computed correctly (see Chain/SP protocol)
- The user verifies that the a is coherent with $\bar{a}$

Between SP and the user the interaction is as follows: the user simply re-computes the commitment to the answer a, and compares to $\bar{a}$ which was sent to the chain, and verified by it. If the user verification fails, the answers need to be sent on chain to determine who was wrong.

Finally, the chain and the SP engage in a polynomial protocol (see the Plonk paper section 4.1), also sometimes called a plonkish proof. This allows to prove a low degree identity over all coordinates of a vector. The identity we prove here is $a = s \circ D$, proving a is well formed. On the prover side this is done by committing the elements, absorbing them into the sponge, computing the polynomial $D(X)S(X) - A(X)$ by interpolating the vectors over the root of unity domain, then dividing it by the zero polynomial $X^N - 1$ thus obtaining $Q(X)$, sampling evaluation point ζ, evaluating all the polynomials on ζ, and finally creating an IPA batch proof for these evaluations. The proof then consists of the commitments, evaluations, and the IPA batch proof. The verifier would absorb the commitments, squeeze out the same ζ, and then perform two checks: that $Q(\zeta)(\zeta^N - 1) = D(\zeta)S(\zeta) - A(\zeta)$ (using the evaluations sent, and that the IPA batch opening is correct. For more details, refer to the original Plonk paper.

As a side note, we could create a batch prove with the following equation: $\prod_{i=1}^k D(X) \prod \beta^i S_i(X) - \prod_{i=1}^k \beta^i A_i(X) \equiv 0$, given a random β. This would allow us to perform a multi-column read; for practical reasons we focus on a single-column read with folding instead.

[2]The implementation can choose a more efficient formatting, assuming the query is sparse.

[3]Note that in general we never want to send anything of size 2^{16} to the chain.

2.2.3 Proving Aggregated Reads

An instance of the previous protocol is $(a, s, D) \in \mathbb{F}^{N\times 3}$, and a committed instance is $(\bar{a}, \bar{s}, \bar{D}) \in \mathbb{G}^3$. An instance is satisfied iff $a = s \circ D$ on every coordinate. Similarly a committed instance is satisfied if there exist pre-images corresponding to a satisfied instance. A relaxed instance is $(u, a, s, D, e) \in \mathbb{F} \times \mathbb{F}^{N\times 4}$ which is satisfied iff

$$D \circ s - u \cdot a - e = 0 \tag{1}$$

on every coordinate, where e is the error term and u the homogenization term. Note that a satisfying instance can be transformed into a satisfying relaxed instance setting $u = 1$ and $e = 0$. We describe two folding algorithms, one working with full instances, one working on committed instances. The one working on committed instances needs an auxiliary input called cross term. FoldingFull (resp. FoldingCommitted) takes two (resp. committed) instances as input, one relaxed and one normal, and output one relaxed (resp. committed) instance which is satisfied iff the two inputs were satisfied. This is inspired by Sangria, adapted to our equation, and leveraging the fact that we fold one relaxed and one non relaxed instance.

We define the cross-term multivariate polynomial for instance $W_1 = (a_1, s_1, D_1)$ and relaxed instance $W_2 = (a_2, s_2, D_2, u_2, e_2)$

$$CT(W_1, W_2) = a_2 + u_2 \cdot a_1 - s_2 \cdot D_1 - s_1 \cdot D_2 \tag{2}$$

Note that in full folding we can avoid hashing the accumulated instance, by keeping in memory the state of poseidon. At iteration i we start with the state of iteration $i - 1$ containing acc_{i-1} and W_{i-1}, which determines acc_i.

Folding read proofs on Ensue. Both the chain and SP perform folding iteratively over time simultaneously. Therefore there are two stages of the protocol – accumulation (that is running all the time), and opening the accumulated instance (that happens once in a while when the SP is ready to do so).

Accumulating instances. In the first part we presented a protocol between the chain and SP which is costly, as every proof incurs an IPA proof and verification. We want to avoid that using folding, to open several proof at once. We also want the SP memory to be independent on the number of proof we open at once. The SP keeps in its state a relaxed instance, folding all preceding proofs since the last opening[4]. This state is called the accumulated instance. Similarly the chain keep in its state the corresponding committed relaxed instance. Note that SP also stores the committed accumulated instances, which is necessary to derive the Fiat-Shamir challenges. When a new read request arrives, SP executes FoldingFull on it together with the accumulated instance, to update its instance. He sends the cross term to the chain, together with $\bar{a}$ the commitment to the answers. The chain can then execute the FoldingCommitted and update its accumulated committed instance. Note that the user is not affected by the folding: he grabs $\bar{a}$, and check it against the received answers, as before. Note that the chain needs to keep all the $\bar{a}$ until we are sure they are correct.

Opening an accumulated instance. This is very similar to the proof of the base protocol, except our target equation is now the relaxed version (Eq. (1)). The SP and the chain engage in a polynomial protocol to prove that the accumulated instance is satisfied. This gives the prover and verifier as presented in Fig. 3.

2.3 Ensuring Availability via VID

Attaching big data blobs to transactions is universally expensive, and constitutes one of the biggest bottlenecks in current Web3 performance. Solutions like DA layers and rollups (often used together) are one way to solve the problem, but their performance is still largely limited by high replication factors. In this section we explain how using techniques from DA we can improve performance of our federated model without paying the full cost of DAs.

[4]It is unclear to me how we want to decide that we stop accumulating to produce a proof. It has implications in how long can SP lie before getting caught. Therefore this probably should be decided per contract.

To enforce interactions over large data while not overloading the chain, Ensue needs to use a mechanism that allows parties to "attest" to the fact the data has been sent (and make it available for a short time), but to not include it on-chain. Note that if SP is honest, it will prioritise off-chain interactions with the user, since those are generally cheaper, and our incentives will be adjusted to keep this option cheaper. However, when the SP is malicious and wants to purposefully disrupt user's interaction with the state this SP is responsible for, SP might not do any off-chain interactions. The issue here is non-repudiation (see Appendix A) – without a third party (a trusted witness or a chain), it is not possible to decide whether user did not act with SP because of the user (wanting to slash the SP) or because of the SP (wanting to disrupt the user). Therefore, for the malicious case we have to introduce such a party.

This description fits the cryptographic primitive called Verifiable Information Dispersal (or VID). Many DA systems use VID to send the data over to the parties, but they are also often designed with the DAS (DA Sampling) protocol in mind, which we do not need. Our core requirement is the ability to use the chain as a "decentralized proxy witness" for channelling the data between the user and SP (or between SP and SP in the case of data transfer), who do not want to collaborate off-chain for some reason.

There are three scenarios in which VID is improving data availability and service reliability of our system.

1. When user and SP do not interact offline (because of the user or SP), and the data reads or data update operations operate over big chunks of data, that would be prohibitively expensive to put on chain.
2. When user wishes to terminate the agreement with the SP and download their data back – the amount of data is, again, assumed to be much bigger than an average read query, and by default SP can withhold this data effectively forever.
3. When user wants to transfer the data from one SP to another. This can be seen as a sub-problem of the previous scenario – if downloading is not a problem, the user can download and reupload. However, with a VID protocol, SPs can now accountably transfer the data between each other without user's involvement.

The gist of how VID solves these issues is quite simple: the party that has to upload the data (e.g. user in the first scenario, or SP in the second and third) tries to do it off-chain first, and if this does not succeed, it executes VID dispersal procedure to the set of pre-chosen nodes. We imagine VID nodes to be the same, or to be a subset, of our consensus nodes, to be able to reuse our 2/3 BFT honesty assumption. Then, after dispersing the data and obtaining signatures of witnessing this data, the party will post this signature on-chain. This transaction is lightweight, since it is only a signature on the data digest. The other party will now immediately reconstruct the data from the VID committee – this is possible under a honest majority assumption together with rational assumptions (VID participants have to be rewarded for being honest). This essentially solves the non-repudiation problem. Note that unlike in most DAs, there is no DAS (sampling), and the nodes only store the data for a short period of time, thus decreasing the storage requirements of VID committee parties significantly. The bottleneck resource here is bandwidth, not storage.

Many VID protocols fit our general requirements, but our specific setting (absence of pairings, necessity to have SNARK-friendly signature verification, etc) dictate a need to have a custom solution. The rest of the section expands on a potential choice of VID for our requirements.

2.3.1 Coset-Opening IPA-based VID

One naive implementation of a VID, roughly following Espresso's design, would look as follows.

Assume that there are n nodes and N is the height of the commitment with $n < N$ (think $N = 2^{15}$). Assume we are dealing with k commitments at the time. A naive way to perform the VID would be to extend N-sized commitments to m-sized ones, where $N = rm$ (r is the code rate, assume e.g. $r = 1/2$), and give every party a *set* of evaluations of size m/n. A message for the VID party i would include all evaluations on $\{\omega^{i+n\cdot j}\}_{j=0}^{m/n-1}$. One derives α by hashing all the commitments, computes a combined commitment (and polynomial) $\hat{C}$ (corresponding to $\hat{p}(X)$), and opens this $\hat{C}$ on the points. Then one computes a quotient polynomial by dividing $\hat{p}(X) - \prod_{\mathcal{I}} \hat{e}_i L_i(X)$ ($\hat{e}_i$ is an α-combined evaluation at row i, $L_i(X)$ is a Lagrange polynomial, $\mathcal{I}$ is an index of the coset) by $(X^m - \omega^{im/n})$ (this equation contains the whole coset). The

division can be implemented fast, similarly to how division by $X^N - 1$ is fast in Plonk. Finally, one derives an evaluation point ζ and uses an IPA proof to open $p(X)$ and quotient evaluation at ζ. To verify the proof, one must compute $p(\zeta) - \prod L_i(\zeta)\hat{e}_i$ and attest that this is equal to $\zeta^m - \omega^{im/n}$ times $q(\zeta)$ (quotient eval). Computation of Lagrange bases on ζ can rely on a single-time precomputations of denominators, and can utilise the fact that products share sub-terms.

Our practical evaluations show that verifying such a proof on the VID side takes about 80 milliseconds (dominated by verifying the IPA proof), and creating a proof for a single VID replica takes 700ms (dominated by creating the IPA proof). The proving time might be potentially optimised even more. Note that both proving and verifying do not scale, in practice, with k (number of commitments) because operations are performed on the combined commitment, and recombination is cheap compared to IPA proof creation.

Finally, one can reuse IPA proofs to improve VID efficiency. Instead of giving a single IPA proof to a party to verify w.r.t. their unique coset, one can "share" an IPA proof between several parties. This requires sending every party i in the group the combined evaluations $\hat{e}_i$ for their neighbours. In this scenario the VID party i actually combines $\hat{e}_i$ from $e_{i,j}$ that it received earlier on its own, but assumes the veracity of other $\hat{e}_i$ belonging to the other VID parties. To make sure that the user does not send different batch opening values to different parties within the same cluster, we can require the disseminator to publish $\sum \hat{e}_i \beta^i$ for some hash-derived β, and ask every VID participants to sign this accumulated value. This seems like a reasonable trade-off for reducing prover costs assuming $k \gg N$, but it can incur extra verification costs in case it might be necessary to cross-check the combined evaluations between VID verifiers. To be investigated.

