Project Untitled Whitepaper [DRAFT]

o1Labs

February 5, 2025

Abstract. Web3 has a state management problem. Existing protocols struggle to effectively work with large amounts of state data due to the inherent costs of communication and replication. Established protocols have driven advancements in retrievability, data availability (DA), and secure replication, but largely focus on immutable state. While these state-of-the-art protocols can provide basic data retrievability, they still fall short of delivering fast mutability and permissionless control of data, often operating in big chunks of data creating barriers for developers and preventing many storage-oriented applications from being run effectively.

We present Project Untitled, a decentralized state management protocol that uses proofs of retrievable commitments, or storage proofs, to provide a way for decentralized applications on Mina and other Web3 ecosystems to access global, mutable state. Our approach is more federated than randomly sharded like with many DA schemes: for every allocated storage, a client and storage provider (SP) have to negotiate the per-agreement storage terms. This allows us to achieve faster throughput while relying on concrete SPs without sacrificing availability, which can still be achieved by storing the data with multiple SPs. Mutability is a key feature of Project Untitled: the protocol allows fast data updates, thus making incentives more fair by only asking users to pay for what they use.

1 Introduction

Storage in blockchain environments suffers from the choose-two trilemma of decentralized, inexpensive, and high-capacity. With many big web3 ecosystems we mostly see this as high fees that users and app developers encounter while trying to interact with any non-trivial storage within a smart contract. With Project Untitled, we introduce a new blockchain for data storage that offers all three requirements, with the following additional unique properties:

- *Mutable:* Data is mutable in-place and the history of the data is not stored unless required. This makes it suitable for use as a practical application state for smart contracts.
- *Ephemeral:* Data is stored for as long as it is needed and no longer. Storage providers are not stuck holding forever onto data that will never be used again.
- *Locally Query-able and Homomorphically Updateable:* Data 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.

Our solution is decentralised, but in a purposefully chosen, federated way. Instead of relying on an anonymous quorum or committee to store the data in a sharded mechanism, we create a platform for two types of agents (users and storage providers) to agree on the particular storage solution. With Project Untitled, 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. Additionally, we enforce data retrievability by requiring SPs to prove they have access to the data even if it has not been queried recently.

Some existing and proposed blockchain storage solutions (e.g. IPFS and Filecoin, or DA solutions like Ethereum's danksharding or Espresso's Tiramisu [Bea+24]) seek to offer other properties that Untitled, by design, does not aim to achieve. These include:

- *Immutability and Indefinite Persistence:* Experience tells us that real-life applications do not normally require such properties and can be replaced with the weaker notion of the data being "provably correctly derived", which Untitled offers instead. Specifically, while Untitled does not offer a full history of the state, it periodically proves that the entire current state of the state is still available and has not been truncated or otherwise corrupted.

– *Replication:* 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 for Untitled. Our aim is to focus on basic retrievability, and since our consensus protocol does not rely on proofs of space, we do not consider this a requirement.
– *Native Data Availability:* Data availability can be ensured in a cryptographic layer on top of Untitled, without having to be built into the protocol (see Appendix B). This 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. We actively consider a possibility where multiple SPs can store the same data, however, we rely on reputational factors to increase the guarantee of the data being replicated (and parties not colluding), as opposed to cryptographic ones. Our incentive calculations consider the possibility of collusion.

Project Untitled crucially relies on proofs of reads, updates, and retrievability, which are all implemented using the toolkit developed by o1Labs. Our current design suggests that custom, minimal plonk-like proofs, almost entirely reduced to the opening of polynomial commitment scheme (PCS), are fast and are easily integrated with our data storage model that relies on PCSs too. Additionally, we are actively considering using our folding library, arrabiatta, for aggregating both block proofs, and individual batching transactions, if it is necessary performance-wise.

From a consensus perspective, we adopt Algorand's Proof of Stake model, which has a great advantage in our scenario that offers lightweight, relatively standardized transactions, but requires fast finality for ease of external integration. Finally, Project Untitled will be a succinct chain, similarly to Mina, which has an additional desirable property of simplifying building bridges between Untitled and other chains.

We envision Project Untitled to be a standalone, separate, and succinct chain. Our first integration target is Mina, but we are building Project Untitled to be ecosystem agnostic. The 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 to and decrease the costs necessary to maintain the target smart contracts.

[Danny: todo: Introduce how an app works reg. state management. Takes as an example the lifecycle of a simple app on Ethereum or on Mina (initialization, reads, writes, finalisation - if required). It introduces the different operations a dev faces. From there, we introduce the different inefficiency culprits. With the introduction of these issues, we can start by adding our intro reg. commitments, state retrievability, etc. It will also be good to base our comparisons to have a real rough idea of data published on chains. Compare also with proto-danksharding and the blob data introduced in Cancun. Can also Project Untitled be used for the rollup-centric approached of Ethereum? Can it be used by the L2?]

[Danny: todo: Describe the network communication cost for the property of retrievability.]

2 Proofs of Storage and Reads

[**AL**: Note inconsistency with the protocol section where "write" is used for the initial write, and "update" for any write that comes after that.]

This section defines the core proving techniques used in Project Untitled - proofs of operations (reading and writing), and proofs of storage.

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.

Misha: How exactly is homomorphic property of PCS helps us to achieve quick granular writes? Can we prove it's faster than some funny broad d-merkle tree?.. verkle tree?

Matt: To answer Misha's above question: a key property that Untitled brings to the table is data ephemerality: data is *not* stored forever in a merkle tree like in QMDB. To prove it's faster, one need only notice that in order to perform and update to d bytes of data, one requires only $O(d)$ updates to field elements, which I think is clearly the fastest possible. You couldn't have, for example $O(\log d)$, updates of field elements, because that would imply a dependence between the data you're updating

Misha: But there might be a concretely more efficient quasilinear solution in $O(d \log d)$ with other properties being better? Like lighter proofs?

2.1 Cryptographic Preliminaries

Misha: todo: Introduce minimal crypto notation, groups, generators.

Project Untitled relies on 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:

- $\mathsf{IFFT}(\{D_i\}_{i=1}^{n}) \to F(X)$.
  - 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 $w. (x, w) \in \mathcal{R}$.

2.2 Basic Protocol of Commitment Retrievability

The core proving layer of the Project Untitled protocol must support the following functionalities: storing data, issuing a receipt on storage (a commitment), updating the data at a point, proving that a read at certain positions is equal to certain values, and proving retrievability – that the whole dataset is still present on the node's machine and can be accessed at any time.

Misha: Do we need a proof of update? **Matthew**: yes, similar protocol to reads [**AL**: The current design does not have a proof of update, it relies on the next storage proof to "confirm" the update]

The interface and semantics of the construction is as follows:

- $\mathsf{Setup}(\lambda)$
  - Defines what do we need to have / generate to run the protocol.
- $\mathsf{CommitData}(\{D_i\}_{i=1}^{m}) \to \{C_i\}_{i=1}^{m}$
  - What a node runs when it receives data. Essentially, it creates a commitment. Can be run on the client too.
- $\mathsf{UpdateCom}(\{C_i\}_{i=1}^{m}, \{S_i\}_{i=1}^{m}, \{\boldsymbol{v}_i\}_{i=1}^{m}) \to \{C'_i\}_{i=1}^{m}$
  - Perform a batch update of commitments, such that when C'_i should commit to D_i for indices j when $S_{i,j} = 0$, and otherwise to v_j.
  - We also give a trivial data update mechanism for completeness: $\mathsf{UpdateData}(\{D_i\}_{i=1}^{m}, \{S_i\}_{i=1}^{m}, \{\boldsymbol{v}_i\}_{i=1}^{m}) \to \{C'_i\}_{i=1}^{m}$ [**AL**: $\{D'_i\}_{i=1}^{m}$?]
    - ∗ **Misha**: In the future it might be more involved if we use some kind of data packing
- $\mathsf{ProveRead}(D, C, \{S_i\}_{i=1}^{k}) \xrightarrow{\$} (\pi, \{C_{\mathsf{v},i}\}_{i=1}^{k})$ **Anais**: not just one D, C but bunch

    - Given a single dataset column D and its commitment C, proves batch knowledge of data D_i on positions defined by selectors S_i (batching k queries). Returns a proof and commitments to the selected data (some commitments may be trivial and dropped in the implementation).
    - **Misha**: TODO: make it batcheble w.r.t. many commitments
- $\mathsf{VerifyRead}(C, \{S_i\}_{i=1}^k, \{\boldsymbol{v}_i\}_{i=1}^k, \pi) \to 0/1$
    - Verifies proof π attesting to the fact that $\forall i \in [k], j \in [d]. C \cdot S_{i,j} = \boldsymbol{v}_{i,j}$ w.r.t. the verifier-sampled (or publicly sampled) randomness. Selectors S_i and vectors $\boldsymbol{v}_i$ can be represented in scalar form and thus be concretely small.
- $\mathsf{ProveRetr}(\{D_i\}_{i=1}^m, \{C_i\}_{i=1}^m, \alpha) \xrightarrow{\$} \pi$
    - Proves that the node has access has access to the data D committed in C. Calling ProveRead with "full selectors" will achieve the same functionality, but we separate ProveRetr as its concrete implementation might be different.
- $\mathsf{VerifyRetr}(\{C_i\}_{i=1}^m, \pi, \alpha) \to 0/1$
    - Verifies the storage proof.

