(WIP) Dilithium: A Proof-of-Archival-Storage Consensus Protocol for Subspace

Chen Feng[1,2], Dariia Porechna[1], Barak Shani[1,3], and Jeremiah Wagstaff[1]
[1]Subspace Labs
[2]University of British Columbia
[3]Matter Labs

***Abstract*—Dilithium is a new proof-of-archival-storage consensus protocol designed to provide superior user experience while maintaining the highest level of consensus security among existing protocols. Dilithium combines KZG polynomial commitment, erasure coding, and function inverting in a unique way to achieve unprecedented efficiency. In this paper, we present the fundamental construction of Dilithium and outline its security proofs. We also present our initial implementation results to demonstrate the protocol's practicality. Our early experimental evaluations show that Dilithium can significantly enhance the user experience while maintaining strong security guarantees, making it a promising candidate for practical deployment.**

I. Introduction

Proof-of-capacity[1] has emerged as an intriguing alternative to proof-of-work leader-election mechanisms. Unlike proof-of-work (which relies on energy-intensive computation), proof-of-capacity leverages disk space as the underlying resource, thereby minimizing energy consumption. One of the unique advantages of proof-of-capacity over other blockchain leader-election mechanisms, such as proof-of-stake, is that it leverages an external and widely distributed resource. As a result, proof-of-capacity is more egalitarian, since it enables anyone to participate in the consensus process, regardless of their computational power or wealth. In addition, proof-of-capacity avoids certain design complexities associated with proof-of-stake [3], making it a compelling alternative for blockchain design.

Broadly, proof-of-capacity protocols can be divided into two categories

- Proof-of-(useless)-space: Prior constructions include Spacemint [12] (which is based on "graph pebbling" [2], [6]), Chia [5] (which is based on inverting "random" functions [1], [6]), Spacemesh [4] (which is based on incompressible proof-of-work [11]). These constructions fill disk space with cryptographic data - chunks of bytes that are reserved exclusively for proving the ownership of data.
- Proof-of-(useful)-storage: As a particular example, Filecoin [13] relies on a proof-of-replication mechanism [7] to store several copies of a file. Their construction is also based on graph-pebbling games, moreover for efficiency they also rely on zkSNARKs. As another example, Subspace whitepaper [14] proposes to use disk space to store the blockchain history. The original construction is based on a computation-bound function called SLOTH.

For further exploration of related works in this area, please refer to Section 1.2 of the Spacemint paper [12] and Section 1 of the proof-of-replication paper [7].

Despite a large variety of existing protocols, we believe that an ideal proof-of-capacity protocol should not only store useful data (in particular blockchain history) but also provide a superior user experience while maintaining the highest level of consensus security among existing protocols. In terms of user experience, the following performance metrics are most relevant

- Initialization (or setup) time: a few hours to days (depending on the amount of disk space)
- Proof-generation time: less than one second
- Proof size: hundreds of Bytes
- Verification time: much less than one second, ideally in the order of milliseconds

However, these goals cannot be achieved with prior constructions including the original Subpsace protocol. This has motivated us to design a new protocol that combines three ingredients, namely, KZG polynomial commitment [8], erasure coding [10], and function inverting [1] in a unique way in order to address the above design challenges.

II. Building Blocks of Dilithium

This section provides an in-depth review of the fundamental building blocks of Dilithium. We present how Dilithium combines KZG polynomial commitment, erasure coding, and function inverting through a novel concept called *storage coins* to achieve efficiency and security. We discuss the technical details of each building block and explain how they work together to form the overall protocol design.

A. Creating Storage Coins

We divide a file F into n pieces $\{d_0, d_1, \ldots, d_{n-1}\}$, each of equal size[2]. Without loss of generality, we view each piece as a row vector of length ℓ over $\mathbb{Z}_p$ (i.e., $d_i \in \mathbb{Z}_p^\ell$)[3]. Then, F

The authors are listed in alphabetical order.

[1]We use proof-of-capacity as an abstract umbrella term, capturing proof-of-space, proof-of-storage, proof-of-replication, proof-of-space-time, proof-of-retrievability, and so on.

[2]We can view F as the blockchain history. In general, F grows over time. Here, we only consider the case that F is fixed and defer the general case to our protocol specification.

[3]Here, we choose $\mathbb{Z}_p$ because we would like to apply KZG polynomial commitment later.

can be viewed as a matrix of size $n \times \ell$ over $\mathbb{Z}_p$. Alternatively, each piece d_i can be viewed as a polynomial $f_i(x)$ over $\mathbb{Z}_p$ of degree at most $\ell - 1$. This allows us to view F as a collection of n polynomials $\{f_i(x)\}_{i=0}^{n-1}$.