2.4 Security Properties

The following is a list of properties that one can imagine our system to have, with dependencies presented on Fig. 4.

1. Responsiveness: commonly issued updates and reads are fast.
   - This corresponds to "99.99% uptime" intuition – some rare queries might be delayed, but on average all queries that are asked by users are answered.
2. Existence: that the data is present *at all*.
3. Complete Responsiveness: *any* data queried (read / write) will be processed fast.
   - The opposite is that rarely-used data can be "read" slower.
4. Data Availability: all data can be retrievied if necessary, albeit inefficiently. Think of how DAS solutions work.
5. Retrievability: that the data can be downloaded fast even from non-cooperating SP. Includes post-contract retrievability.

The relation between them is as follows. Clearly, complete responsiveness (3) implies responsiveness (2), and complete responsiveness (3) implies existence (1). Retrievability (5) implies data availability (4) – if one can download data efficiently at any point it's available; but data availability (4) does not imply retrievability (5) since data might be available, but downloading it might be an inefficient process. Finally, retrievability (5) implies existence (2); and data availability (4) implies existence (2) with high probability (assuming most parties do not drop the data hoping to reply DAS queries still).

Our system satisfies responsiveness and retrievability, excluding complete responsiveness, under a combination of cryptographic and rational assumptions. The presence of batch read proofs imply responsiveness, since not being responsive is leading to SP being slashed. The IVC serves a dual role: to prevent non-repudiation issue, but to also allow provable downloads of all the data. The latter implies retrievability: our protocol will enforce a party to disseminate all the data to the VID, and inability to do so leads to slashing.

Achieving privacy. Note that we do not focus on zero-knowledge in this construction, due to the fact that we assume all the data on the blockchain to be publicly available. This means that even though we are using ZK-compatible proof systems, in practice we will not care about their ZK aspects; similarly we do not blind our commitments unless necessary for soundness or correctness; etc.

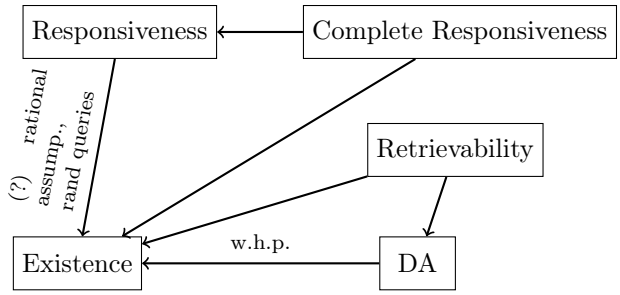

Figure 4: Relationship between storage-related security properties.

Nonetheless, data requested to be stored can be encrypted (so that the server provider will not learn the raw data), and our commitments can be adapted to work with blinders. This means we could provide storage privacy, by always referring to the commitments of updates (C_S, C_A) and commitments to data instead of the referring to the data directly.

3 Main Protocol

3.1 Protocol Key Terms

We define the following terms as part of the vocabulary necessary to describe the main protocol. The protocol relies on several key components for its operation. *Accounts* are typed ledger entries containing a *balance*, which represents the amount of native tokens available. *Storage Providers* (SPs) are nodes responsible for storing network data, who must provide *collateral* as a guarantee for safely handling this data. Data is organized into *chunks*, the smallest unit configured by the protocol, while a *commitment* serves as a constant-size representation binding to data pieces for cryptographic proof verification. The *exchange rate*[5] defines the price per chunk for operations, expressed as nano-native tokens. *Storage contracts* hold collateral submitted by providers, with each contract scoped to a single commitment and involving both a *contract requester* who initiates creation and a *contract acceptor* who provides the necessary collateral. These parties agree upon *terms* that specify exchange rates, deadlines, and other metadata. The protocol supports three main operations: *write* for storing and allocating data, *read* for querying stored chunks, and *update* for editing already allocated data. Finally, an *epoch* represents a fixed number of blocks that unlock all locked payments and collateral.

3.2 Network Overview

Node operators in the Ensue ecosystem will submit transactions, verify proofs, validate and produce blocks, and gossip to peers. The purpose of the network is to store and serve data requested by applications in a reliable way through a succinctly verifiable blockchain.

Similarly to Mina, the chain's succinctness is powered by snark workers that produce SNARKs for transactions in parallel. Additionally, validators collectively produce and verify blocks with a BFT-based proof of stake algorithm. This guarantees fast finality, non-malicious quorum within the network, and a loose oversight of deadlines that need to be maintained for request responses.

Deadlines are needed because participants and storage providers communicate asynchronously. Every request transaction is expected to be answered with a corresponding proof transaction within a configurable block window. By keeping the interactions minimal, this request-answer mechanism coupled with deadlines allows the network to perform and validate operations with minimal bandwidth. Any violations to the deadlines established by the protocol will cause a slash on the violator's collateral.

[5]Exchange rates on chain will be represented by the nano-version of the native token per chunk i.e. $10^9 \times \frac{\text{native token}}{\text{chunk}}$

Type	Functions	Rewards and Incentives
Storage Providers	• Storing data • Handling update and write requests • Answering read requests • Monitoring the network	• Participant fees for each request • API credit
Validators	• Consensus participation: block production and block validation • Purchasing SNARK work • Including internal transactions and acknowledgments in blocks	• Block rewards • Acknowledgment rewards • Slashing rewards • Transaction fees
VID Nodes	• Participating in the VID protocol, facilitating on-chain read and update queries. • May coincide with consensus validators	• Direct rewards for data dissemination
SNARK Workers	• Producing and selling SNARK work for transactions included in blocks	• Satisfied SNARK bids
Base Nodes	• Syncing to the network and submitting local transactions to the mempool. Both storage providers and validators inherit all functionality from base nodes.	
Seed Nodes	• Serving a reliable list of peer addresses so new nodes joining the network can discover peers	

Table 1: Node types, roles and revenue streams

Each node type will have a different computational overhead and business model. See Table 1 for more details.

3.3 Ledger Construction

We logically separate our chain into "sub-ledgers", containing differents parts of the protocol state. The construction will inherit Mina's design: the staged, snarked, and staking ledger are the same as in Mina. The only important exception is the new *read ledger*:

- *Read ledger*: requests that need to be acknowledged or proven.
- *Staged ledger*: accounts for transactions that have been sequenced but not yet snarked.
- *Snarked ledger*: accounts for transactions that have been included in the staged ledger and snarked.
- *Staking ledger*: account staking used to represent the stake repartition.

Recall that in Mina, each account is the leaf of a Merkle tree. The Merkle root hash is included in blocks to maintain a succinct representation of the current chain state. State will continue to be maintained separately for snarked and unsnarked transactions, meaning that there will be both a staged and snarked ledger. Furthermore, a staking ledger will maintain the staked balances for each account. Therefore, the account with the highest stake balance will always have the highest probability of producing blocks.

The read ledger will be used to track the state of asynchronous interactions by maintaining a sliding window of requests that are yet to be acknowledged. As requests are acknowledged, they will be removed from the ledger. Each time a new block is proposed, a segment of un-acknowledged requests outside the window will be popped and a new root hash will be computed. The new root will then be verified by the network and internal slashing transactions — that punishes the storage providers responsible for processing the request — will be sequenced in the block. Note that not all the requests have to go through the read ledger, and it is possible to perform some of them off-chain.

Type	Fields	Notes
Regular Account	• Public Key • Balance • Storage Provider Terms • Locked Balance	• A regular account is a type for both requesters and storage providers. • The storage provider terms are only set for storage providers. • The locked balance is the amount of native token committed to requests. They are transferred to storage providers during a final agreement. This balance cannot be withdrawn or used until the end of an epoch.
Storage Contract	• Public Key • Commitment • Collateral Balance • Read Escrow Balance • Agreed Terms • Contract Requester • Contract Acceptor • Permission Bits	• The collateral balance will be the collateral submitted by the storage provider in the agreed-upon terms. Each one will be scoped to a single data commitment that resulted from a write. • The read escrow balance will correspond to the locked payments for read. When the underlying read request is fulfilled within the deadline, the corresponding payment is delivered to the storage provider; if not, the payment is returned to the requester. • Permission bits control the readers for the contract.

Table 2: Description of accounts in the staged and snarked ledger i.e. Merkle leaf structure

3.4 Account Types

The ledger supports two types of accounts: regular ones (implementing the basic funds transfer / locking functionality), and storage contracts (a hardcoded type of smart contract that controls all the storage logic including access mechanics). The structure of the account types is described in Table 2.

Both storage providers and contract requesters will be represented as regular accounts in the ledger. However, storage provider terms will only be set for storage providers. This field serves as a bit to indicate when an account belongs to a storage provider and is a requirement for accounts submitting contract agreement transactions. It includes values that are "likely" to be accepted. There will be no protocol enforcement of these terms. The only purpose is for requesters to get quick information on ideal terms with no negotiation and no understanding of market rate.

Negotiation in Ensue is built around terms, which are provided dynamically for each write request. If a contract is accepted, the creator will have a lease for a fixed size, contiguous blob of memory allocated by the storage provider. This blob of memory binds to a single commitment and is stored in the contract. An update request can be made to update the commitment in the contract. Reads and Updates to the commitment must be priced according to the terms agreed upon. During contract creation, the requester will propose terms with the following attributes:

- *Expiration.* Block height until which the contract is valid. Also includes the "wrap-up period" duration during which the storage provider has to allow user to download the data, or to transfer it to another storage provider.
- *Data size.* Size of the data that needs to be stored in unit of chunks.
- *Exchange rates for reads and updates.* The monetary terms defining on-chain and off-chain costs of operations and how they relate to each other.
- *Termination compensation.* The compensation due to the contract requester if the acceptor wants to end service early.
- *Initial write payment.* The amount of native token that will be locked up and paid after a final agreement.

There should be a precondition for transactions with terms: that the termination compensation cannot be higher than the demanded collateral amount. This will ensure each termination is intended when a storage provider can no longer host data. A storage provider should not find termination equally or less profitable than getting slashed.

The storage contract account type will be added to the ledger once a storage provider agrees on the terms proposed by the requester. Once a final agreement is submitted by the acceptor, the storage contract

account will hold the collateral of the SP. Two deadline fields will be used to track both the termination block of the contract, and the block at which the next read proof should be submitted by. The requester and acceptor will also be included in the account for easy reference.

3.5 Actions and Transactions

This section will explain the nature of each asynchronous interaction for each request type, along with details of internal transactions. Additional context will be provided for how the protocol manages side channels for data transfer. Side channels allow for data that is initially written during contract creation to not be included in blocks, saving a significant portion of memory in each block. The current version of the protocol design supports writes, reads, and updates of arbitrary size, but is optimised for smaller data (KBs / MBs), low latency, and higher transaction throughput – the properties we believe are most crucial for fast agentic interactions.