The full protocol is presented in Fig. 1 and Fig. 2.

Anais: Write (for first time) is a kind of an update? Or is that a Store? Have a think about language for consistency. Why Store is not called ProveWrite? Retr vs Fetch? etc.

```
Setup($\lambda$)
    % $|\mathbb{G}| = p$ prime, $\mathbb{H} < \mathbb{G}$
    % $|\mathbb{H}| = \mathrm{ord}\,\omega = 2^i$ for $i \geq \log d_{\mathsf{max}}$
    $\mathbb{G}, \mathbb{H} = \langle\omega\rangle \leftarrow \mathsf{GroupGen}(1^\lambda)$
    $\{G_i\}_{i=1}^d \leftarrow \mathsf{PCS.Setup}(\mathbb{G}, 1^\lambda)$
    $\{\mathcal{L}_i\}_{i=1}^d \leftarrow \mathsf{GenLagrangeBases}(\mathbb{G}, \omega, d)$
    return $(\mathbb{G}, \mathbb{H}, \sigma, \{\mathcal{L}_i\}_{i=1}^d)$

CommitData($\{D_i\}_{i=1}^m$)
    for $i \in [m]$ do
        Let $F(X) \leftarrow \prod \mathcal{L}_i^{D_i}$
        $C_i \leftarrow \mathsf{PCS.Commit}(F(X))$
    return $\{C_i\}_{i=1}^m$

UpdateCom($\{C_i\}_{i=1}^m, \{S_i\}_{i=1}^m, \{\boldsymbol{v}_i\}_{i=1}^m$)
    % Run by both user and the server
    for $i \in [m], j \in S_i$ do
        $C_{i,j} \leftarrow C_{i,j} \cdot \mathcal{L}_i^{v_{i,j}}$
    return $\{C_i\}_{i=1}^m$

UpdateData($\{D_i\}_{i=1}^m, \{S_i\}_{i=1}^m, \{\boldsymbol{v}_i\}_{i=1}^m$)
    % Run by the server only
    for $i \in [m], j \in S_i$ do
        $D_{i,j} \leftarrow D_{i,j} + v_{i,j}$
    return $\{D_i\}_{i=1}^m$
```

Fig. 1. Project Untitled core commitment related proving protocols.

Workflow example. We describe a small example with a verifier V wanting to delegate its storage to an untrusted prover P. This has no binding relation to Project Untitled's mechanics and is primarily illustrating the semantics.

1. Init: V takes their data $\{D_i\}_{i=1}^m$, computes the commitment to its data $\{C_i\} \leftarrow \mathsf{CommitData}(\{D_i\})$, and sends the data to P, who also calls CommitData in the same way. Then V can delete its data, keeping only $\{C_i\}$.
2. Query data: V sends the selector S_i he wants to access to P, who calls ProveRead and responds with and a proof of evaluation π, commitments to the selected data $C_{\mathsf{v},i}$, and (potentially) the data itself. We do not model direct data retrieval in the cryptographic part of the protocol. V verifies it using VerifyRead.
3. Check retrievability: V sends a random α, P finds the data D_i and commitments C_i corresponding to the V, and calls ProveRetr, returning the proof π to V. V calls VerifyRetr on this proof and commitments C_i that V stores. In Project Untitled, the randomness α is created by the chain, and can be the same for everyone. **Misha**: Generating unbiased randomness with the chain is not trivial.
4. Update the data: V sends an update $(\{S_i\}_{i=1}^m, \{\boldsymbol{v}_i\}_{i=1}^m)$, then prover runs UpdateData on the data and UpdateCom on its commitment, and verifier runs only the UpdateCom on their side locally.

Anais: todo: update algorithm in practice: explain why updating commitment works, relate to IFFT in Store. Just some knowledge-sharing paragraphs. Develop the math.

```
$\mathsf{ProveRead}(D, C, \{S_i\}_{i=1}^k)$
    $\boldsymbol{v}_i \leftarrow S_i \cdot D$
    Squeeze $\beta \leftarrow \mathsf{H}(\dots)$
    Obtain polynomials $V_i(X), S_i(X), D(X)$ by interpolating the vectors
    $T(X) \leftarrow \sum \beta^i (D(X) \cdot S_i(X) - V_i(X))/Z_{\mathbb{H}}(X)$
    Create $\mathsf{Com}(V_i), \mathsf{Com}(S_i), \mathsf{Com}(D), \mathsf{Com}(T)$
        Note: $v$ and $S$ can be sparse: $C_{\mathsf{v},i} \leftarrow \prod_j \mathcal{L}_j^{v_{i,j}}$
    Absorb commitments
    Squeeze $z \xleftarrow{\$} \mathsf{H}(\dots)$
    Compute evaluations $V_i(z), S_i(z), D(z), T(z)$
    % Prove batch opening of commitments w.r.t. their evaluations on $z$
    $\pi \xleftarrow{\$} \mathsf{PCS.BatchProve}(\dots)$
    return $(\{C_{\mathsf{v},i}\}, \pi)$
```

```
$\mathsf{VerifyRead}_{\mathsf{pk}}(C, \{S_i\}_{i=1}^k, \{\boldsymbol{v}_i\}_{i=1}^k, \pi)$
    Squeeze $\beta \leftarrow \mathsf{H}(\dots)$
    Compute (sparse) commitments to $v_i$:
        $C_{\mathsf{v},i} \leftarrow \prod_j \mathcal{L}_j^{v_{i,j}}$
    Absorb commitments
    Squeeze $z \leftarrow \mathsf{H}(\dots)$
    assert $T(z)Z_{\mathbb{H}}(z) = \sum \beta^i (D(z) \cdot S_i(z) - V_i(z))$
    assert $b \leftarrow \mathsf{PCS.Verify}(\pi, \dots)$
    return true
```

```
$\mathsf{ProveRetr}(\{D_i\}_{i=1}^m, \{C_i\}_{i=1}^m, \alpha)$
    Squeeze $x \leftarrow \mathsf{H}(\dots)$
    Compute $\hat{C} \leftarrow \prod_{i=1}^m C_i^{\alpha^i}$
    Compute $\hat{D} \leftarrow \prod_{i=1}^m D_i^{\alpha^i}$ and corresponding $\hat{D}(X) = \sum \alpha^i D_i(X)$
    Compute $y \leftarrow \mathsf{Eval}(\hat{D}(x))$
    $\pi \xleftarrow{\$} \mathsf{PCS.Prove}(\dots)$        % Prove $\hat{D}(x) = y$
    return $(\hat{C}, \pi, y)$
```

```
$\mathsf{VerifyRetr}_{\mathsf{pk}}(\{C_i\}_{i=1}^m, \pi, y, \alpha)$
    Squeeze $x \leftarrow \mathsf{H}(\dots)$
    Reconstruct $\hat{C} \leftarrow \prod_{i=1}^m C_i^{\alpha^i}$
    % Check $\hat{D}(x) = y$ for $D$ committed in $\hat{C}$
    $b \leftarrow \mathsf{PCS.Verify}(\pi, \hat{C}, x, y)$
    return $b$
```

Fig. 2. Project Untitled core commitment related proving protocols.