We assume that each farmer creates a farmer ID $\texttt{id}$ uniformly at random over some domain. For example, $\texttt{id}$ can be a (cryptographic) hash output of a farmer's public key $\texttt{pk}$ (i.e., $\texttt{id} = H(\texttt{pk})$). We also assume that each farmer selects m pieces (out of n pieces) in a random yet *verifiable* way based on its farmer ID. We denote these m pieces (or equivalently, polynomials) by $\{g_0^{\texttt{id}}(x), \ldots, g_{m-1}^{\texttt{id}}(x)\}$. For example, each $g_i^{\texttt{id}}(x)$ can be selected as $f_{(\texttt{id}+i)\%n}(x)$, where $\%n$ is the mod n operation[4]. As another example[5], each $g_i^{\texttt{id}}(x)$ can be selected as $f_{H(\texttt{id},i)\%n}(x)$.

We are now ready to explain how a farmer creates storage coins. Let

$$F^{\texttt{id}}(x) = \begin{bmatrix} g_0^{\texttt{id}}(x) \\ g_1^{\texttt{id}}(x) \\ \vdots \\ g_{m-1}^{\texttt{id}}(x) \end{bmatrix}$$

be a column vector of m polynomials. Then, we can evaluate $F^{\texttt{id}}(x)$ at the following ℓ points: $\texttt{id}$, $\texttt{id}+1$, ..., $\texttt{id}+\ell-1$[6]. Each evaluation $F^{\texttt{id}}(\texttt{id}+j)$ is a column vector of length m over $\mathbb{Z}_p$. We call it a **storage coin** associated with $\texttt{id}$ at point $\texttt{id}+j$. For the purpose of efficiency, a farmer can evaluate ℓ points simultaneously for every polynomial $g_i^{\texttt{id}}(x)$ by using some fast polynomial-evaluation algorithms with $\mathcal{O}(\ell \log^2(\ell))$ complexity. Alternatively, we can evaluate $F^{\texttt{id}}(x)$ at another set of ℓ points: $\texttt{id}$, $\texttt{id}\cdot\omega$, ..., $\texttt{id}\cdot\omega^{\ell-1}$, where ω is a primitive root of unity over $\mathbb{Z}_p$ with $\omega^\ell = 1$ or $\omega^{2\ell} = 1$.

Why is a storage coin useful? If we collect all ℓ storage coins associated with $\texttt{id}$, we can recover $F^{\texttt{id}}(x)$ through polynomial interpolation. In particular, if we collect all ℓ evaluation points for some polynomial $g_i^{\texttt{id}}(x)$, we can recover $g_i^{\texttt{id}}(x)$ through polynomial interpolation. (There are some fast algorithms for polynomial interpolation with $\mathcal{O}(\ell \log^2(\ell))$ complexity.) In other words, storage coins allow us to recover some pieces of the file F efficiently. Hence, they represent **useful storage** for F.

How can a farmer prove to some verifier that a storage coin $F^{\texttt{id}}(\texttt{id}+j)$ at point $\texttt{id}+j$ is created correctly? This is a standard polynomial commitment problem. For example, we can use the KZG commitment scheme. Let A_i be the KZG commitment of $f_i(x)$ for $i \in \{0, 1, \ldots, n-1\}$. We consider the following two cases.

1) The verifier knows $\{A_i\}_{i=0}^{n-1}$. In this case, the farmer just needs to create a KZG proof for the polynomial evaluation $g_i^{\texttt{id}}(x)$ at point $\texttt{id}+j$, because the verifier can check whether the choice of polynomial $g_i^{\texttt{id}}(x)$ is correct. For example, if $g_i^{\texttt{id}}(x)$ is selected as $f_{(\texttt{id}+i)\%n}(x)$, the verifier can check whether the KZG proof is consistent with $A_{(\texttt{id}+i)\%n}$.
2) The verifier doesn't know any A_i. In this case, we can create a KZG commitment T for $\{A_i\}_{i=0}^{n-1}$ and then give T to the verifier[7]. The farmer needs to provide two KZG proofs: one for the polynomial evaluation $g_i^{\texttt{id}}(x)$ at point $\texttt{id}+j$ and the other for the correctness of $g_i^{\texttt{id}}(x)$. (That is, $g_i^{\texttt{id}}(x)$ is chosen correctly according to our rule. For example, the commitment of $g_i^{\texttt{id}}(x)$ is indeed $A_{(\texttt{id}+i)\%n}$, which can be proven by using T.)