Off-chain interactions. Before each step of the process is described, it is important to be clear on the notion of off-chain interactions (also called side channels) in Ensue. All the presented workflows have off-chain and on-chain variants, where the former relies on an option to communicate directly, avoiding more heavy interaction with the chain. Logistically, this requires the node (storage provider, core node, or VID node) to have an externally facing service that accepts data. We do *not* expect regular users to be acting like servers in the network (that is, to have a globally accessible open port). Our workflows are designed in such a way that the off-chain flow is preferred if parties are collaborative – it incurs significantly less cost in bandwidth and computation, and the incentives are designed to prioritise it. Furthermore, the off-chain flows are designed in such a way that the off-chain interaction precedes any future on-chain interaction, and the artifacts created during the off-chain phase are non-deniably proving that the interaction happened, achieving fairness. However, due to non-repudiation problem, if parties do not want to interact off-chain (and it is theoretically impossible to establish whose fault it is), any party can force an on-chain interaction, which is more costly but is still fairly fast. In this case incentives are set up so that a rational party will always prefer the off-chain flow, since on-chain flow is more costly for everyone in a "fair" way.

In terms of engineering choices, we are envisioning a semi-anonymous p2p discovery layers that participants can rely on to establish direct connection to each other while avoiding eclipsing / DOS attacks. There will also be a standard specification for message format sent over such side channels, which is outside of the scope of this document.

Batching read and write requests. We operate in the optimistic setting where we assume that SP *promises* a correct read or update to the user (via means of a transaction or a signature), and, afterwards, SP has to post the "sequencing" transaction itself together with the cross term. The user can challenge the SP if their sequence is different from the sequence requested by the user (and attested by SP's signature), and SP is slashed if this has been detected. For sequences that already happened or were posted on chain, the user does not have to do anything – the blockchain validity includes the validity of read proofs, thus a valid chain implies an existence of a read proof attesting to the previously promised action. Using this approach we can improve overall system performance by sending less transactions, in batches, and sent by the storage provider, who is assumed to have significantly higher uptime than the user. Additionally, the optimistic approach allows us to utilise interactive folding for even better aggregated efficiency, as explained in Section 2.2.3.

3.5.1 Contract Creation Action

The goal of the contract creation action is to initiate a new storage contract from scratch[6] – this includes mainly negotiating the terms of the contract, choosing a storage provider, and populating the storage with the initial data. We describe the action in Fig. 5.

[6]In the future, we will also add the functionality of creating a contract as a "continuation" of another expired contract, for smooth storage ownership transfer.

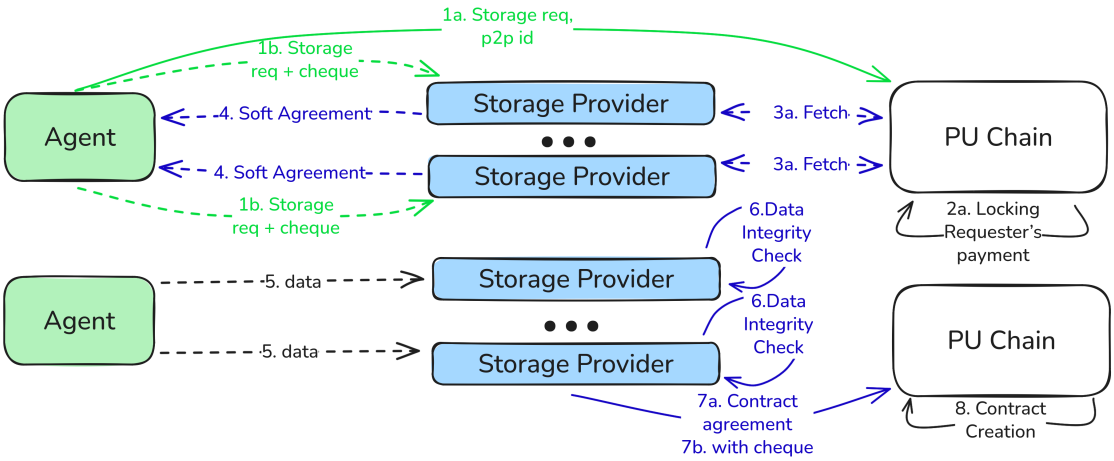

Figure 5: Storage contract creation and initial allocation write flow. It features two alternative flows: "a" flow is on-chain SP discovery, "b" flow is simpler and assumes a p2p SP discovery mechanism.

The workflow has two variants: on-chain ("a" on the diagram) and off-chain ("b" on the diagram). We describe the on-chain variant first. A request is sent by an application to establish a contract. A storage provider allocates data for this request. The required components are terms, a fee for the initial write, and the commitment for the data. Since the party that will accept the request is unknown, the fee for the initial write will be temporarily locked in the requester's locked balance. Once this transaction is included in a block, storage providers may gossip a soft agreement that declares an intent to handle the data off-chain. This allows for multiple storage providers to acknowledge the request. When the requester sees a soft agreement, they may broadcast the data that they intend to write to the storage provider over the side channel. The size of the data must be less than or equal to the number of chunks agreed upon in the terms, and also match the commitment in the initial request. If these properties hold, the storage provider will store and submit a final agreement transaction that creates the storage contract, and transfers the collateral from the storage provider balance into the contract balance. The locked balance will also be unlocked and transferred to the contract acceptor. The first acceptor to submit a final agreement with a sufficient collateral balance will be the storage provider responsible for the contract.

The main difference of the off-chain variant of the flow ("b" on the diagram) is that it requires one less transaction from the user in case the user knows which SPs it wants to negotiate with in advance. Instead of on-chain discovery and locking the funds, the user may send the storage request off-chain to the chosen providers together with a one-time signature on the current timestamp, contract terms, and a "cheque" – the message allowing SP to transfer a pre-agreed amount of funds to the storage contract. In the end of the flow, SP submits this original user's permission to the chain, thus effectively creating the contract instead of the user. The assumption of the off-chain flow is realistic in case the user interacted with certain storage providers before, or in case the user relies on a third-party explorer that allows SP discovery[7].

Design rationale and security (sketch). The main intuition behind the security of this action is fairness and inability to replay messages.

We elaborate on the distinct scenarios and properties first in the "a" workflow:

1. Completeness: if both the user and the storage provider are honest, then the contract will be created (by the SP who is fastest to respond).
2. No messages can be replayed. The storage request is a transaction, so it cannot be replayed by the

[7]We tentatively assume the existence of a storage provider explorer, potentially with SP reputation support, since it is a great UX tool that we think would benefit the protocol in many ways.

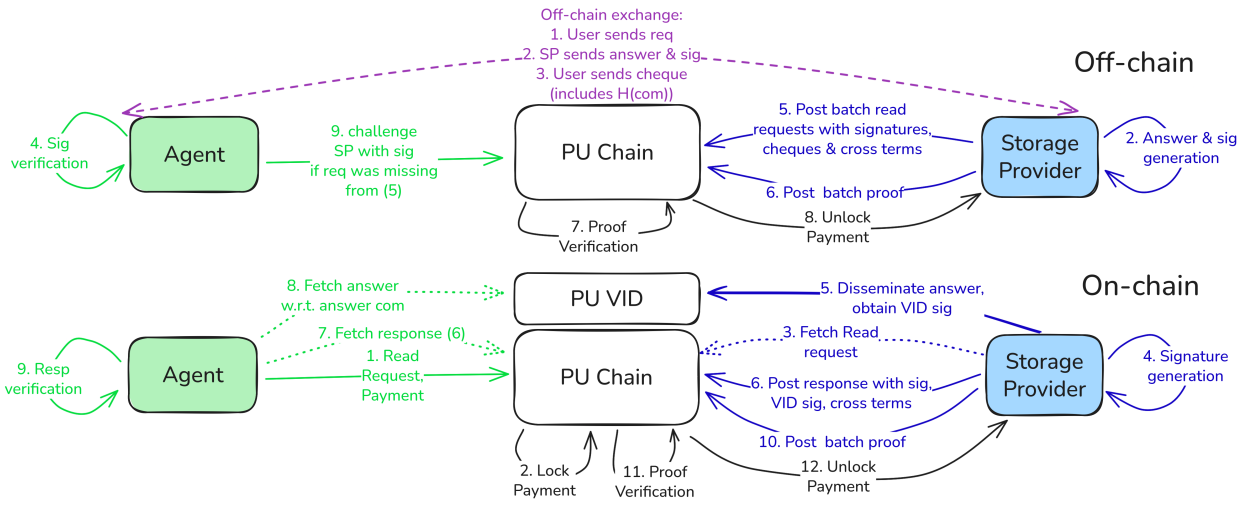

Figure 6: Read request flow.

blockchain design. The smart contract will only accept one SP per contract, and the SPs contract agreements are contract specific, so these cannot be reused across queries.

3. User only pays once, and only if the smart contract is initiated. This is trivially guaranteed by the smart contract structure itself

In the "b" workflow the completeness works the same, and other properties are:

1. The user only pays once, and only in case the contract is created. This is less trivial than in the "a" workflow due to the fact that user issues multiple cheques. However, the property is achieved because the cheques can be only redeemed when the contract is created, and because the contract can only be created once and with a single cheque.
2. Storage provider receives the promised payment if the user has enough funds to initiate it. Again, this is less trivial in "b" due to the fact that the cheque does not lock the payment. This property holds since the contract will not be created if the cheque cannot be redeemed due to user's insufficient balance. The caveat is that in this case the user forced data computation on SPs side without necessarily paying for it. However, given that the SP can check the cheque validity on-chain (if user has enough funds), the user would have to move the funds out between sending the data and contract redeeming, which incurs a transaction cost perhaps comparable to the cost of data uploading. A rational user will not do this, and a malicious user wanting to DOS the storage provider has to spend non-negligible funds to do so, making the attack costly.

3.5.2 Read Action

The goal of the read action is for the user to query stored data at a given set of indices. We describe the workflow in Fig. 6 in two variants – collaborative off-chain and default on-chain. Read action validity relies on the read proofs described in Section 2.2.

We first describe the on-chain flow. A read can be made to a storage contract once it has been created via final agreement. Anyone with the right access can send a read transaction, which requests a limited number of chunks for a specific commitment (formally described by a boolean selector vector per commitment). Since a storage contract already exists, the read request payment can be stored in its account. If a read request is not agreed within the read deadline, the request payment will be refunded and the stored collateral will be slashed. The storage provider then fetches the read request, generates an answer with signature, and disseminates the answer to the VID. After SP obtains the VID signature it posts the response on chain – the

chain only sees the commitments, not the data itself. Finally, the user queries the data from the chain and VID, which concludes the first phase of the protocol. The VID part can be skipped if the data queried is public and small, in which case it is more efficient to put it on chain; one of the VID's main point is to make big reads economically feasible. In the second phase of the protocol, after some time passes, the storage provider posts batch proof for the aggregation of read requests – after it is accepted by the chain the SP unlocks all the payments for the reads.