2.3 Commitments and Data Format

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, $\{G_i\}$ are the pre-generated bases, and r is the commitment randomness (which we will omit in most cases). Now, due to us working over the subgroup $\mathbb{H}$, instead we will have to commit to interpolations of our data over this group. That is, when $\{D_i\}_{i=1}^d$ 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 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 for homomorphically changing the values in 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 $D_i' = D_i + v$.

2.4 Proving Reads

For $\mathsf{ProveRead}$ we use standard hyperplonk-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 $V_i = S_i \cdot D$ 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

$$\prod_{i=1}^{k} D(X) \prod \beta^i S_i(X) - \prod_{i=1}^{k} \beta^i V_i(X) \equiv 0 \mod \mathbb{H}$$

Which is equivalent to $\forall i \in [k], j \in [d].\, D_j \cdot S_{i,j} = V_{i_j}$ under the assumption that β is indeed randomly sampled.

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

Delegating verification to the chain. While we describe the verifier as a single algorithm, it can be split into two sections. The $\mathsf{VerifyRead}$ procedure can be run if the verifier has access to $C_{\mathsf{v},i}$, and thus if the chain has access to the value commitment it can do this verification. As a second step, when the proof π is confirmed to be valid with respect to a set of value commitments $C_{\mathsf{v},i}$ and the corresponding selectors, the user owning the data $\{v_i\}$ can then reconstruct the commitment. Due to the commitment binding property, this will convince the user of the validity of the instruction, assuming that the user has observed the first phase happening on the chain.

This optimisaiton can be useful since (1) verifying the proof is more computation intense, while (2) communicating the data is more bandwith intense. By splitting the verification in two phases, we keep the computation on-chain (as it will be absorbed into the recursive pickles-style proof anyway), while preventing sending big amounts of payload data to all the nodes.

Misha: TODO: if we want to hide data from the server, we have to come up with a smarter way to do update transactions without attaching the payload to the transaction directly. This is hard cause you need to make sure data was sent and received off chain.

Sending less openings. The prover can send only evaluations of $T(z)$ and $D(z)$, and then the IPA proof will only prove only these two openings w.r.t. commitments $\mathsf{Com}(T)$ and C. On the verifier side, after checking the IPA proof, one can still check $T(z)Z_{\mathbb{H}}(z) = \sum \beta^i(D(z) \cdot S_i(z) - V_i(z))$ since given z together with selectors and return values in plain text, the verifier is capable of reconstructing evaluations $S_i(z)$ and $V_i(z)$, as well as $Z_{\mathbb{H}}(z)$, themselves.

Misha: TODO validate the validity of this claim

Batching read proofs over data chunks. The structure of $\mathsf{ProveRead}$ uses single D and C right now, but in practice the data might be stored in multiple commitments, and we would like the operation to be batched over multiple $\{C_i\}$.

Misha: TODO write out which equations we need to satisfy.

Proving Retrievability As of the storage proofs, the approach is similar and even more straightforward. Our task is to prove knowledge of the whole dataset behind the commitment – this is strictly more general than the proof of read, and can be implemented as a proof of read with full identity selectors, but instead we provide an even simpler protocol. The verifier provides a random α, [**Marc**: This random alpha can be fixed as the hash of all the C., The eval point x is what needs to be sent] and the prover has to show the opening to the commitment $\hat{C} = \prod \alpha^i C_i$, that internally contains a linear combination $\sum \alpha^i D_i$. Proving the opening of the commitment can be easily done with a simple IPA evaluation proof at a random point x, that is $\hat{C}(x) = y$.

The rationale behind soundness of the argument is as follows:

- A malicious prover would not be able to provide an evaluation of a degree d polynomial without knowing all its coefficients, and malicious prover cannot prove the opening of any other polynomial than committed in any individual C_i due to binding.
- When recombining multiple commitments with a fresh randomness α, the prover has to know all the individual openings for every commitment. **Misha**: Can I reduce it to smth? dlog embedded into RO?

– The replay attack would not be possible due to non-malleability of IPA commitments – an adversary seeing an opening of $\alpha^i C_i$ on point x still cannot produce y without reconstructing the polynomial. And if after polynomially many queries $\mathcal{A}$ is able to obtain d evaluations of some $D_i(X)$ behind $C_i(X)$, then $\mathcal{A}$ "knows" the vector D_i.

2.5 Security Guarantees

The security properties of our proof systems are standard NIZK ones plus retrievability:

- Correctness(es) (excluding correctness of PCS):
  - Running ProveRead and ProveRetr on valid data leads to verifiable stuff.
  - If the prover has access to the whole data, ProveRetr will not abort.
- Soundness of read proofs: if the read proof verifies, then the data in the commitment indeed has that value at that index.
- Retrievability: if the storage proof verifies, then the prover has access to at least $(1-\varepsilon)|D|$ part of the data where D is the dataset we refer to. In our trivial case, we aim to have $\varepsilon = 0$ but as storage proofs will become bigger, we might think about other techniques.

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.

We expect the PCS scheme to be complete, sound, and commitments to be binding. **Misha**: Which other properties do we need? Do we need knowledge extraction?

Anais: Notes about privacy of data

2.6 Practical Considerations

Misha: TODO: Trade-offs for which chains / proof-systems can be supported in this choice, why this curve and not the other, etc.

3 Protocol Key Terms

We define the following terms as part of the vocabulary necessary to describe the main protocol.

Account A typed entry in a ledger that contains a balance. It is required for token accounting in the protocol.

State Replicator A node responsible for storing data in the network. [**Marc**: I liked the term storage provider. The term replicator implies that the sate is replicated, which is not true in general.] **Misha**: plus

Balance The amount of native tokens in an account.

Commitment A constant size representation that binds to a piece of data. It is used to produce and verify cryptographic proofs.

Collateral The native tokens provided by a state replicator as a guarantee to safely handle network data.

Chunk The smallest unit of data for the network. It maps to a fixed number of bytes configured by the protocol.

Exchange Rate The price per chunk for a certain operation. It is expressed as nano-native tokens per chunk [1].

Storage Contract An account type that holds the collateral submitted by a state replicator. Each storage contract is scoped to a single commitment. After it is created, the state replicator is responsible for serving requests for the data.

Contract Requester The address of the account that initiated contract creation.

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

Contract Acceptor The address of the state replicator that accepted the storage contract and submitted collateral required for contract creation.

Terms The conditions agreed upon by the contract requester and contract acceptor for storage. They contain exchange rates, deadlines, and other relevant metadata.

Write A process that stores and allocates data within a state replicator.

Read A process that queries chunks of data stored by a state replicator.

Update A process that edits data that has already been allocated by a state replicator.

Epoch A fixed number of blocks that unlock all locked payments collateral. All write requests submitted during an epoch are only valid during this epoch.

4 Network Overview

Node operators in the project untitled 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.

Similar 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 state replicators communicate asynchronously. Every request transaction (see Transaction Type Quick Reference) 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 a minimal bandwidth. Any violations to the deadlines established by the protocol will cause a slash on the violator's collateral.

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

5 Ledger Construction

Ledger construction will mirror Mina's with the exception of the read ledger (see Table 2). 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 state replicators responsible for processing the request — will be sequenced in the block. **Matthew**: TODO: not all requests need to go through the read ledger, direct p2p communication between a co-operating state node and read requester will be more efficient, and we can use this as a fallback.

Table 1. Node Types, Roles and Revenue Streams

Type	Roles	Revenue Stream
State Replicators	– Storing data – Producing and submitting storage proofs – Handling update and write requests – Answering read requests – Monitoring the network	– Participant fees for each request – API Credit
Validators	– Participating in proof of stake: block production and block validation – Purchasing snark work. – Including internal transactions and acknowledgments in blocks.	– Block Rewards – Acknowledgment Rewards – Slashing Rewards – Transaction Fees
Snark Workers	– Producing and selling snark work for transactions included in blocks.	– Satisfied Snark Bids
Base Node	– Syncing to the network and submitting local transactions to the mempool. Both State replicators and Validators inherit all functionality from base nodes.	
Seed Node	– Serving a reliable list of peer addresses so new nodes joining the network can discover peers.	