B. Creating Random Challenges

We start with a high-level overview. At each time slot (say, one second), every farmer observes some global randomness (also referred to as the global challenge). Based on this global challenge, each farmer determines exactly one storage coin (out of ℓ storage coins) to scan. Since a storage coin contains m elements in $\mathbb{Z}_p$, this gives a farmer m "tickets" to win the lottery. A ticket is called a winning ticket if it is "close enough" to the global challenge (in terms of the Hamming distance or some other distance).

More formally, let $\mathcal{C}_t$ be the global challenge at time slot t. Then, each farmer computes

$$H(\mathcal{C}_t, \texttt{id}) \bmod \ell$$

in order to decide which storage coin to scan. For example, if

$$H(\mathcal{C}_t, \texttt{id}) \bmod \ell = j,$$

then the farmer will scan $F^{\texttt{id}}(\texttt{id}+j)$, hoping to find a winning ticket (out of m tickets). Alternatively, if $\mathcal{C}_t$ and $\texttt{id}$ share the same domain, each farmer can simply compute

$$\mathcal{C}_t \oplus \texttt{id} \bmod \ell$$

to decide which storage coin to scan, since $\oplus$ operation is easier than $H(\cdot)$ operation. Once a farmer finds a winning ticket, it can produce a new block. Clearly, this leader-election mechanism is in spirit similar to proof-of-work (one CPU, one vote) and proof-of-stake (one coin, one vote).

In the above construction, anyone in the system can figure out whether a particular farmer has a winning ticket, because all the operations depend on the farmer's public key $\texttt{pk}$ (through $\texttt{id}$) but not on the secret key $\texttt{sk}$. The construction can be made privacy-preserving as follows. Each farmer creates a **local challenge** by signing the global challenge $\mathcal{C}_t$ with its secret key $\texttt{sk}$. Based on its local challenge, each farmer determines exactly one storage coin (in the same way as before) to find a winning ticket that is close enough to its local challenge. A farmer reveals its local challenge only after a winning ticket is found.

[4]By abuse of notation, we convert $\texttt{id}$ (from its domain) into an integer here.

[5]This example corresponds to a standard coupon collector's problem. In order to make sure that every piece is selected by some farmer, we need a total of $O\left(\frac{n \ln n}{m}\right)$ farmers.

[6]Again, by abuse of notation, we convert $\texttt{id}$ into an element in $\mathbb{Z}_p$.

[7]Since A_i is not an element in $\mathbb{Z}_p$, we cannot apply KZG directly. Instead, we need to hash A_i so that $H(A_i) \in \mathbb{Z}_p$.

C. Masking Storage Coins

The previous construction is susceptible to the so-called on-the-fly computing attacks. Specifically, an attacker can store only one copy of F and then "emulate" as many farmer IDs as possible by using its computation rather than storage resource. Essentially, a storage coin $F^{\mathtt{id}}(\mathtt{id}+j)$ is a collection of m polynomial evaluations at point $\mathtt{id}+j$, which can be computed on the fly.

In order to mitigate this attack, we propose to "mask" storage coins via function inverting. Consider a family of functions $\text{FUNC}_{\mathtt{seed}} : \mathcal{D}_k \to [2^k]$, where $\mathcal{D}_k$ is the domain[8] and $[2^k] = \{0, 1, \ldots, 2^k - 1\}$ is the co-domain. Note that both domain $\mathcal{D}_k$ and co-domain $[2^k]$ depend on the parameter k.

For simplicity, we assume that $\text{FUNC}_{\mathtt{seed}}(\cdot)$ is surjective in this section (which will be relaxed in later sections). This allows us to define a "right inverse" $\text{MASK}_{\mathtt{seed}} : [2^k] \to \mathcal{D}_k$ of $\text{FUNC}_{\mathtt{seed}}$ such that for any $\mathtt{index} \in [2^k]$, we have

$$\text{FUNC}_{\mathtt{seed}}\left(\text{MASK}_{\mathtt{seed}}(\mathtt{index})\right) = \mathtt{index}.$$

Essentially, $\text{MASK}_{\mathtt{seed}}(\mathtt{index})$ returns a pre-image of $\mathtt{index}$ under $\text{FUNC}_{\mathtt{seed}}$.