The collaborative off-chain flow is more lightweight in terms of transactions, and it does not use VID since the data is sent directly from SP to the user. The second phase of the flow is the same as in the on-chain variant. The first phase differs in that the user and SP engage in an off-chain interaction where user sends the request, SP sends the answer data and signature, user verifies this signature (on the data commitment that it reconstructs), and acknowledges it with a cheque. This cheque can be later used by the storage provider to claim the payment.

If the off-chain flow is not initiated due to one of the two parties being inactive or sending incorrect data, any party can force the interaction on-chain: user can post the on-chain request, and SP can force it by ignoring off-chain requests.

There also exists a third option, not described on the diagram – if SP wants to send the user the proof directly in the off-chain phase (instead of doing so optimistically), it can do so. In this case the user's off-chain cheque acknowledgement must include the fact that user does not require batch proof to include this particular request, and SP can claim the fee for it as soon as it wants. This approach is more immediate, however creating individual non-aggregated proofs is more costly to the storage provider, so we envision this mode of communication to be rather particular to applications which can afford to pay for almost instant response authentication guarantees.

Design rationale and security (sketch). We analyse several properties of our protocol separately, starting from the off-chain flow first.

Off-chain flow properties:

1. After the off-chain exchange, SP can be forced to include the batch proof, and SP will get the payment for it. This is clearly achievable since the user and the SP exchange signatures. Storage providers gets a cheque that it will attach to the sequencing transaction later. If SP does not include signature, the user can challenge SP. Since transactions have to be included in a particular order, it is not possible for the SP to insert it after reads to related commitments (previous and next) were included.
2. If the off-chain exchange is aborted, both parties incur negligible losses (and can then enforce the on-chain flow). If SP aborts after the request is sent (SP does not reply with an answer), SP can still post this request on-chain, in which case SP will incur costs without being paid. If the user aborts after SP sends the answer, then SP has to post this request on-chain without being paid; a rational SP will the enforce the on-chain flow. We consider the cost losses to be negligible (single read fee), and the SPs to be more powerful than the users (so they bear this negligible cost and not the user).
3. The interaction cannot be replayed. The only action that the agent could replay is to double blame the SP; however this is impossible due to the fact that the signature is binding the request, and slashing cannot occur twice with respect to the same request. The only action that the SP can replay is to attempt to issue several read transactions or claim the cheque several times: however the cheque binds the request, and every request has a unique index and is only valid within a single timeframe or epoch.

On-chain flow properties:

1. If user posts a read request on chain, it will be sequenced and fulfilled, and the user will successfully fetch it afterwards. This is due to the fact that the read requests have to be fulfilled by the read ledger: if SP does not post the response with signature and VID sig, and cross terms, it will be slashed. Since VID signature has to verify, by honest majority assumption we are convinced that the user will be able to download the answer from the VID afterwards.
2. If the user posts a read request, the SP will be paid after it fulfils the request. This is trivially guaranteed by the virtue of locking the payment, and unlocking it after the batch proof is delivered.

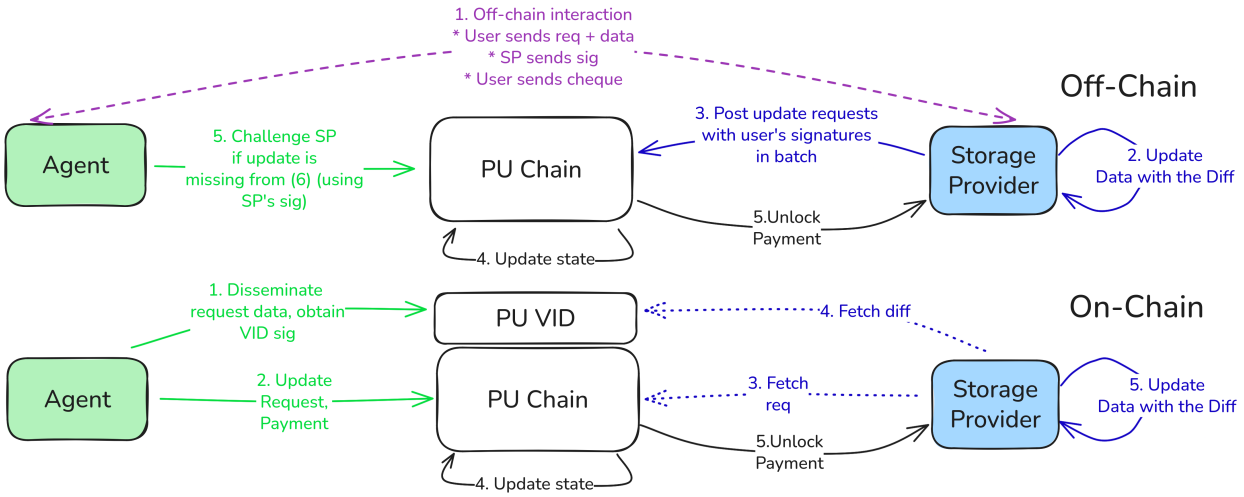

Figure 7: Update request flow.

3. Interactions cannot be replayed: trivially guaranteed by the virtue of all the messages going on-chain: SP cannot replay user's query or payment because it is registered once, and un-registered when the batch proof is delivered.

3.5.3 Update Action

An update is a user-issued action that requests a data update at certain indices. It is issued to the SP or the storage contract in charge of the data, by anyone authorized by the terms of the contract. An update can be requested either off-chain, or on-chain, with a dedicated set of transactions for each of the methods. We describe the update action in Fig. 7 in two variants: off-chain and on-chain.

In the on-chain version, the requester computes the diff needed to update the data and commitment, and then sends an update request to the chain. If the diff is heavy, which we assume to be quite common, the fisrt step is to actually disseminate the diff to the VID, and post the on-chain transaction with the VID signature. When the request transaction is included in a block, the chain uses this diff to update the contract's commitment in its internal state. The storage provider should monitor the network, update the storage, and recompute the commitment. No response is expected from the storage provider, since the next read proof will accurately acknowledge the request.

The off-chain version requires the user to compute the diff and the updated commitment, and send an update request containing this new commitment directly to the storage provider. The storage provider then checks that the data is consistent with the user's signature, and replies with the signature acknowledging the request validity. The user replies with the cheque, which allows SP to later claim user's funds. After some time the transaction is included by the storage provider. The payment locking for the off-chain update follows the same epoch system as the storage request payment. If for any reason the storage provider does not include the update request that it signed earlier into the block, the requester will have to post an on-chain challenge instead , which will lead to SP being slashed.

Essentially, in the off-chain flow the storage provider is doing the sequencing and has to reconcile the on-chain transactions with off-chain transactions, thus publishing the off-chain transactions together with user's signatures is necessary to establish a canonical request order. It is imperative that storage providers process on-chain request transactions the same order they appear in blocks. This means that all reads before an update must be proven with the old commitment, and all reads after must be proved with the new commitment.

There is no proof submitted for this operation, but following reads / proof requests will expect proofs

matching this new commitment. If the storage provider did not correctly update the data, it will be unable to provide these future proofs.

Design rationale and security (sketch). The interaction and the properties are very similar to the read action.

For the off-chain flow:

1. If the off-chain interaction succeeds, then either the update will be submitted by the SP and SP is going to get paid, or SP is going to get slashed. This is due to the signature exchange in the off-chain part: the signature from SP ensures user can challenge SP on update non-inclusion, and the signature from the user (cheque) makes sure SP will claim the money from the user.
2. If SP aborts after user sends the request and data, no party loses any funds. SP *could* post the update on-chain without being paid, but it does not have to.
3. If the user aborts after SP sends the signature, then SP has to post the update on-chain, but it will not receive the payment, and has to force the on-chain interaction. This is similar to the read proof, and the lost cost is negligible.
4. No messages can be replayed. Similar to read proof – requests are unique transactions.

For the on-chain flow:

1. After user posts the data to the chain (and VID), the storage provider will have to, and will able to update the data and reply to the further read queries. This is trivially guaranteed by everything being on-chain: the chain knows the update directly. download SP will be able to do this since by honest majority assumption it will be able to download the data from the VID.
2. If the user posts the update request on-chain, the SP gets paid. Trivially due to the payment being attached to the on-chain transaction.

3.6 Contract Lifecycle

After the contract is created, it is valid up until a future block proposed by the requester and agreed upon by the storage provider. The end block can be extended by a "renew transaction" sent by the requester that initiated the contract, which is either acknowledged by the storage provider, or a termination request is sent by the SP when the deadline has been reached, so they may collect the collateral in the contract balance and then terminate the contract.

Contract termination. At any time, the storage provider can send a termination order, with a compensation specified in the contract.

- If the requester accepts, it sends an acknowledgement; the contract is then deleted, the collateral is unfrozen and a compensation, specified by the terms of the contract, is taken from the collateral and sent to the Requester
- If the requester refuses, no termination is performed

Contract renewal. At any time, the requester can send a renewal request with new terms

- If the storage provider accepts, it sends an acknowledgement. The contract is updated, and, if needed, collateral is adjusted.
- If the storage provider refuses, the contract remains as it is.

Slashing. Slashing is a mechanic in the protocol to dis-incentivize storage providers from misbehaving. Each underlying contract account has collateral as part of its ledger balance. The collateral is submitted on the final contract agreement transaction, and is subject to slashing if the proof deadline in the contract state is violated.

A slashing mechanism which is activated if a transaction is not included is unusual. Indeed honest player could be slashed if the chain does not work properly. In particular it is common in all blockchain to have sequences of (nearly) empty blocks due to issue on the peer-to-peer network, or a bug in the block production software. To avoid this we propose the following. Set n^8 the number of blocks we deem necessary to include an operation when the chain works correctly. The naive mechanism would decrement n at every block, until the proof is submitted, or it reaches zero and the player gets slashed. We instead compute the proportion of wanted proof that are being submitted at each block. This will be our measure of the chain health. If no proof are submitted it is zero, and if all are it is one. We would then decrement by this proportion.

Each slash included in a block will be represented as an internal transaction, where a percentage of the collateral is rewarded to the block producer, and the remaining amount is sent to a treasury.

Each block producer is responsible for including slashed transactions in proposed blocks. There should be a reward for including them in blocks. In each block, space is reserved for a certain number of internal transactions. If a proposed block misses a slash when it could have been included, the network will invalidate the block. Validators and block producers can run a process locally that is used to keep track of contract accounts that are likely to be slashed.

Additional discussion on the slashing mechanism are discussed in **??**

Guarantees.

- Network validates internal transactions with the ledger
- Block producers may not submit slashing transactions for requests that are not expired
- Self slashing and reporting is not possible because the block producer is rewarded, and the reward is only percentage of the collateral
- It is not possible to submit a block without including internal transactions when internal transactions are available

3.7 Internal Transactions

These types of transactions are used as internal triggers that modify the ledger when a deadline is met. The two instances where this occurs is slashing and storage contract deletion. When a block is proposed, queries will be made to the proof ledger and the staged ledger to find when a deadline is violated. Because block space is limited, not all the violations can be acted upon at the same time. Thus, there is some leniency to the deadline logic.