Table 2. Ledger Types and Properties

Ledger name	Description	Properties
Staged Ledger	State of accounts for transactions that have been sequenced but not yet snarked	From Mina, deletable entries
Snarked Ledger	State of accounts for transactions that have been included in the staged ledger and snarked	From Mina, deletable entries
Staking Ledger	State of account staking used to represent the stake repartition	From Mina, deletable entries
Read Ledger	State of requests that need to be acknowledged or proven	Deletable entries

6 Account Types

Both state replicators and contract requesters will be represented as regular accounts in the ledger. However, state replicator terms will only be set for state replicators. This field serves as a bit to indicate when an account belongs to a state replicator and is a requirement for accounts submitting contract acknowledgement 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.

Table 3. Description of Accounts in the Staged and Snarked Ledger i.e. Merkle Leaf Structure

Type	Fields	Notes
Regular Account	– Public Key – Balance – State Replicator Terms – Locked Balance	– A regular account is a type for both requesters and state replicators. – The state replicator terms are only set for state replicators. – The locked balance is the amount of native token committed to requests. They are transferred to state replicators during a final acknowledgement. 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 – Expiration Block – Proof Deadline – Permission Bits	– The collateral balance will be the collateral submitted by the state replicator 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 state replicator; if not, the payment is returned to the requester. – Permission bits control the readers for the contract.

Negotiation in Project Untitled 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 state replicator. 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 at which the contract is valid until

Data size Size of the data that needs to be stored in unit of chunks

Exchange rates for reads and updates

Storage proofs heartbeat Number of blocks to increment the proof deadline when a storage proof is submitted.

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 acknowledgment.

There should be two preconditions for transactions with terms. The first is that the proof deadline block cannot be higher than the expiration block, which ensures at least one storage proof is provided per contract. The second is the termination compensation cannot be higher than the demanded collateral amount. This will ensure each termination is intended when a state replicator can no longer host data. A state replicator should not find termination equally or less profitable than getting slashed.

The storage contract account type will be added to the ledger once a state replicator agrees on the terms proposed by the requester. Once a final acknowledgement is submitted by the acceptor, the storage contract account will hold the collateral of the replicator. Two deadline fields will be used to track both the termination block of the contract, and the block at which the next storage proof should be submitted by. The requester and acceptor will also be included in the account for easy reference.

The structures for these account types is described in Table 3.

7 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 large writes, small updates, and small reads. While designing this version, the following requirements were kept:

7.1 Requirements and Reference

- Storage contract terms are negotiated between the original writer and state replicator
- Anyone should be able to read without creating a contract as long as they pay the exchange rate accepted in the terms for reads
- Updates to the contract commitment may only be done by the original writer
- Commitment maps contract to data, along with the corresponding terms that were negotiated
- Storage contracts are deletable entries in the staged and snarked ledger
- Permission bit only applies to the read

A quick reference guide to each transaction type is provided in Table 4.

7.2 Transaction Processes

Fig. 3. Key to interpret communication diagrams

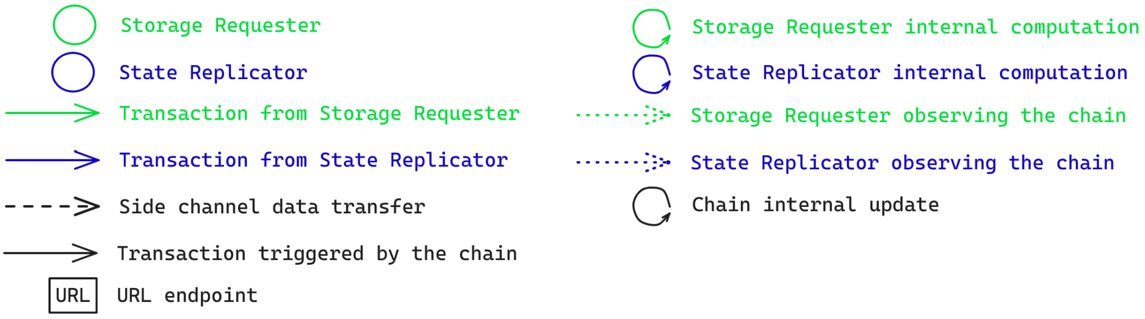
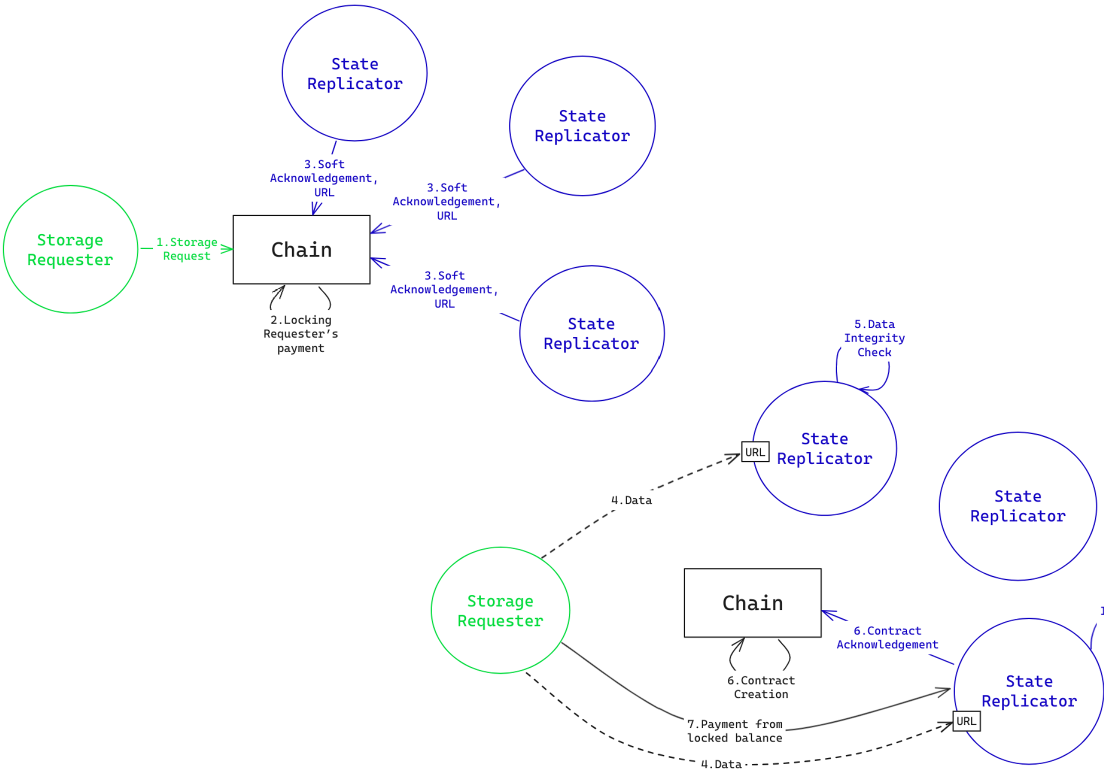

Storage Requester
State Replicator
Transaction from Storage Requester
Transaction from State Replicator
Side channel data transfer
Transaction triggered by the chain
URL
URL endpoint
Storage Requester internal computation
State Replicator internal computation
Storage Requester observing the chain
State Replicator observing the chain
Chain internal update

Fig. 4. Diagram for Write Request Flow and Storage Contract Creation



Write

Side Channels Before each step of the process is described, it is important to be clear on the notion of side channels in Project Untitled. In the case of a write request the write requester must send the data to the state replicator off-chain. Logistically, this requires the state replicator to have an externally facing service that accepts data. Furthermore, data sent by the requester through the side channel should include a signed commitment for the data intended to be written. This is for state replicators to verify the sender, and check if the data maps to the pending request. Every soft acknowledgement will come with a URL sent by the acceptor, so the storage requester knows where to send data that they intend to write. There will be a standard specification for message format sent over side channel, but that is a design that is outside the scope of this overview. **Matthew**: Side channels can also be set up in the p2p layer, which standardises the interface more.