Next, we explain how to mask a storage coin

$$F^{\mathtt{id}}(\mathtt{id}+j) = \begin{bmatrix} g_0^{\mathtt{id}}(\mathtt{id}+j) \\ g_1^{\mathtt{id}}(\mathtt{id}+j) \\ \vdots \\ g_{m-1}^{\mathtt{id}}(\mathtt{id}+j) \end{bmatrix}.$$

We denote by $\mathtt{commit}\left(g_i^{\mathtt{id}}(x)\right)$ the KZG commitment of $g_i^{\mathtt{id}}(x)$. For each $g_i^{\mathtt{id}}(x)$, we set

$$\mathtt{seed} = \mathtt{id} \oplus H\left(\mathtt{commit}\left(g_i^{\mathtt{id}}(x)\right)\right).$$

Then, for any $j \in [2^k]$, we compute $g_i^{\mathtt{id}}(\mathtt{id}+j) \oplus H\left(\text{MASK}_{\mathtt{seed}}(j)\right)$ as a masked version of $g_i^{\mathtt{id}}(\mathtt{id}+j)$, denoted by $\tilde{g}_i^{\mathtt{id}}(\mathtt{id}+j)$. Applying this procedure to all $\{g_i^{\mathtt{id}}(\mathtt{id}+j)\}_{i=0}^{m-1}$, we obtain a **masked storage coin**. As we will soon see, for the purpose of efficiency, a farmer can mask ℓ points for a polynomial $g_i^{\mathtt{id}}(x)$ and then move to another polynomial, say $g_{i+1}^{\mathtt{id}}(x)$. Alternatively, we can use $H\left(\text{MASK}_{\mathtt{seed}}(j)\right)$ to mask $g_i^{\mathtt{id}}(\mathtt{id}\cdot\omega^j)$ instead of $g_i^{\mathtt{id}}(\mathtt{id}+j)$, where ω is a primitive root of unity[9].

Why is the masking function $\text{MASK}_{\mathtt{seed}}(\cdot)$ useful? First, the attacker has to invert m functions on-the-fly in order to emulate a masked storage coin. This requires excessive space-time resources, as we will see later. Second, the attacker cannot reuse its space-time resources to emulate different farmer IDs. In particular, even if two farmer IDs have a same polynomial in common, the attacker still needs to invert two different functions (because of the difference in $\mathtt{seed}$).

D. Putting Everything Together

We are now ready to put all the building blocks together. Recall that the file F consists of n pieces $\{d_i\}_{i=0}^{n-1}$, which can be viewed as n polynomials $\{f_i(x)\}_{i=0}^{n-1}$. Recall that A_i is the KZG commitment of $f_i(x)$ and T is the KZG commitment of $(H(A_0), \ldots, H(A_{n-1}))$. We assume that T is public information. Let π_i be the KZG proof for $H(A_i)$. With π_i, anyone in the system can verify whether A_i is consistent with the public information T.

Each farmer generates a key pair $(\mathtt{sk}, \mathtt{pk})$ and derives its farmer ID $\mathtt{id}$ (e.g., $\mathtt{id} = H(\mathtt{pk})$). With a given $\mathtt{id}$, the farmer selects m polynomials $\{g_i^{\mathtt{id}}(x)\}_{i=0}^{m-1}$ and retrieves their KZG commitments $\{\mathtt{commit}\left(g_i^{\mathtt{id}}(x)\right\}_{i=0}^{m-1}$ together with the proofs with respect to T. Then, the farmer creates ℓ storage coins $\{F^{\mathtt{id}}(\mathtt{id}+j)\}_{j=0}^{\ell-1}$ and generates their masked versions by using the function $\text{MASK}_{\mathtt{seed}}(\cdot)$ as described before. Finally, the farmer stores ℓ masked storage coins as well as some metadata (i.e., m commitments $\{\mathtt{commit}\left(g_i^{\mathtt{id}}(x)\right\}_{i=0}^{m-1}$ together with their proofs). This process is often called **plotting** in the literature.

Recall that a global challenge $\mathcal{C}_t$ is generated at time slot t, based on which each farmer selects one masked storage coin and collects m lottery tickets from it. If a farmer finds a winning ticket, it has to prove the following

- the winning ticket (say, $\tilde{g}_i^{\mathtt{id}}(\mathtt{id}+j)$) is indeed close enough to $\mathcal{C}_t$
- the unmasked element $g_i^{\mathtt{id}}(\mathtt{id}+j)$ is correct

This process is called **farming** in the literature.

The above construction provides a new leader-election mechanism, which can be combined with some longest-chain protocol to produce a consensus algorithm. A diagram of the above construction will be provided at the end of this paper to better illustrate the interaction of different building blocks.

III. Extensions

Our basic construction assumes that $\text{FUNC}_{\mathtt{seed}}(\cdot)$ is surjective. In this section, we explain how to relax this assumption with erasure coding through two examples.