3.8 Data Representation: Encoding and Decoding, Encrypting and Decrypting

Cryptographic primitives only understand the algebraic structures it is built upon (i.e. scalars from $\mathbb{F}$ and elliptic curve points): any data that deals with a polynomial commitment scheme needs to be encoded as scalars.

For this reason, the storage providers, and, more generally, the chain will never deal with data as bytes, only as scalars; the storage requester is responsible for encoding the data as scalars when sending it to a node, and decoding the scalars as data when receiving them. The encoding process consists in writing bit-to-bit the raw data in a scalar, and the decoding process is symmetric.

In addition to the encoding and decoding, the storage requesters have to be aware of the fact that their data is always susceptible to be published on-chain, through reads or updates; they have the choice to encrypt the data before sending it, using an encryption scheme that will not harm data indexing logic for read and updates. Many different data encoding schemes may be applied when writing data, based on the application's use case.

[8] This number is quite low. We need to account for block producer censoring operation, and the time to re-emit the operation once or twice if something goes wrong. This could be the order of ten.

4 Succinct Consensus

We envision Ensue as a succinct chain, similar to Mina, with the ability to verify the entire chain in constant time. More specifically, one should be able to get the Merkle root of a blockchain state, a height, and a SNARK proving that this indeed is the Merkle root of the Ensue chain at this height.

To achieve this, we need our consensus to be SNARK friendly and instant finality. We will construct such a consensus in this section, together with a SNARK statement proving the transition from one state to another. Note that this construction differs from Mina's construction in that our consensus is BFT based, not Nakamoto based, in order to achieve instant finality. This is essential for our agentic use case presented in the next section. This also allows us to create a SNARK proving the strong property that this is *the* state at a certain height, which is not something Mina can enjoy due to the forks possibilites of Nakamoto based consensus.

High level outline. The idea for our full protocol is to execute a PBFT type consensus, while adding a SNARK verifying that 2/3 of the finalization were obtained for the block. The SNARK will be computed by SNARK workers (see Section 3.2). The SNARK statement can be splitted in several subparts, so that SNARK worker can work concurrently on small pieces. One of the main technical difficulties of this section is to avoid checking a large number of signatures in the SNARK, which would incur a prohibitive cost, which we address using multi-signatures presented in 4.4.

4.1 The Simplex Protocol

We choose as a basis for our construction the Simplex protocol [CP23] due to its performance and simplicity. Note that we can also apply the modifications from [Sho23]. We recall the Simplex protocol (omitting some aspects) in this section for completeness, before adapting it to make it SNARK friendly and permissionless. For a complete description and proof of security of Simplex see Section 2 and 3 in [CP23].

Simplex is a very simple protocol quite similar to PBFT and Tendermint. At each round the block leader proposes a block, and then the validators need to reach a quorum of 2/3 twice: once for pre-voting phase, and once for finalization phase. Fix a set of players $[n]$. Denote by $\mathbf{L} : \mathbb{N} \to [n]$ the block leader selection function associating a player to a block height. In our model, messages are broadcasted to the whole network, which is often practically implemented via the P2P network layer underlying many blockchain implementations. The node validator workflow per block is as follows:

1. **Leader proposal.** A height h, compute $\mathbf{L}(h)$, if you are the block leader, and broadcast your proposed block.[9]
2. **Dummy block.** Start a timer. If no block proposals are received at the end of the interval defined by the timer, pre-vote for the dummy block.
3. **Pre-vote.** If a correct proposal is received, broadcast your pre-vote for it.
4. **Finalize.** Upon seeing 2/3 of the votes for a block, broadcast finalize and the pre-vote quorum for this block. Cancel the timeout timer. Move to the next height.

Before presenting the SNARK statement in Fig. 10 we slightly modify Simplex.

4.2 Leader Election and Dummy Block

Election Leader Our first modification is about the leader election. The simplex algorithm works in a permissioned setting : it assumes a fixed ordered set of participants, allowing for a simple leader selection function based on a random oracle applied to the height of the chain. Our set of participants is dynamic, and thus a deterministic leader election as in Simplex would allow an adversary to create advantageous public keys.

[9] This part is actually done on the notarized chain, i.e. before the finalization quorum has been gathered. However, this section focuses on making simplex SNARK friendly so we omit networking details.

To thwart this attack, we add some unpredictable value. Each consensus participant will generate a key pair for it's Verifiable Random Function (VRF). Each block of the chain will contain a random seed, defined to be the evaluation of the block producer's VRF applied to the last seed, as done in Algorand [CM19]. As in Mina, we choose the VRF presented in Appendix C of Ouroboros Praos [Dav+18], which is SNARK friendly. We can then easily derive from the seed a priority list (weighted by stake) of block producers for a given height. Note that the choice of the leader will not be proven in the SNARK. This is not an issue, as honest participants will not vote for a block created by the wrong leader.

Dummy block. Having a priority list instead of a single leader per height also allow us to skip dummy blocks. More precisely, vote for dummy block (i.e. timeout) are still casted, but when receiving a quorum for a dummy block, the next block producer from the priority list is chosen. This is a nice optimization in our case, as we would not want to create an expensive SNARK for a dummy block.

4.3 Incentives and Security Assumptions

Our second modification is adding incentives. Simplex analyses its algorithm in a classical Byzantine setting, assuming that a certain portion of the stake is Byzantine (behave arbitrarily), and the rest follows the algorithm. What we want in a blockchain setting is a bit stronger. Similarly, a portion of the stake will be Byzantine. What we then want is that following the algorithm is the best strategy for the remaining participants. Another way to put it is that we assume that non Byzantine consensus participants are *rational* rather than *honest*.

We incentivize people to participate in the consensus by adding some monetary creation benefiting the block producer and participants whose finalization messages are included.

We have to additionally implement a slashing mechanism when someone pre-votes for two blocks at the same height. Indeed, it is otherwise rational for a consensus participant who signed the wrong block to broadcast a second signature when he realizes his mistake, in order to get his reward. This could lead to forks, thus to remedy this we rely on a denunciation mechanism. If a consensus participant sees two pre-votes or finalization messages from the same person in the same round, he can send a denunciation operation including the two signatures as proof. The double signer is slashed, part of the slash is given as a reward to the denunciator, and the rest is sent to the treasury[10]. The chain also needs to check that the double signer is slashed only once. To do this in the succinct setting we use a variation of Merkle tree allowing to efficiently prove in a SNARK non-membership: Cartesian Merkle Tree [CKR25].

Due to slashing, the consensus participants need to freeze a part of their stake to ensure they can pay the slashing fine if needed. Only the stakeholders who perform the freezing operation will participate in the consensus, and the 2/3 of the stake necessary for the quorum will be calculated based on the share of the stake taking part in the consensus.

4.4 Multi-Signatures and SNARK Efficiency

4.4.1 Purpose of Multi-Signatures

We now present our integration of multi-signatures, which is our last modification. The Simplex consensus (as other BFT based algorithms) requires a lot of signature verifications to be performed at each step. While we can use regular signatures on a curve which is native in our SNARKs (e.g. compatible Schnorr with Poseidon hash), verifying too many of them can still be costly be: elliptic curve multiplications, even native, are costly; and having numerous public keys as public input also is.[11] For these reasons, we will use a multi-signature scheme to significantly reduce the cost. Using a multi-signature scheme alone does not lead to a significant gain. But we can improve in-circuit performance by using a technique that saves us a large part of the cost for aggregating the public key of the signers that were present in the last signing committee.

[10] An amount needs to be burned to avoid adversary denunciating themselves, which effectively nullifies the slashing.

[11] We could publicly input a commitment to the signatures which we open inside the SNARK, but the cost of opening in the SNARK would also be prohibitive.

Naive approach 1	Multisig approach	Multisig and diff approach
$n \cdot (\mathsf{SigVer} + \mathsf{MT} + \mathsf{Add})$	$n \cdot (\mathsf{SigVer} + \mathsf{EC} + H + \mathsf{Add})$	$n \cdot H + n_{\mathsf{diff}} * (\mathsf{EC} + \mathsf{MT} + \mathsf{Add})$

Figure 8: This figure compares the cost of a naive approach, and using multi-signature with and without a diff approach. n is the number of consensus participants whose signature is included in the block. n_{diff} is the number of consensus participants whose signatures were not present in the last block. In practice we expect $n_{\mathsf{diff}} \ll n$. H is a hash, MT a Merkle tree path (approximatively 20 to 30 hashes), EC an elliptic curve multiplication, SigVer a signature verification (approximatively one hash and two elliptic curve multiplications), and Add an addition on the number of bits chosen to encode the stake share.

We call this technique the diff approach, as the SNARK applies a diff to the previously aggregated public key and the share of the stake. We sum up the complexity of the SNARK checking the quorum with different approaches in Fig. 8. The chosen approach with the multi-signature and the diff is presented in the rest of the section. The elliptic curve multiplication and hashes are paid to verify (multi)-signatures, while the Merkle tree path and addition to verify that enough of the stake is present.

4.4.2 Choice of MuSig2 as a Multi Signature Algorithm

We base our work on MuSig2 [NRS21], which we present in Section 4.4.3 with some modifications. A multi-signature scheme is an interactive protocol in which a group of signers interact to produce a joint signature on a message. In our case the signers are consensus participants, and the message is the block they vote for. However, unlike in the case of multi-signatures, our group of signer is not fixed. Some consensus participants might disconnect (unintentionally or maliciously) during the protocol and we cannot afford to abort in this case. What we would really want is a non-interactive protocol: every consensus participant attaches their signature to the block, the block producer collects them, computes an aggregated public key and signature of the voters, and publishes the block with the signature and aggregated public key. Non-interactive multi-signatures do not exist, but MuSig2 will suffice for two reasons:

- MuSig2 is only two round, with the first one being a pre-computation round. This is enough for our use-case, using the blockchain to publish in advance the outputs of the first round.
- MuSig2 can also work when some participants disconnect in between the two rounds, which is necessary to avoid aborting (which is necessary in e.g. FROST and variants [KG20; CKM21; Ruf+22]).

4.4.3 Consensus-friendly Variant of MuSig2

We present the modified version of MuSig2 in Fig. 9. Our main modification is to adapt to a setting in which some user that were to participate disconnect. To achieve this we simply absorbe more data into the random oracle in the second phase. For this we introduce three sets of participants: $S \subset P' \subset P$. P will be the set of consensus participants, i.e. a superset of the signers. P' will correspond to participants in P who have published their pre-computation round. And S will correspond to the actual signers. We also use the first phase of MuSig2 non-interactively as a pre-setup.
We fix a group $\mathbb{G}$ where the discrete logarithm is hard, $\mathbb{F}$ the finite field of cardinal the order of the group (i.e. the scalar field of the curve), and g a generator. A variable indexed by S, P or P' is the set of variables corresponding to the actors in S, P or P'. We use a random oracle $\mathcal{RO}$ with persistent state. $\mathcal{RO}$ is equipped with the functions $\mathcal{RO}.\mathsf{absorb}(\cdot)$ and $\mathcal{RO}.\mathsf{squeeze}(\cdot)$. Unusually, the squeeze functionality can take an optional argument. When squeezing is performed with an optional argument, the state of the random oracle is not updated.