Contract Creation A request will be sent by an application to establish a contract for a new piece. A storage provider should allocate data for this request. The required components are terms, a fee for the initial write and the commitment for that 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, state replicators may gossip a soft acknowledgement that declares an intent to handle the data with a URL. This allows for multiple state replicators to acknowledge the request. When the requester sees a soft acknowledgement, they must broadcast the data that they intend to write to the state replicator 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 state replicator will store and submit a final acknowledgement transaction that creates the storage contract, and transfers the collateral from the state replicator balance into the contract balance. Locked balance will also be unlocked and transferred to the contract acceptor. The first acceptor to submit a final acknowledgment with a sufficient collateral balance will be the state replicator responsible for the contract.

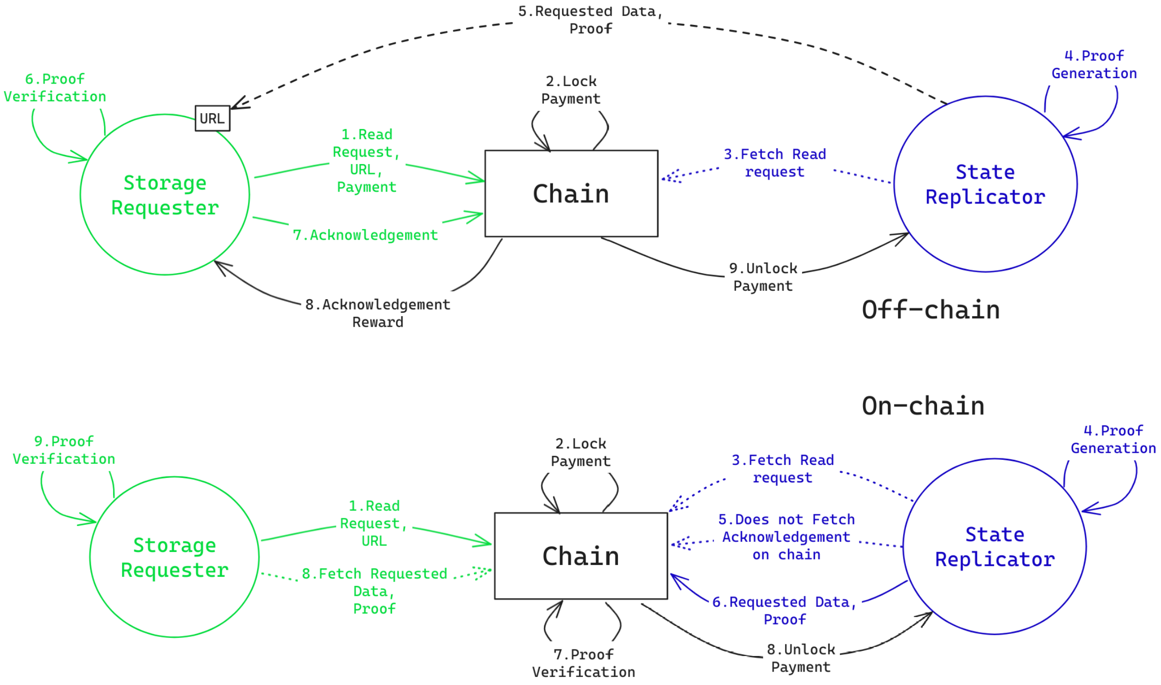
If a final acknowledgement is submitted before a soft acknowledgement the acceptor may end up slashed since they won't have the data for the storage proof. While this design requires some overhead from both parties to monitor transactions, it was one of the few approaches that allowed for large writes, small updates, and small reads. The different cases it addresses are described in Table 5.

Each party i.e. the requester and acknowledger require on and off chain services. In the diagram above each requester runs a service that forwards write data to the state replicator side channel. Slashing occurs in this design for all misbehaviours on and off chain. This is because the commitment should match the original data, and fail the storage proof if this is not the case. In short this design offers the following guarantees:

- Incentive to correctly price requests due to request fee locking
- As several state replicators can answer the same contract at the same time, Requester has more guarantee that its contract will be finalized. If a state replicator drops the contract creation, another candidate can replace it.
- If the Requester decides to trick the Acceptor by not sending the data, it will not harm the Acceptor as it will just not send the final acknowledgement, and thus not be committed on data it does not have.

Reads A read can be made to a storage contract once it has been created via final acknowledgement. Anyone can send a read transaction, which requests a limited number of chunks for a specific commitment. Since a storage contract already exists, the read request payment can be stored in its account. If a read request is not acknowledged within the read deadline, the request payment will be refunded and the stored collateral will be slashed. In contrast with writes, the side channelling is not required but recommended. The downside for not side channelling is the data read is posted on chain when the read proof is submitted. If the requester does not want to deal with side channel acknowledgements and does not want the data to be publicly available on chain, it needs to encrypt before computing the commitment and sending it to the state replicator; this way it can only be interpreted by the original writer. Many different data encoding schemes may be applied when writing data that are based on the application's use case. The validity of responses and acknowledgements for read requests is equivalent to the validity of the underlying proofs, which claim the data read binds to the previously known commitment.

Fig. 5. Diagram for Read Request Flow



Another significant detail to consider is batched read proofs. It will always be computationally cheaper to submit a batched read proof for several reads. Due to this fact it will be more economical to submit off-chain proofs, since proofs sent over channels can be bigger than the ones posted on chain.**Matthew**: Update for batch read protocol: proofs aggregating all reads are now of constant size, appropriate to always send on-chain. Individual proofs add overhead that we'd like to avoid. [**AL**: The issue is that data is needed for the proof to be verified. So even if the "proof" itself is constant size, the data needs to be carried along.] Batch proofs posted on chain will have to fit in blocks. Any reader that receives a batch proof should be able to verify the data they requested was read by the state replicator. The side channel serves as an ideal best case, and the chain proofs serve as a worst case.

Read Flow

- It is made to a commitment contract and can be done by anyone that pays in accordance with the terms of negotiation.
- The read proof is submitted by the replicator within the preconfigured protocol delta.
- Within this delta the reader and replicator can establish a side channel to send the data. The requester then submits an acknowledgement transaction on chain that satisfies the requirement for read proof.
- If the side channel is not established or if the requester does not send an acknowledgement, the replicator should submit the read proof on the chain, which includes the read data within the negotiated window.

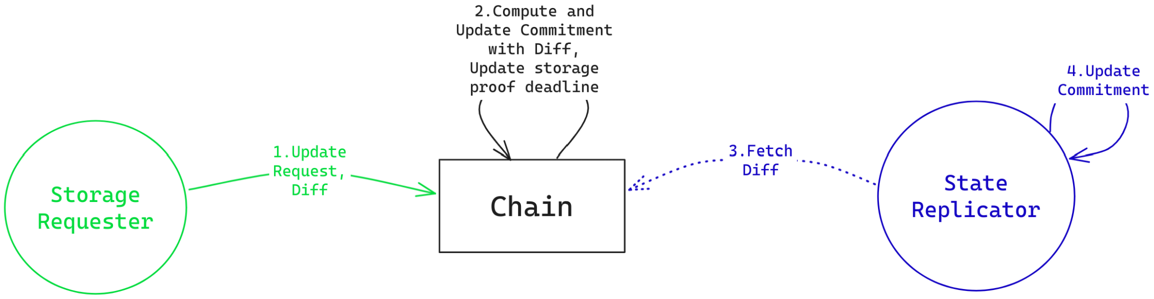
Guarantees

- The side-channel is private, faster and lighter than the public on-chain system, which is slower and heavier but is completely trustless.
- The Requester, whether via side-channel or on-chain, has the guarantee that the read data is valid thanks to the proof provided by the state replicator.
- If the Requester decides to not send acknowledgement, the state replicator can send its proof on chain to answer the request.
- The Requester is incentivized to give acknowledgement to be rewarded.
- The state replicator is incentivized to use the side channel, as avoiding to make read public may constrain other willing readers to also send a request to see the data.

Updates An update is made to the storage contract in charge of the data. Only the storage contract requester has the authority to make this request. The requester computes the diff needed to update the data and commitment, and then sends an update request to the chain. 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 state replicator should monitor the network, update the storage, and recompute the commitment. No response is expected from the state replicator, since the next storage proof will accurately acknowledge the request.