Example 1: Construct $\text{FUNC}_{\mathtt{seed}} : [2^k] \to [2^k]$ as

$$\text{FUNC}_{\mathtt{seed}}(x) = \text{HASH}(\mathtt{seed} \| x),$$

where $\|$ denotes concatenation and $\text{HASH} : [*] \to [2^k]$ is a cryptographic hash function. Then, under the random oracle model, for any given $\mathtt{seed}$, $\text{FUNC}_{\mathtt{seed}} : [2^k] \to [2^k]$ is a random function that maps an input in $[2^k]$ to an output uniformly chosen from $[2^k]$.

Let $\mathtt{index} \in [2^k]$. We have the following two cases.

1) $\mathtt{index}$ has at least one pre-image under $\text{FUNC}_{\mathtt{seed}}$. In this case, $\text{MASK}_{\mathtt{seed}}(\mathtt{index})$ is well defined and we simply mask $g_i^{\mathtt{id}}(\mathtt{id}+\mathtt{index})$ as we did before.
2) $\mathtt{index}$ has no pre-image under $\text{FUNC}_{\mathtt{seed}}$. In this case, $\text{MASK}_{\mathtt{seed}}(\mathtt{index})$ doesn't exist and we cannot mask $g_i^{\mathtt{id}}(\mathtt{id}+\mathtt{index})$, leavng it unmasked.

Then, for each storage coin $F^{\mathtt{id}}(\mathtt{id}+\mathtt{index})$ at point $\mathtt{id}+\mathtt{index}$, we only store its masked elements (and discard its unmasked elements).

Previously, we need a total of ℓ storage coins (in order to recover the original pieces). How many storage coins do we

[8]Two examples of $\mathcal{D}_k$ will be given in the next section.

[9]In the C-KZG library, ω is chosen such that $\omega^{2\ell} = 1$.

need now? Clearly, this number L should be between ℓ and 2^k (i.e., $\ell < L \leq 2^k$). In fact, this question is closely related to the balls-and-bins problem. Suppose that we have 2^k balls and 2^k bins with each ball placed into a bin uniformly at random and independent of each other. Then, the fraction of empty bins is approximately $\frac{1}{e}$ for large 2^k. In other words, the fraction of non-empty bins is approximately $1-\frac{1}{e}$. This implies that $\frac{\ell}{L} \geq 1-\frac{1}{e}$ if we would like to have at least ℓ non-empty bins out of the first L bins. In fact, the probability of an "error event" (of having fewer than ℓ non-empty bins out of the first L bins) can be approximately bounded by $\exp\left(-2^k D_{KL}(1-\ell/L, 1/e)\right)$. The larger L is, the larger the term $D_{KL}(1-\ell/L, 1/e)$ is, and the smaller the error probability. This can be used as a guideline for choosing L.

Example 2: Let $p : [2^k] \to [2^k]$ be a random permutation. Define the domain $\mathcal{D}_k \subset [2^k] \times [2^k]$ as the set

$$\{(x_1, x_2) : p(x_1) = \pi\left(p(x_2)\right)\},$$

where π can be any involution without a fixed point (e.g., flipping all the bits). Construct $\text{FUNC}_{\texttt{seed}} : \mathcal{D}_k \to [2^k]$ as

$$\text{FUNC}_{\texttt{seed}}(x_1, x_2) = \text{HASH}(\texttt{seed} \| x_1 \| x_2).$$

In Example 2, we can view any $(x_1, x_2) \in \mathcal{D}_k$ as a ball and so our analysis for Example 1 still applies here. Note that $\text{MASK}_{\texttt{seed}}(\cdot)$ here returns an element in $\mathcal{D}_k$ instead of $[2^k]$.

There are some interesting space-time lower bounds in the literature related to Examples 1 and 2.

Lower Bound 1: For any oracle-aided algorithm $\mathcal{A}$, it holds that for most random functions $\text{FUNC} : [2^k] \to [2^k]$, if $\mathcal{A}$ is given advice (that can arbitrarily depend on FUNC) of size S, makes at most T oracle queries and inverts FUNC on $\epsilon 2^k$ values, we have

$$S \cdot T \in \Omega\left(\epsilon 2^k\right). \tag{1}$$

Lower Bound 2: For any oracle-aided algorithm $\mathcal{A}$, it holds that for most functions $\text{FUNC} : \mathcal{D}_k \to [2^k]$ constructed in Example 2, if $\mathcal{A}$ is given advice (that can arbitrarily depend on FUNC) of size S, makes at most T oracle queries and inverts FUNC on $\epsilon 2^k$ values, we have

$$S^2 \cdot T \in \Omega\left(\epsilon^2 2^{2k}\right). \tag{2}$$

However, Lower Bound 2 only applies when $T \leq \left(2^k/4e\right)^{2/3}$. This restriction seems to be mostly related to the proof technique. Here, we conjecture that it still holds even when $T > \left(2^k/4e\right)^{2/3}$.