4.4.4 Protocol Flow using MuSig2

We want to use the alogrithms from MuSig2 in the context of Simplex. Each block defines a set of consensus participants P. At each block our goal is to create a multi-signature of a subset S of consensus participant

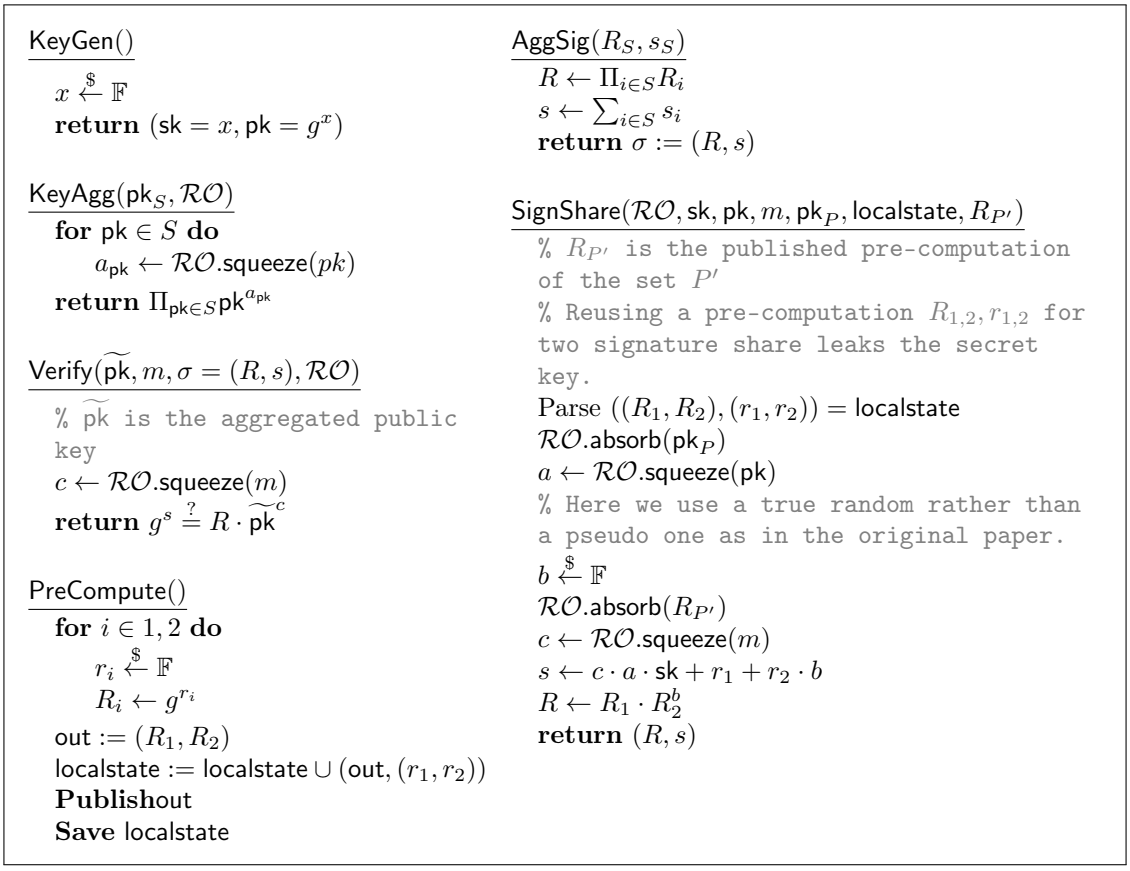

Figure 9: The SNARKified consensus-friendly variant of MuSig2

$\mathsf{BlockTransition}(\mathsf{st}_{\mathsf{old}}, \mathsf{st}_{\mathsf{new}}, B, \sigma, \mathsf{pk}_{\mathsf{diff}})$

Fetch $\mathcal{RO}, \mathsf{pk}_{\mathsf{old}}, R_{P'}, \mathsf{rt}_{\mathsf{stake}}, n_{\mathsf{stake}}, n_{\mathsf{last}}, \mathsf{rt}_{\mathsf{removed}}, \mathsf{rt}_{\mathsf{added}}$ from $\mathsf{st}_{\mathsf{old}}$
Compute $\mathsf{pk}_{\mathsf{new}}$ from $\mathsf{pk}_{\mathsf{old}}$, $\mathsf{pk}_{\mathsf{diff}}$ and $\mathcal{RO}$
Verify that the set of signer does not contain signer no participating in the consensus using $\mathsf{pk}_{\mathsf{diff}}$ $\mathsf{rt}_{\mathsf{removed}}, \mathsf{rt}_{\mathsf{added}}$,
Update $\mathcal{RO}$ with $R_{P'}$
Verify σ against $\mathsf{pk}_{\mathsf{new}}$ with the updated $\mathcal{RO}$
Verify that enough of the stake is present using $\mathsf{rt}_{\mathsf{stake}}, n_{\mathsf{stake}}, n_{\mathsf{last}}$
Verify that block B applied to $\mathsf{st}_{\mathsf{old}}$ gives $\mathsf{st}_{\mathsf{new}}$ (which includes the update of $\mathcal{RO}, \mathsf{pk}_{\mathsf{old}}, R_{P'}, \mathsf{rt}_{\mathsf{stake}}, n_{\mathsf{stake}}, n_{\mathsf{last}}, \mathsf{rt}_{\mathsf{removed}}, \mathsf{rt}_{\mathsf{added}}$)

Figure 10: The SNARK statement for a block transition.

representing 2/3 of the stake. We will provide in the next section a SNARK statement using that multi-signature to achieve succinct consensus. In this section, we present the non-"snarkified" protocol. The chain will act as the verifier.

Whenever they want, consensus participants execute $\mathsf{PreCompute}()$ as many times as they want and publish the result on chain. If no pre-computation are present in the last state, they will be temporarily excluded from the consensus. Then, at each block, the Simplex reaches its **Finalize** step in which the participants should broadcast a signature of the block :

- The consensus participants execute $\mathsf{SignShare}$ with an empty random oracle and send their share to the block producer.
- The block producer verify those, using the usual Schnorr verification, i.e. Verify with an empty random oracle[12]. He then executes AggSig, and broadcast his block attaching the resulting signature and the set S.
- The chain absorbs pk_P, and executes KeyAgg on S. The chain absorbs $R_{P'}$ (i.e., taking one precomputed R for each consensus participant who has one available) in its random oracle and executes Verify with the aggregated key and updated random oracle. The chain then deletes the used pre-computation for each member of P'[13].

4.5 SNARK Statement for Succinct Consensus

We present the SNARK statement to prove the validity of our consensus in Fig. 10. This is the snarkified version of Section 4.4.4, with an additional twist: we reuse the set of signers from the last block and apply a diff to compute the aggregated public key, rather than computing it from scratch. We also reuse the old random oracle to absorb a diff. This allows us to achieve the complexity of Fig. 8.

The transition functionality is as follows. st is the state of the blockchain (in practice, we publicly input a Merkle root of it). This statement proves a transition between two states $\mathsf{st}_{\mathsf{old}}$ and $\mathsf{st}_{\mathsf{new}}$. Our state will store (among other things): $\mathsf{rt}_{\mathsf{stake}}$ the root of the Merkle tree of the current consensus participant, n_{stake} the total amount of stake in $\mathsf{rt}_{\mathsf{stake}}$, the random oracle initialized with pk_P, the aggregated public key used in the last block $\mathsf{pk}_{\mathsf{old}}$, the amount of stake this public key amounted to n_{last}, the Cartesian[14] Merkle root of consensus participants removed in the last block $\mathsf{rt}_{\mathsf{removed}}$, the Merkle root of consensus participants removed in the last block $\mathsf{rt}_{\mathsf{added}}$. The block producer supplies $\mathsf{pk}_{\mathsf{diff}}$, two lists of public keys, containing the signers present in the last block and not in the new, and vice versa (i.e. the keys to add and those to subtract from $\mathsf{pk}_{\mathsf{old}}$.

[12] Note that this also act as a verification of the message authenticity

[13] We delete a pre-computation for each member of P', not S as it is possible that a member of P' signed with its pre-computation and that the signature would not have been included in the block.

[14] we use a Cartesian Merkle tree here as we will need proof of non membership

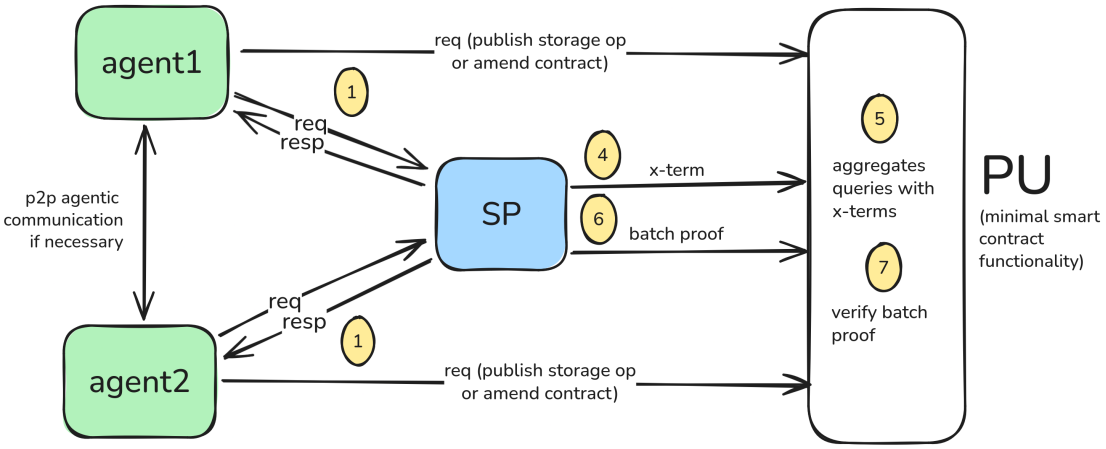

Figure 11: Standalone execution workflow between several agents and Ensue.

The only cost which is linear in the number of signers is the update of the random oracle with the pre-computation round. One hash is the cheapest cryptographic operation we can do, so having a linear cost in this is optimal. Note that if enough of the stake signs, the block producer can choose to use only the signatures of participants with big portion of the stake and throw away the rest of the signatures. Since in practice the stake follows a steeply right-skewed distribution, this allows one to use a much smaller number of signatures.

The SNARK for a transition will be aggregated in the SNARK of the chain, effectively proving that this is *the* block at this height of the Ensue chain.

5 Agentic Execution Layer

This section describes how agents can interact with each other using Ensue. Our protocol provides great flexibility in this regard, thus we present several possible scenarios with increasing levels of functionality and complexity, aimed to suit most types of existing and future agentic AI usecases.