It is imperative that state replicators process 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. In the same spirit, any storage proof regarding an outdated commitment is invalid. In order to not overload the state replicator with update requests, each update request extends the deadline for the storage proof. As a result, the state replicator can't be slashed for getting an update extremely close to the storage proof deadline.**Matthew**: TODO: reconsider in the context of batched read/write proofs. We can also punt the write to the next challenge window.

Fig. 6. Diagram for Update Request Flow



Guarantees

- The commitment is updated independently by all parties (Requester, chain, state replicator) with the provided diff, which guarantees the correspondence of the data and commitment for everyone, and keeps the chain as the main source of trust: it will be the responsibility of the Requester to send the right diff, and the responsibility of the state replicator to update with the right diff.
- There is no proof submitted for this operation, but following reads / proof requests will expect proofs matching this new commitment. If the state replicator did not correctly update the data, it will be unable to provide these future proofs.
- Any proof that is asked before the update is included on-chain, refers to the old commitment. Any proof that is asked after the update inclusion, refers to the new one. Thus the chain always keeps a consistent chronology, and this is the responsibility of the state replicator to follow the chain's ordering.
- Each update extends the storage proof deadline, so the state replicators has enough time to craft a storage proof for the new commitment.

8 Contract Lifecycle

After the contract is created, it is valid up until a future block proposed by the requester and agreed upon by the state replicator. The end block can be extended by a "renew transaction" sent by the Requester that initiated the contract, which is either acknowledged by the state replicator, or a termination request is sent by the replicator 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 Requester or Acceptor can send a termination order, with a compensation specified in the contract. This deletes the contract, unfreezes the collateral and sends the compensation to the opposite party.
- At any time, the State replicator 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 the compensation is 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 State replicator accepts, it sends an acknowledgement. The contract is updated, and, if needed, collateral is adjusted.
  - If the State replicator refuses, the contract remains as it is.

9 Storage Proof Lifecycle

It is required by the protocol for state replicators to submit proofs of storage at a certain frequency. The contract is initiated with a block number that represents a deadline for the next required storage proof. When the storage proof is added the deadline is extended depending on the terms negotiated upon during contract creation

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

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.

Guarantees

- Network validates internal transactions with the ledger
- Block producers may not submit slashing transactions for contracts 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

10 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.

11 Consensus

We are currently looking to use Algorand's BFT mechanism mainly for the following reasons:

- The latest block certified is also finalized and therefore low latency.
- There is no slashing. [**Deepthi**: We need to carefully examine the participation requirements Algorand has and how to deal with offline nodes or stake delegations impacting quorum due to stake getting accumulated in a few validators]
- Long fork attack prevention is possible in the succinct setup

Since Project Untitled is a succinct blockchain, we need to check that consensus was executed correctly in the SNARK i.e., verifying a blockchain proof guarantees the following:

- Previous block hash matches the public input and the chain height increments by 1
- VRF evaluation of the block proposer was done correctly and is below a certain threshold set for leader election
- Verifies recursive proof of a certain ledger state ensuring all the transactions since genesis corresponding to the ledger state were applied according to the protocol

To be defined:

- Block selection rules including tie break when there are multiple blocks proposed.

11.1 Preventing long fork attacks in a succinct chain

One of the important considerations when designing a succinct chain is to be able to protect against long fork attacks without having to look up the history because nodes do not need history to validate the chain.

Long fork attacks or grinding attacks in PoS are a type of attack in which a malicious validator tries to manipulate the block producer selection process to increase the chance of being selected to produce blocks.

In PoS systems, block producers are typically selected based on:

- Their stake amount
- A random seed/value
- Some time-based parameters

The attack is as follows:

1. The attacker looks at multiple possible combinations of block content, timestamps and other parameters that influence the random seed
2. They then calculate which combination would give them the highest probability of being selected as the next block producer or for future blocks
3. By "grinding" through these different combinations, they try to find the most advantageous setup that maximizes their chances of selection

Algorand uses the previous block's randomness to determine the randomness for the next block's VRF evaluation (this applies even if the agreed-upon block is empty for a round and there is no leader).

So for malicious nodes, there are only two options to determine the randomness of the current round:

- Hash(Signature_of_the_leader(randomness_at_previous_block), round_number), if the block is not the default empty block.
- Hash(randomness_at_previous_block, round_number) otherwise

Neither of these depend on block data, making grinding attacks impossible through different block hash attempts. While there's a possibility of influencing randomness by splitting stake to introduce more proposers, the randomness depends on the leaders' signatures. The ledger used for selection is k blocks prior to preventing such manipulation. The argument is that adversaries cannot manipulate randomness to their advantage, as sufficient entropy is created by various honest block proposers in the last k rounds.

11.2 Long fork attack prevention

Long fork attacks involve hiding or not propagating blocks until a sufficiently long chain is generated by accumulating more stake over time. This attack is prevented by examining the history and verifying that each block in the fork has achieved quorum. Hidden blocks wouldn't have achieved quorum unless an honest majority of stake had voted on it, making this generally impossible to exploit in Algorand. However, since we want a succinct chain, we can't look into the history. We need to encode the chain quality in the latest block so that nodes can deterministically select the honest chain.

Two potential solutions:

1. As part of block proof verify all certificates of the previous block and assert that total certified stake is >= honest majority of stake. This ensures a block can be proven only if the previous block achieved quorum and therefore impossible to construct a valid looking fork in isolation. This solution may be [prohibitively] expensive because the number of certificates that need to be verified per block could be in hundreds or even close to 1000 depending on the number of verifiers per round.
2. Encode chain density for a window of slots or rounds, i.e., total number of non-empty blocks per window of rounds, similar to Mina. This computation alone may not be sufficient because Algorand guarantees correctness over liveness - meaning an empty block could be certified by an honest majority if Byzantine nodes successfully interfere during voting. However, since VRF is computed for each round and step, the probability of such interference occurring frequently seems small. To be determined if this is negligible enough that we can encode chain density as a monotonically decreasing rolling density of blocks per window of rounds and use that to assess chain quality. The concept

is that at the beginning of a long fork, the number of blocks will be proportional to the malicious stake, and as it progresses, more rounds are won to produce blocks regardless of whether blocks are attested. Therefore the chain density at the tip of a malicious fork will be less than that of the chain density of the honest chain. Chain density in Mina is described in Ouroboros Samasika paper and specified here.

12 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 state replicators, 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.

Payments [**Austin**: We're currently defining the protocol's economic structure and the incentives through a potential token with experts. More to come soon, below are largely placeholder sections]

Consensus delegation

12.1 Bucket abstraction

How we model the account state – what we need to store to represent the state commitments etc. Essentially: account format + deals interaction

12.2 Native token - economics

12.3 Incentives

12.4 Protocol guarantees

Implicitly: address attack vectors

13 Interacting with Project Untitled from zkApps on Mina

13.1 Interface

13.2 Bridging

14 Ecosystem Agnostic Interface

14.1 Bridging with EVM

14.2 Benchmarks and usability

14.3 Example of a chain-agnostic app

15 Competitive analysis / comparison with existing solutions

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.

[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`.

[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`.

Table 4. Transaction Type Quick Reference

Transaction Type	**Transaction**	**Description**
Simple Transfer	Simple Transfer of Native Token	Transferring native token from one account to another
Contract Handling	Write Request	Proposition by a network participant to create a contract account with a state replicator.
	Contract Renewal	Proposal made by the requester to extend the terms of the contract. This has to be unacknowledged by the contract state replicator.
	Contract Termination	Request made by the state replicator or requester. It frees up the collateral within the contract, but requires a fee depending on the terms of the contract.
	Termination agreement	Agreement from the requester to the state replicator's termination request
	Soft Acknowledgement (gossip)	State replicator sends this request to notify its agreement with the terms offered by the requester. This is a signal for the requester that the state replicator is available to establish a side-channel to receive the data.
	Contract Acknowledgement	Acknowledgement made by a state replicator which means that the state replicator knows the data and will store it according to the contract terms. Once this transaction is included, the state replicator is accountable for sending storage proof and answering relevant requests. This can be used to finalize a contract creation or a contract renewal.
Request	Read Request	Request made to a contract to read written data.
	Update Request	Request made to a contract to edit written data and replace the on-chain commitment.
Proof Submission	Storage proof	State replicator transaction that proves data is stored. Its inclusion updates the corresponding contract with the next storage proof deadline.
	Read proof	State replicator transaction that contains data stored at some indices and a proof of correctness of this data. State replicator is slashed if the proof is not served in the required time window.
Acknowledgement	Data Acknowledgement	Requester's acknowledgement that they received some read data. It saves the state replicator from having to submit a read proof within the proof window agreed upon in the contract.
Update Account Info	Update Account Info	Update information tied to the account like locked stake and state replicator terms. Currently the contract has fixed terms determined by negotiation.
Internal Operation	Storage Slashing	Transaction that exposes a contract whose storage proof request has not been fulfilled. It triggers the loss of the collateral for the state replicator that did not respect the storage proof terms.
	Read Slashing	Transaction that exposes a read request that has not been satisfied before the deadline. It triggers the loss of the collateral for the state replicator that did not respect the read proof terms.