Let us compare these two lower bounds. For simplicity, we set $\epsilon = 1$ and $S = \sqrt{2^k}$. Then, Lower Bound 1 says that $T \in \Omega\left(\sqrt{2^k}\right)$ and Lower Bound 2 states that $T \in \Omega\left(2^k\right)$. That is, Lower Bound 2 requires significantly longer time T under the same amount of space S, because the function in Example 2 is more complicated than that in Example 1. Intuitively, we can achieve a better lower bound

$$S^q \cdot T \in \Omega\left(\epsilon^q 2^{qk}\right)$$

with $q \geq 3$ if we make the function FUNC even more complicated. A particular approach is given in [1], which is the key idea behind the Chia project.

Finally, recall that the number of storage coins L satisfies $\ell < L \leq 2^k$. Since we need to invert $\text{FUNC}(\cdot)$ on L values (for $\texttt{index} \in \{0, 1, \ldots, L-1\}$), we can rewrite the above lower bound in a simpler form

$$S^q \cdot T \in \Omega\left(L^q\right). \tag{3}$$

In the plotting phase, we essentially find pre-images of all L values and then XOR a pre-image of `index` (if it exists) with the corresponding polynomial evaluation at `id+index`. This corresponds to one extreme case of the space-time lower bound where we set $S = L$. On the other hand, on-the-fly computing corresponds to the other extreme case where we set $S = 1$ and obtain $T \in \Omega\left(L^q\right)$.

IV. Security Analysis

In this section, we conduct an initial security analysis based on the previous space-time lower bound. Our key observation is twofold. First, the previous lower bounds capture the amount of resources one needs to have in order to emulate storage coins. Second, our new construction can be analyzed by using the general framework proposed by Kiffer, Neu, Sridhar, Zohar, and Tse [9].

A. System Model

The protocol proceeds in slots of duration τ. For simplicity, we consider a fixed set of N farmers with equal capability. We are particularly interested in the large system regime $N \to \infty$. In each slot, each farmer can win the lottery with probability ρ/N, independently of other farmers and slots, where ρ is a system parameter. When a malicious farmer wins the lottery at slot t, it can create multiple blocks, which is a behavior called equivocations. Note that equivocations never happen in proof-of-work protocols (when $\tau \to 0$) but can appear in our construction as explained in [9]. A malicious farmer can emulate storage by launching Hellman attacks (as explained in [1]) or it can simply follow our proof-of-archival-storage construction but deviate from the longest-chain protocol (say, launching nothing-at-stake attacks). We assume that the fraction of malicious farmers is at most β. Here, β can be viewed as a security threshold indicating the maximum fraction of malicious farmers a system can tolerate.

B. Emulating Storage Coins

The previous lower bound (3) suggests the following. In order to emulate L storage coins of a farmer (i.e., emulating a farmer ID), one needs mS amount of space and makes at most mT oracle queries with $S^q \cdot T \in \Omega\left(L^q\right)$ (where m is the number of pieces a farmer stores as defined before). This allows us to compare our construction with Chia, since the lower bound (3) applies to both. Using our notation, the parameter setup in Chia can be described as $k = 32$, $L = 2^{32}$, $m = 1$, and $q = 7$. In our setup, we can choose $k = 22$, $L = 2^{22}$, $m = 2^{10}$, and $q = 10$ to conduct a fair

comparison, because the required disk space is proportional to mL. Under the above setups, on-the-fly computing requires at most T oracle queries to emulate a farmer ID in Chia with $T \in \Omega\left(2^{224}\right)$, while it requires at most $2^{10}T'$ oracle queries in our construction with $T' \in \Omega\left(2^{220}\right)$. Similarly, an attacker $\mathcal{A}$ with space S needs to make at most T oracle queries in order to emulate a farmer ID in Chia with $S^7 \cdot T \in \Omega\left(2^{224}\right)$, whereas an attacker $\mathcal{A}'$ with space $mS^{\frac{7}{10}}$ needs to make at most mT' oracle queries in our construction with $S^7 \cdot T' \in \Omega\left(2^{220}\right)$. This implies that our construction is comparable to Chia (under appropriate parameter selection) in terms of mitigating Hellman attacks for emulating storage.

Next, we would like to point out that our construction enjoys efficient plotting. With a relatively small k, all the plotting operations in our construction can be done in memory, thereby eliminating the so-called "sort on disk" operation—the bottleneck of the Chia plotting algorithm. This enables us to achieve significantly shorter initialization time than Chia.