5.1 Standalone Execution Mode

Agents can use Ensue in a standalone way, as a pure memory layer, relying on Ensue for correct retrievability, data availability, and permission control. Without Ensue, none of these three properties are guaranteed unless one trusts the storage provider, or tries to lower the necessary trust by interacting with several SPs, potentially with the help of a reputation system.

The premise of this scenario was explained in Section 1.1. In this section, we build on top of it, adding more details to the components. The standalone workflow is presented in Fig. 11.

contract types allowed. However, this is already enough for many real-world applications, as the following two examples illustrate.

Data selling scenario. In this scenario, a *seller* party owns certain data that it periodically updates (e.g. financial statistics, or CVs of chosen candidates per category who are on the job market), and *sells* to potential *buyers*. Data can be sold in two ways: either as a "single access", or via the "subscription" model: some users access all the data, lower tier users only access some parts of it. Using Ensue, implementing this is easy – the seller first allocates the storage and uploads the data, and then changes read permissions for other parties via amending the terms of the contract. Thus in the subscription model the access is granted

without a deadline, but on a particular segment. In the one-shot model the access should be given only for a fixed, short amount of time, enough for the party to download the data.

Let us walk through exactly how permissions are enforced. The basic guarantee that Ensue provides is that all the valid reads and updates will be executed. What the blockchain system *does not* enforce is the opposite – that the invalid reads and updates will not be executed. However, the updates are not an issue – given that SP needs to maintain the "correctly updated" version of the state, it is not economically viable for it to maintain several concurrent versions of the state. The inability to prevent malicious reads from happening is generally not possible without some sort of encryption, and assuming there is none, we assume that an honest SP does not have any interest in giving away user's data to random participants. These concerns may be additionally reduced by introducing a reputation system of sorts. The data could also be encrypted, which would guarantee privacy against malicious SPs – however, then the seller unlocking the data for a buyer needs to publish a symmetric key encrypted under the buyer's public key. A similar system is implemented in the Atomic and Fair Data Exchange via Blockchain$^{(1)}$ paper, however the main soundness bottleneck here is that the buyer must be convinced that the data is indeed encrypted with the secret key. This could be proven using an additional SNARK, but this adds much more complexity than, perhaps, is rationally necessary.

Finally, note that when the data is not large, the users can request normal read queries from the SP, and, as discussed before, these will be satisfied, and their execution is enforced by the chain. However, if the data is big, instead of proving read with respect to a data vector, the SP will prove the read with respect to its (concise) commitment. The data itself has to be then sent off-chain to the party who is requesting the read, and, as we discussed earlier, a non-collaborative SP retaining the data will be forced to disperse it using VID in the worst case. Thus, the protocol works very similarly even in the case where large data chunks have to be downloaded at once.

Generic PSO architecture. Now we present the following generic "proposer-solver-oracle" (PSO) scenario. The workflow's three main parties are: *proposers* who create requests, highly specialized *solvers* who compete to provide solutions, and impartial *oracles* who evaluate those solutions (think of an oracle as a "judge"). The process begins when a proposer creates a smart contract containing their problem statement (such as finding suitable apple buyers with specific requirements) and uploads it to a Ensue's chain. The proposer then selects one or more oracles who specialize in fact-checking and reasoning validation, setting their public keys in the same contract. The proposer also establishes rules for participation, including submission costs, eligibility criteria (algorithmic or non-algorithmic), timeframes, and privacy settings that determine who can access what information. Solvers must be authenticated to submit their responses: authentication can be either trivial (anyone who pays certain amount may submit), or it might require talking to the user in advance for it to whitelist the solver. Once authenticated, solvers upload their solutions to designated storage locations within the contract, potentially including structured data with references for better provenance tracking. Oracles (or the proposer itself) then thoroughly evaluate each solution and upload their assessments. After receiving multiple solutions or reaching a timeout, the proposer selects a winner based on the oracles' evaluations. In a more fair implementation, the proposer must justify their selection, and the oracle will validate this reasoning to prevent conflicts of interest. Once complete, rewards are distributed according to the contract terms, relevant data may be preserved in long-term storage, and the original contract is removed. Note that during this process the proposers might not see other proposers' solutions, to guarantee more fair competition.

Despite its simplicity, the PSO architecture is incredibly generic; here are some concrete examples of how it can be instantiated. We omit describing the oracle, since it is role-agnostic and similar for all the examples.

1. Security audits:
    - Proposer: a party who wants the code investigated. The code can be either private (committed in the smart contract), or just referenced by the URL.
    - Solver: bug finding AI that submits vulnerability reports to Ensue (privately), and expects to be paid for it. Given that the fact of data upload is visible on chain (while the data can be sent

off-line), this serves as an evidence of the upload, thus making it harder for the proposer to deny the reward.

2. Healthcare insurance quotes:
   - Proposer: individual who has access to the private healthcare data, or a private clinic having access to the data of their clients. Proposers might only want to authenticate solvers who are accredited by the healthcare system / have high ratings, and they do not want data to be stolen.
   - Solvers: insurance companies competing to give the best quota to users (individually or through their clinics).
3. Headhunting & recruiting:
   - Proposer: person who is looking for a job. The CV and experience is private. Wants to authenticate only companies that are in the list "or similar to that" that it gives to the judge. The solvers do not know it.
   - Solvers: companies looking for jobs, or HR agents looking for candidates for their sub-contractor companies.
4. Targeted private recommendation system (or advertisement):
   - Proposer: a user who has their recommendation vectors stored in Ensue, and who wishes to receive customized feedback and recommendations, but does not want to sell the data to the big tech.
   - Solver: a party who can query only some parts of the user's state (e.g. corresponding to particular direction of interests), and suggest a more targeted recommendation strategy.
   - This usecase aligns well with Vana's approach of "letting users own their data while allowing other parties to derive some value from it".
5. Regulation compliance:
   - Proposer: a regulated party publishing periodic reports of its activity to its own private SP "bucket".
   - Solver: a party that can justify why the published behaviour is well-regulated according to the law, or otherwise flag it.
   - Oracle: one of the accredited regulation compliance judges, who might enter the procedure if something is flagged.

Many more examples of PSO shape can be thought of: supply chain optimisation, accounting or tax advice, distributing rewards for DAO users, etc.

5.2 Leveraging External Execution via Bridges

Note that the previous standalone mode does not achieve *full fairness* if we assume that Ensue is not a full-fledgded L1 execution layer with arbitrary smart contract support itself. The parties have to trust each other to receive the rewards (or release the data), or judge the quality of the answers provided. This can be solved by using Ensue together with an actual generic smart contract chain.

Luckily, Ensue is designed to easily bridge with other execution layers (L1s and L2s such as Mina or Ethereum), thus allowing agents to engage in truly decentralized interactions, and compete without taking risks and sacrificing fairness. This is the true spirit of Web3 – agents interacting with a smart contract can define rules on how they can solve problems, collaboratively or competitively, and distribute rewards according to the services provided.

Bridging with Ethereum. The bridged workflow is presented in Fig. 12. First, it is important to note is that this example is not ethereum-specific and can be easily generalised to other L1 or L2 platforms; ethereum is chosen for explanatory convenience and simplicity.

The main difference with the standalone model is that in in the bridge model the transaction sequencing is externalised. Instead of submitting transactions directly to Ensue, users now by default talk directly to

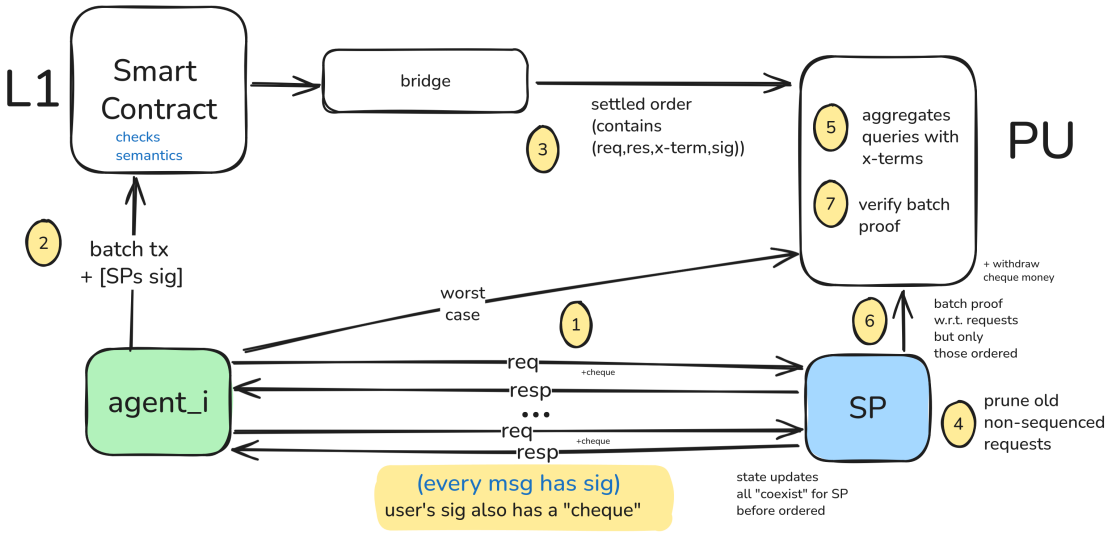

Figure 12: Bridged execution workflow with an external execution layer (here, Ethereum as L1).

the SP, and receive "promises" in the shape of signatures on the requested queries. Then the L1 transaction to an ethereum smart contract can be executed, where the smart contract will accept SP's signature instead of a proper read proof. After some time, these transactions settle on ethereum, and thus bridge makes them available on Ensue. Since then the workflow is as before: the Ensue and SP collectively aggregate the "promises", and eventually SP has to batch-prove them correctly.

One crucial difference is that due to the fact most L1s and L2s do *not* have instant (or even fast) finality, and since we want to support potentially high transaction throughput, we have to assume that in this model SP does not "know" which state transition is correct. Instead of assuming a linear history, SP only assumes the "true settled" state as dictated by the bridge. Before transactions settle on L1, concurrent state progression are "ephemeral" for the SP and can co-exist at the same time. An easy illustration of this behaviour would be to imagine two independent L1 miners competing for the block, in which case there are two "ephemeral state realities" defined by two unequal memory pools (transactions on L1 can be reordered).

This workflow is much more powerful. Instantiating the PSO scenario we presented earlier now gives us much more power. For example, now permissioning can be programmatic – e.g. a proposer is able to submit a voice if it satisfies a certain predicate (whitelisted by another smart contract, can present a DID reputation signature, interacted with this user before), or proposers can be compared automatically for winning (e.g. vulnerability report with the highest objective score wins). In the data selling example, users now can programmatically assert that the data they update the contract with satisfies certain criteria (e.g. it's signed by a third party for provenance reasons). In other words, having a full-fledged smart contract allows agents to not only read and write into memory and control its permissions, but also decide on the terms of the interaction in a fair, predetermined, smart-contract regulated way.