Table 5. Properties of Writes and Contract Creation

Soft	R	A	SC	Data	Final	Outcome
Sent	Good	Good	Good	Data received & stored by Acceptor	Sent	Contract created
Sent	Good	Good	Bad	Data received does not match the soft acknowledged commitment	Not sent	No contract created, but another party can still send an acknowledgement.
Sent	Bad	Good	Good	Data received does not match the soft acknowledged commitment	Not sent	No contract created, but another party can still send an acknowledgement.
Sent	Bad	Bad	Good	Data received does not match the soft acknowledged commitment	Sent	Contract created. Acceptor will be slashed when a proof is requested.
Sent	Good	Bad	Good	Data received but not stored	Sent	Contract created. Acceptor will be slashed when a proof is requested.
Sent with incorrect commitment	Good	Bad	Good	No data sent, as soft acknowledgment's commitment is wrong	Not sent	No contract created, but another party can still send an acknowledgement.
Sent with an incorrect commitment	Good	Bad	Good	Requester does not send data to a soft acknowledgment that doesn't match its request	Not sent	No contract created, but another party can still send an acknowledgement, or Acceptor can resend a soft acknowledgment with the correct commitment.
Sent with an incorrect commitment that Requester knows the data for	Bad	Bad	Good	Data received are valid regarding the soft acknowledgment.	Sent for a wrong commitment	No contract created, as final acknowledgment is invalid (corresponds to a commitment that was not requested).

- Soft = Soft Acknowledgement
- Final = Final Acknowledgement
- R = Requester
- A = Final Acknowledgement
- SC = Side Channel
- Data = The handling of sending and receiving data
- Good = Honest
- Bad = Malicious

- A honest requester sends the correct data when a soft acknowledgement is sent for its request.
- A honest Acceptor sends soft acknowledgements for commitment that have been requested, check the data when it receives it and only send final acknowledgements when it stores the correct data.
- A honest Side Channel transfers the data to the Acceptor as it was delivered by the requester

Appendices

A Related work

1. Fisch: Proofs of Space on Useful Data
   - https://eprint.iacr.org/2018/678.pdf
2. Fisch: Tight Proofs of Space and Replication
   - https://eprint.iacr.org/2018/702.pdf
   - A replication and space proofs protocol used in the novel (2020?) filecoin. The core contribution is designing a memory heavy function, computing a graph, and arguing why this function is hard to parallelise and how much time/memory it takes.
   - ZigZag spec for filecoin:
     - https://github.com/Stebalien/filecoin-specs/blob/master/zigzag-porep.md
3. A Novel Approach to Proof-of-Replication via Polynomial Evaluation
   - https://eprint.iacr.org/2023/1569
   - They XOR the data with the result of the memory hard function (graph comp) on a replica tag, and then prove the evaluation of that polynomial on a random challenge.
4. stoRNA: stateless Transparent Proofs of Storage-time
   - https://eprint.iacr.org/2023/515.pdf
5. Data availability:
   - A16z summary for danksharding: https://a16zcrypto.com/posts/article/an-overview-of-danksharding-a
   - Also see:
   - Espresso sequencing network:
     - https://eprint.iacr.org/2024/1189.pdf
     - The sequencing network is built on top of a data availability (DA) level, see page 4 for details. Layer 1, Savoiardi, is a verifiable information dispersal (VID) protocol, in which a piece of data is split into chunks, encoded, and sent to many parties. To destroy data availability, the adversary would have to control 1/3 stake in the system. Retrieving data from Savoiardi takes a lot of communication complexity though. Layer 2, Mascarpone, fixes this by adding a small random committee feature for faster retrieval: the committee nodes have to store the whole block data, and so with high probability one can return the data.
   - ZODA: Zero-Overhead Data Availability
     - https://angeris.github.io/papers/da-construction.pdf
     - Rather a paper on data availability.
   - The Accidental Computer: Polynomial Commitments from Data Availability
     - Related (built on top of?) ZODA.
     - https://angeris.github.io/papers/accidental-computer.pdf
   - Vitalik: 2D Data Availability using Kate Commitments
     - https://ethresear.ch/t/2d-data-availability-with-kate-commitments/8081
     - TLDR: Assume the data is N blobs, each blob is M chunks, so imagine a table of height M and length M. Commit to each chunk (each row, as a vector) using Kate commitment. This gives you N commitments. Now, the claim is: it is possible to extrapolate each row from length N to length $2N$ using only commitments, homomorphically. So we can build $C_{N+1} \dots C_{2N}$ without having access to the witness. This gives us $C_1 \dots C_{2N}$ commitments for the table twice the length, with a lot of redundancy. Send the data for these commitmetns to random nodes, and it only requires N out of $2N$ nodes to restore every row in the table.
6. Authenticated Data Structures
   - QMDB: https://github.com/LayerZero-Labs/qmdb
     - It's just a fast authenticated data structure, no replication or data storage proofs.
   - Look for more ADSs?
7. Fair exchange.
   - Atomic and Fair Data Exchange via Blockchain: https://eprint.iacr.org/2024/418.pdf
     - This solves the problem of fair exchange when the parties want to interact, but does not solve the problem of parties not wanting to interact (fulfil their obligations).

        - The solution is on Figure 1: A sends encrypted data, A and B agree on the payment and key on-chain atomically, and then B can decrypt the data with the key.
    - `https://onlinelibrary.wiley.com/doi/pdf/10.1155/2021/5781354`
8. **TODO** is watermarking relevant?
9. **TODO** investigate PoW as a memory hard function.
10. Compact Proofs of Retrievability
    - `https://eprint.iacr.org/2008/073.pdf`
    - A foundational, highly cited 2008 paper providing very simple proofs of retrievability. Close to what we're doing naively with Project Untitled, but with BLS signatures instead of SNARKs.
11. Filecoin docs
    - PoRep: `https://filecoin.io/proof-of-replication.pdf`
    - Whitepaper: `https://filecoin.io/filecoin.pdf`

B WIP: Replication and Retrieval

Thoughts on replication problem in Project Untitled – what is it, why would we do it, how, and what is possible / what is no?

B.1 Types of data-related problems and concepts

There are quite a few properties on storing data in a decentralized manner one might want to achieve:

1. Quick authentication / state management: Assuming the node stores the data, how expensive is it to prove membership, update, or exclusion from it?
    - This is what Authenticated Data Structures (vector commitments being part of) achieve.
    - The problem is related to Project Untitled but not immediately:
        - The typical scenario this is interesting in is proving updates to the global authenticated state, such as mina. However, in Project Untitled, we might (or might not) want the whole state to be compacted to a short digest.
        - ADSs do not care about retrievability of the data in full. You can design a "forgetful" Merkle tree that delegates the non-often queried pieces to the other nodes.
    - Indeed, VC is an ADS, and we crucially rely on an ADS for membership proofs. But ADS solutions are quite often tailored to querying parts of the data structure, while in our case we *also* want to achieve retrievability (see next bullet), that is making sure that the whole
    - What we want at the very least is an ADS that supports (1) fast queries and updates, but also (2) full data retrievability.
2. Retrievability: Can I (effectively) retrieve the data that is supposedly stored by the server?
    - Proving retrievability (PoR) implies that the interactive protocol allows the client to retrieve the data after polynomial number of interactions.
    - Proving Retrievability from a Commitment (PoRC) is a specialisation of PoR where the receipt that the user gets from the server is a compressing commitment.
3. Data Availability: Given a piece of data that is sharded across many nodes (in a lossy and aversarial nework), can I make sure I can ultimately retrieve it by running some queries against the nodes?
    - Proving Data Possesion (PoD, eprint) is a subtype of DA, and it only promises you that the server has your data, without necessarily enforcing retrievability. Think of a scenario where a big archive wants to prove it still has the data and it's not been damaged, but we don't want to retrieve all this data simultaneously at any point.
    - Otherwise, DA commonly implies many nodes, and aims to store the data in a way smarter than RAID1, where each user's storage costs are lower than $|D|$ (while the total network costs may be more than $n \cdot |D|$ for n number of users).
    - Data Availability does not necessarily imply fast retrievability: some solutions would split the data into small blocks and send their encodings to many nodes in the network, thus requiring a lot of querying to piece together the original file.
4. Proof of Space: Given a piece of data D, can we be sure that the server is actually using at least $(1-\epsilon)|D|$ memory to store this data?