Finally, we would like to comment on the memory access. It is well known that Hellman attacks require a large amount of memory access. How can we derive a tight lower bound on the memory access? We aim to answer this question for Hellman attacks as well as other similar attacks (e.g., the one based on rainbow tables) as our future work. We believe that a lower bound on memory access better captures the "hardness" of emulating storage in our setup.

C. Consensus Security

We notice that our proof-of-storage construction is compatible with the general framework proposed in [9][10]. This allows us to apply the Theorem 2 of [9] to our construction, obtaining the following security result. For all $\beta < \frac{1}{2}$ and system parameters ρ, τ satisfying the condition (16) in [9], our proof-of-storage longest-chain consensus protocol with equivocation removal is secure. By investigating condition (16), we notice that we can achieve a higher security threshold β than Chia due to our new lottery design and the use of equivocation removal. We will provide an in-depth discussion in a more elaborated version of this work.

D. Discussion

The analysis above shows that in order to participate in the consensus protocol, a farmer can choose between different resources, namely storage S or oracle queries T that capture the amount of computation needed in order to participate in the lottery. Note that with a correct choice of function FUNC, the latter also requires a substantial amount of memory access.

We now turn to discuss how we design the system such that rational farmers always prefer to allocate the storage resource to the system, rather than the computation resource or a combination of both. This not only ensures that the protocol does not degrade into proof-of-work, but also eases the analysis as it allows us to focus on a single resource, namely storage. The full version of this work will give a rigorous justification.

[10]Indeed, we have confirmed this with one of the authors of the paper.

First we reiterate that since the $\text{MASK}_{\texttt{seed}}(\cdot)$ function depends on the farmer ID, creating any 2 different plots requires running the function twice as many, thus expending the resources needed to run $\text{MASK}_{\texttt{seed}}(\cdot)$. Moreover, notice that the seed depends also on the particular position j in the column vector $F^{\texttt{id}}$ (that is, on $g_i^{\texttt{id}}(x)$), thus even for a fixed $\texttt{id}$, any change to the plot requires expending the resources needed to run $\text{MASK}_{\texttt{seed}}(\cdot)$.[11]

Secondly, recall that the function $\text{MASK}_{\texttt{seed}}$ is parameterized by a "difficulty" parameter k. We can therefore set k high enough such that creating plots "on the fly" is not profitable, compared to creating a single plot and storing it, which requires running $\text{MASK}_{\texttt{seed}}$ only once.

In more detail, we measure the cost of storing a single piece of (masked) information d_i in between challenges $\mathcal{C}_t$.[12][13] A single piece d_i gives some amount c (e.g. $c = 1$) of lottery tickets per challenge $\mathcal{C}_t$.

Then, we measure the cost of running $\text{MASK}_{\texttt{seed}}$ with parameter k once. Running $\text{MASK}_{\texttt{seed}}$ once enables to produce a single masked piece d_i.[14] This gives the farmer c lottery tickets per challenge $\mathcal{C}_t$ as well. Since we have control over k, we can make sure that the latter cost is significantly higher than the former, thus making protocol deviation an irrational strategy.

V. Initial Experimental Results

In this section, we present the initial Rust implementation results of our proposed construction. Our evaluations were obtained on common consumer hardware - Apple M1Pro 2021, 10 cores (8 performance and 2 efficiency) and 16 GB memory. We evaluate the performance of each component as well as the entire construction. Our early evaluation results are encouraging, and we believe that there is still room for further optimization.

For the KZG polynomial commitment, we currently derived public parameters locally for testing purposes only. We chose $\ell = 2^{15}$ so that the size of d_i is 1MiB. The following four operations are required to implement KZG polynomial commitment. The evaluation results are provided below.

[11]We explain that even though the presentation above assumes the m selected pieces d_i of F are stored in one plot, in practice we divide them into several *sectors*, each with its own *sector id*, and so a farmer could try many sector id's. The comment above explains that for each of these attempts, the farmer would have to pay the cost of running $\text{MASK}_{\texttt{seed}}$. The plotting protocol is still deterministic, as describes above, so one could not create 2 different sectors with the same farmer id and sector id.

[12]We do not discuss here the methods of cost measuring, but we highlight that one can think of the alternative cost a farmer with, say, 100GB has for some period of time, given the market for storage. We can then derive the corresponding values for the size of any d_i and the challenge slot duration. We remark that this cost should also take into account the operational cost, like electricity.

[13]We would like also to consider the cost of obtaining the storage device in first place, assuming the farmer did not have free space – the reason is to explain that the "attacker" may use their funds to buy storage. This should be a negligible cost in the long run, and we do not discuss it here. This is also the case for running $\text{MASK}_{\texttt{seed}}$ in the (honest) protocol.