Bridging with Mina and zkApps. Mina, being an L1, can be a drop-in replacement for ethereum in the previous example. One additional layer of complexity in this scenario is added by the fact that ZK transactions on mina cannot be easily reordered, as private transactions have to refer to a particular state of the chain. While this does not impede the functional aspects, it likely decreases performance, as Mina is, first and foremost, a settlement layer. Ensue can integrate with mina directly, but it would most likely be more efficient to bridge with protokit instead.

Benchmark	Lower Bound	Average	Upper Bound
FoldingProver	37.932 ms	38.138 ms	38.363 ms
FoldingVerifier	499.56 µs	500.24 µs	500.96 µs
ProverRelaxed	732.84 ms	736.66 ms	742.93 ms
VerifierRelaxed	80.869 ms	81.115 ms	81.389 ms

Table 3: Benchmarks for folding and proving protocol presented in Fig. 3 and Fig. 2.

Digests and light L1 contracts. We imagine that a smart contract on L1 side contains a list of commitments as the way to represent the data. This achieves high throughput and allows for transaction reordering on L1 side. However, a double Pedersen commit digest is also possible, which will decrease the storage amount in the contract even further, at the cost of asking the blockchain to do a bit more work. We discuss this in **??**.

Other remarks. Note that agents do not even need to know about Ensue in this bridge scenario (otherwise necessary for access contract modification and storage allocation), and can execute most commands through the bridge or direct 1:1 connection to storage providers. This even further developer developer experience.

5.3 Lowering Entry Threshold: MCP Proxy

To simplify developer experience, we also envision people running MCP proxy servers which can interact with Ensue chain instead of them, thus freeing developers from the need to use any blockchain-related infrastructure at all. A server like this can use the same SDK as we provide to the end users, but to "proxy" its actions to other agents via MCP. This hybrid model may be particularly attractive to developers new to web3, or people who may not use our SDK directly for some technical reason.

6 Evaluation and Performance

We are currently in the process of implementing a prototype and testing its performance end-to-end.

Benchmarks for folding and proving aggregated proofs are presented on Table 3. This was run on a standard consumer machine (16-core AMD Ryzen 7 PRO 8840U). We envision read proofs, together with consensus verification itself (dominated by verification of signatures and stake recomputation), to be the biggest bottlenecks in the system.

The estimated costs of the naive Plonk-based VID were previously presented in Section 2.3.

References

[Bea+24] J. Bearer et al. *The Espresso Sequencing Network: HotShot Consensus, Tiramisu Data-Availability, and Builder-Exchange*. Cryptology ePrint Archive, Report 2024/1189. 2024. URL: https://eprint.iacr.org/2024/1189.

[CKM21] E. Crites, C. Komlo, and M. Maller. *How to Prove Schnorr Assuming Schnorr: Security of Multi- and Threshold Signatures*. Cryptology ePrint Archive, Report 2021/1375. 2021. URL: https://eprint.iacr.org/2021/1375.

[CKR25] A. Chystiakov, O. Komendant, and K. Riabov. *Cartesian Merkle Tree*. 2025. arXiv: 2504.10944 [cs.CR]. URL: https://arxiv.org/abs/2504.10944.

[CM19] J. Chen and S. Micali. "Algorand: A Secure and Efficient Distributed Ledger". In: *Theoretical Computer Science* 777 (Feb. 2019). DOI: 10.1016/j.tcs.2019.02.001.

[CP23] B. Y. Chan and R. Pass. *Simplex Consensus: A Simple and Fast Consensus Protocol*. Cryptology ePrint Archive, Paper 2023/463. 2023. URL: https://eprint.iacr.org/2023/463.

[CS96] T. Coffey and P. Saidha. "Non-repudiation with mandatory proof of receipt". In: *ACM SIGCOMM Computer Communication Review* 26.1 (1996), pp. 6–17.

[Dav+18] B. David, P. Gazi, A. Kiayias, and A. Russell. "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain". In: *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*. Ed. by J. B. Nielsen and V. Rijmen. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 66–98. DOI: 10.1007/978-3-319-78375-8_3. URL: https://doi.org/10.1007/978-3-319-78375-8_3.

[DGO19] I. Damgård, C. Ganesh, and C. Orlandi. "Proofs of Replicated Storage Without Timing Assumptions". In: *CRYPTO 2019, Part I*. Ed. by A. Boldyreva and D. Micciancio. Vol. 11692. LNCS. Springer, Cham, Aug. 2019, pp. 355–380. DOI: 10.1007/978-3-030-26948-7_13.

[Fis18a] B. Fisch. *PoReps: Proofs of Space on Useful Data*. Cryptology ePrint Archive, Report 2018/678. 2018. URL: https://eprint.iacr.org/2018/678.

[Fis18b] B. Fisch. *Tight Proofs of Space and Replication*. Cryptology ePrint Archive, Report 2018/702. 2018. URL: https://eprint.iacr.org/2018/702.

[GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. Cryptology ePrint Archive, Report 2019/953. 2019. URL: https://eprint.iacr.org/2019/953.

[KG20] C. Komlo and I. Goldberg. "FROST: Flexible Round-Optimized Schnorr Threshold Signatures". In: *SAC 2020*. Ed. by O. Dunkelman, M. J. Jacobson Jr., and C. O'Flynn. Vol. 12804. LNCS. Springer, Cham, Oct. 2020, pp. 34–65. DOI: 10.1007/978-3-030-81652-0_2.

[Moh23] N. Mohnblatt. *Sangria: A Folding Scheme for PLONK*. https://github.com/geometryresearch/technical_notes/blob/main/sangria_folding_plonk.pdf. Accessed: 2025-07-11. 2023.

[NRS21] J. Nick, T. Ruffing, and Y. Seurin. "MuSig2: Simple two-round Schnorr multi-signatures". In: *Annual International Cryptology Conference*. Springer. 2021, pp. 189–221.

[PG+99] H. Pagnia, F. C. Gärtner, et al. *On the impossibility of fair exchange without a trusted third party*. Tech. rep. Technical Report TUD-BS-1999-02, Darmstadt University of Technology ..., 1999.

[PJS23] Y. Park, M. H. Jeon, and S. U. Shin. "Blockchain-Based Secure and Fair IoT Data Trading System with Bilateral Authorization." In: *Computers, Materials & Continua* 76.2 (2023).

[Rab+23] R. Rabaninejad, B. Abdolmaleki, G. Malavolta, A. Michalas, and A. Nabizadeh. "stoRNA: Stateless Transparent Proofs of Storage-time". In: *ESORICS 2023, Part III*. Ed. by G. Tsudik, M. Conti, K. Liang, and G. Smaragdakis. Vol. 14346. LNCS. Springer, Cham, Sept. 2023, pp. 389–410. DOI: 10.1007/978-3-031-51479-1_20.

[Ruf+22] T. Ruffing, V. Ronge, E. Jin, J. Schneider-Bensch, and D. Schröder. "ROAST: Robust Asynchronous Schnorr Threshold Signatures". In: *ACM CCS 2022*. Ed. by H. Yin, A. Stavrou, C. Cremers, and E. Shi. ACM Press, Nov. 2022, pp. 2551–2564. DOI: 10.1145/3548606.3560583.

[Sho23] V. Shoup. *Sing a song of Simplex*. Cryptology ePrint Archive, Paper 2023/1916. 2023. DOI: 10.4230/LIPICS.DISC.2024.37. URL: https://eprint.iacr.org/2023/1916.

[Tas+24] E. N. Tas et al. "Atomic and Fair Data Exchange via Blockchain". In: *ACM CCS 2024*. Ed. by B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie. ACM Press, Oct. 2024, pp. 3227–3241. DOI: 10.1145/3658644.3690248.

Appendices

A Non-Repudiation and Accountability

There is an inherent tension between the desire to make interactions off-chain (to improve performance) and inability to make them "fair". In this section, we elaborate on the problem and provide some intuitions justifying design of our protocol, especially related to the VID part.

If the storage provider (SP) stores some data that the user wants to update and the SP "promised" to do it, it is hard to perform this update without posting the data on the blockchain or some data availability (DA), VID, or gossip layer. That is, ideally we would want the blockchain to be concise, and the data to be only seen by the sender and receiver. But this seems hard, since it is hard to force two parties to interact, and prove whose fault was in not interacting.

The problem is sometimes called *"non-repudiation"*, as defined e.g. by Coffey and Saidha [CS96]. In their terms, non-repudiation is non-deniability that interaction happened. Non-repudiation guarantees that the origin of data and its integrity can be proven, making it impossible for someone to claim that they did not send or receive certain information.

Another work by Park et al. [PJS23] defines non-repudiation in the following way: "Non-repudiation and fraud prevention: The participants must carry out their responsibilities for data trading, and the transactions cannot be denied later by either of the parties involved in the data trading. Furthermore, malicious behaviours such as providing incorrect data or alleging false compensation must be prevented. In this case, no benefit should be provided to the malicious party".

There exists a related yet different problem of "atomic fair exchange", addressed e.g. in the work by Tas et al. [Tas+24]. Their problem is making sure that two parties either get what they want (A receives data, B receives payment) at the same time, or not at all. However, the assumption here is that parties want to interact, while in the non-repudiation case the parties might not want to. Our client might want to slash the server, and our server might not want to do extra updates work. An impossibility result for a similar "fair exchange" setting is presented in [PG+99], relating strong fair exchange to a consensus problem; again, this assumes that the parties are willing to intearct.

The following analogy might be useful in illustrating the impossibility of achieving non-repudiation without a third party. Assume a post service delivery (client) is tasked to deliver the user (server) a parcel. The claim is that "hard evidence of delivery" is not possible without two parties interacting, or a witness. Several cases are possible:

- The delivery service did not deliver the parcel, and instead they destroyed it, and there is no other parties in this world except for the user and the post service. The post service can say that they attempted the delivery, blame the user for not receiving it (the server was down / not responding). Adding photo or video technology into the assumption: the post service could make the photo of your porch with a parcel, but they could also have made that and still have taken the parcel away. The post service could have made a video of calling the door, but then maybe the video is fake – we kind of rely on videos being a good impartial witness here, which seems to have no good analogy for networking. In conclusion, it seems close to impossible to provide "hard evidence" of attempted delivery.
- The delivery service delivered the parcel, but the user wants to claim back the money. The user says that the parcel was not delivered and was stolen, and that the user never had it in their hands. Unless there is a witness (person, or video) of you receiving the parcel and actively acknowledging it, this seems hard to avoid. So, again, no "hard evidence" without interaction seems possible, and in the absence of it user can claim their version of the story.
- The delivery service did deliver the parcel, but it was actually stolen, so the user did not receive it. This is similar to the (2) case, and kind of indistinguishable from it.

In summary: even if the blockchain contains some data commitment, it seems hard (or likely impossible) to prove that the actual data inside this commitment was not communicated because of the sender or the receiver.

The most direct and simple solution is to have some temporary witnessing-VID layer, as we illustrate in Section 2.3. Parties receive the data in a sharded way, but do not store it for longer than like T minutes (several minutes), which is how fast we expect the server to reply.