  - Note that even though one can often compress the data and store it using less space, unless it has really low entropy the actual space being used is still close to $|D|$. At the same time, *proving* that one uses at least that much memory is not a trivial task.
  - I am not sure we care about proofs of space in Project Untitled.
5. Replication: Can we be sure that the data is properly replicated, that is two parties (potentially colluding) can prove they store the data separately, and not using each other's storage?
  - Replication is a hybrid of retrievability (PoR) and proof of storage (PoS).
  - Implies DA transitively since it's a RAID1 basically.
  - Regarding PoRep $\implies$ PoS: Is it in the definition or in the solution? Most PoReps rely on a memory-hard functions that essentially force the prover to do a certain computation using as much memory as the data.

B.2 The problem of replication

Replication is a property of a proof-of-space protocol that allows requesting multiple parties to store the same piece of data, and being sure that this data is actually stored as several independent copies. This might be desirable for e.g. storage reliability – the permissionless nature of blockchains means that any node can opt out of the network at any time, and replication ensures that the damage caused by this is minimised.

Achieving sound replication in a distributed setting is non-trivial. Some attacks, listed in filecoin's PoRep paper, include:

1. Sybil attack.
  - An adversary $\mathcal{A}$ creates N sybils and instead of storing N replicas (copies) of the same file, it stores just one. On request to retrieve the data, a node controlled by $\mathcal{A}$ will query the only node that stores the data, and proxy-relay that file.
2. Outsourcing attack.
  - A subtype of the previous attack. Even when there is no coordination between the parties, if querying the data is available to anyone, $\mathcal{A}$ can pretend to store "another copy of the data", while in fact never storing anything, and using existing data source.
  - If we're using the data on smart contracts, then access to the data should be public, therefore this attack is possible.
3. Generation attack.
  - If the data is low-entropy and easy to re-generate, adversary might not store anything at all, and re-generate the data on request. E.g. if the data is a vector $\{H(m,i)\}_i$ for a small m and H being the hash function, the adversary needs to store only m.
  - The generation attack is not really a concern for us, but many PoRep schemes are secure against it anyway.

What we're ultimately worried about is (1) and (2).

B.3 Does Project Untitled need replication?

Questions to discuss:

1. Is there another incentive mechanism to prevent parties from throwing out data essential for the protocol? Slashing raises the costs for abrupt exit.
  - Even with slashing, how do you prove that the data has been "transferred" to another provider?
  - Clients have to reupload their data then?
  - The node itself will declare "exiting" and suggest re-selling its rights to hold the data to someone else?
  - Note that although this might work similarly in practice (because people love their money and are mostly benign?), the security guaranteed by the replication protocol is higher: the cost of dropping a certain file would have to include the cost of collusion among potentially unknown parties.
2. Is there any product value in guaranteeing that the data is replicated?
  - Maybe users want the data to be "more safe".

  - Maybe we can argue that with replication we can distribute same piece of data over many geographical regions, thus making queries faster? Without PoRep it would be *hard* to incentivise nodes to do it.
  - Reputation / choosing holders separately.
  - Do we even need replication? Maybe we need data availability?

3. What's the main technical innovation of Project Untitled?
   - Right now Project Untitled's design seems quite straightforward and is 95% engineering and very little cutting edge innovation. Maybe it's a good thing.

B.4 Solutions and problems

How replication is solved in many cases: the server has to prove that they store a certain "sealed" (in the terminology of filecoin) replica of this data. That is, not the data itself, but some kind of expensive-to-compute transformation on the data (but reversible). Filecoin's seal function was originally a symmetric encryption/decryption with an encoding key (derived from the storing party and replica id).

In the next approach of Filecoin, called ZigZag, the slow to compute and memory intense function is a specific computation along a specfic type of Depth Robust Graph. Imagine a complicated layered graph each node of which is a hash of its incoming node values. The source nodes are just $H(\mathsf{seed} \| i)$. Then (for all the nodes or only on the last layer) we do $c_i = d_i \oplus e_i$ where $e_i = \mathsf{Enc}(\mathsf{PrevLayerParent}(e_i) \| k_i)$ (Enc is either VDF or identity), and finally $k_i = H(\mathsf{SameLevelParents}(i))$. In the end we get c_i which are replica-encodingns of d_i (data) according to some graph. All hashes have an extra see.d

Another solution is PoRep via polynomial evaluation (2024, Ateniese et al.). This paper still uses memory-hard functions such as graph computations, but most of its focus is not on this functionality. It constructs the replica encoding as a xor of a certain hash of this hard computation *and* the data, and then interprets the result as a polynomial. The verification query is an evaluation of this polynomial. This all sounds quite relevant to what we're doing, and I think it is, except that the memory-heavy part is explicitly assumed to be trusted, while in real-life the prover node would have to prove this computation is done correctly. It is not clear whether proving this is feasible.

Cool research problem: proof of replication with concretely cheap homomorphic updates to the values.

C Ensuring replication

Replication is a desirable property to have, as it is a simple method have confidence that one's data is secure against storage providers dropping one's data, either accidentally or maliciously. Happily, this is not a property that needs to be built into Untitled, but can be layered on top of it, allowing the user of the protocol to make the necessary design decisions and trade-offs required for their particular application.

We present here two ways of ensuring replication of data.

The first is a cryptographic method that is suitable when your app is only going to be used by certain known trusted parties and – barring leaking of the secret key used in it – ensures replication by storage providers. As a bonus, it also provides a level of data privacy, while still maintaining the desirable "updateable" nature of Untitled. The storage provider would have to break the encryption in order to store the data just once, providing a cryptographic guarantee that this will not happen.

The second is an economic method that is suitable when you need a decentralized application where the data in question does not have a clear fixed set of trusted "owners" who manipulate it, and instead is publicly mutable (presumably according to certain rules enforced in a smart contract). It does not *ensure* replication like the first method, but instead makes it more expensive for a storage provider to not replicate your data than replicate it, and so a rational storage provider will do so.

Centralized Cryptographic Replication The basic idea is to use public-key cryptography to encrypt the data on a per-field element basis. This can be done while preserving local updateability, but requires anyone who updates or reads the data to have access to the private and public keys, and also requires the storage providers to *not* have access to the private keys.

Matt: unsure how to phrase – as pseudocode or mathematically?

Decentralized Economic Replication The goal here is to instead make it more expensive for the storage provider to *not* replicate the data while still responding to random challenges, than it is to just replicate it. This involves a trade-off however, as the readers/writers of the data must now also pay a computational cost when accessing the data. The idea is to use symmetric encryption with static, randomly-chosen keys to mix some noise into the data. Crucially, updates will be sent in plaintext, acting upon mixed data, but not being mixed themselves. This means that it is easy to apply an update to mixed data (it is just a homomorphic addition), but updating unmixed data requires one to first mix the data, then apply the update, then re-unmix the data for storage again. By tuning the economic cost of mixing/unmixing, one can convince a rational storage provider against attempting to store the data once only, and unmixed. All readers/writers of the data will have to perform this mixing and unmixing dance, however, in order to perform an update – effectively assuming that the application is decentralized enough that this cost will be distributed over the large number of users.