[14]Here we assume that the dishonest farmer has the entire file F unmasked, hence she can draw any piece d_i, and assign it no extra cost (as this can be amortised among all of the attacker IDs).

- Converting d_i into $f_i(x)$ through inverse FFT $\approx 1.6ms$
- Computing the commitment A_i of $f_i(x) \approx 40ms$
- Creating a single KZG proof for $f_i(x) \approx 40ms$
- Verifying a single KZG proof $\approx 865\mu s$

We implemented the function-inverting component by using Chia PoS tables with $k = 20$, $L = 2^{16}$, and $q = 7$.[15] We converted their code from C++ to Rust and did various optimizations. We have the following evaluation results

- Generating 7 function tables $\approx 325ms$
- Masking 2^{15} points of a polynomial $\approx 7.3ms$
- Constructing a Chia proof-of-space $\approx 186ns$
- Verifying a Chia proof-of-space $\approx 25\mu s$

Our implementation achieved orders-of-magnitude improvement over Chia's C++ implementation. We will explain the reasons for this improvement in a more elaborated version of this work.

Finally, we report the evaluation results of the entire construction for leader election (without the longest-chain consensus).

- Plotting phase with $m = 1000 \approx 300s$
- Scanning a single storage coin $\approx 580\mu s$
- Recovering a polynomial $\approx 880ms$
- Proving a winning ticket $\approx 970ms$

The plotting phase shows consistent usage of 3 GB of memory, suggesting that we can further increase k and q. Scanning a storage coin is a simple operation, only involving reading the coin from disk. Recovering a polynomial is a complicated process, involving reading from disk all the storage coins that contain chunks of it, re-generating the Chia function tables, XOR-unmasking, and erasure code recovery. Proving a winning ticket consists of three steps: recovering the "winning" polynomial, computing a Chia proof-of-space, and creating a KZG proof. The first step is the bottleneck, and the other two steps are quite efficient.

ACKNOWLEDGEMENT

The authors would like to thank Nazar Mokrynskyi for his impressive Rust implementation of the proposed construction. The authors would also like to thank Shashank Agrawal and Joachim Neu for fruitful discussions throughout this project. CF would like to thank Khadijeh Bagheri, Hoang Dau, and Hassan Khodaiemehr for their useful comments on a draft version of this work.

REFERENCES

[1] Abusalah H., Alwen J., Cohen B., Khilko D., Pietrzak K., and Reyzin L. *Beyond Hellman's Time-Memory Trade-Offs with Applications to Proofs of Space* Advances in Cryptology – ASIACRYPT 2017, pp. 357–379.

[2] Ateniese, G., Bonacina, I., Faonio, A., and Galesi, N. *Proofs of Space: When Space Is of the Essence* SCN 2014: Security and Cryptography for Networks, pp. 538–557.

[3] Azouvi S., Cachin C., V. Le Duc, Vukolic M., and Zanolini L. *Modeling Resources in Permissionless Longest-chain Total-order Broadcast* arXiv:2211.12050 [cs.CR], https://doi.org/10.48550/arXiv.2211.12050.

[15] We will try other parameters (such as $k = 22$ and $q = 10$) as our next steps.

[4] Bentov I., Loss J., Moran T., and Shani B. *The Spacemesh Protocol: Tortoise and Hare Consensus… In… Space* https://spacemesh.io/blog/spacemesh-protocol-v1.0/, https://drive.google.com/file/d/18I9GPebWqgpvusI1kMnAB9nayBbL-1qN/view.

[5] Cohen B., and Pietrzak K. *The Chia Network Blockchain* https://docs.chia.net/green-paper-abstract/, https://docs.chia.net/assets/files/Precursor-ChiaGreenPaper-82cb50060c575f3f71444a4b7430fb9d.pdf.

[6] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak *Proofs of Space* Advances in Cryptology – CRYPTO 2015, pp. 585–605.

[7] Fisch B. *PoReps: Proofs of Space on Useful Data* Cryptology ePrint Archive, Paper 2018/678, 2018. [Online]. Available: https://eprint.iacr.org/2018/678.

[8] Kate A., Zaverucha G., and Goldberg I. *Polynomial commitments* Technical report, 2010. CACR 2010-10, Centre for Applied Cryptographic Research, University of Waterloo.

[9] Kiffer L., Neu J., Sridhar S., Zohar A., Tse D. *Security of Blockchains at Capacity* Cryptology ePrint Archive, Paper 2023/381, https://eprint.iacr.org/2023/381.

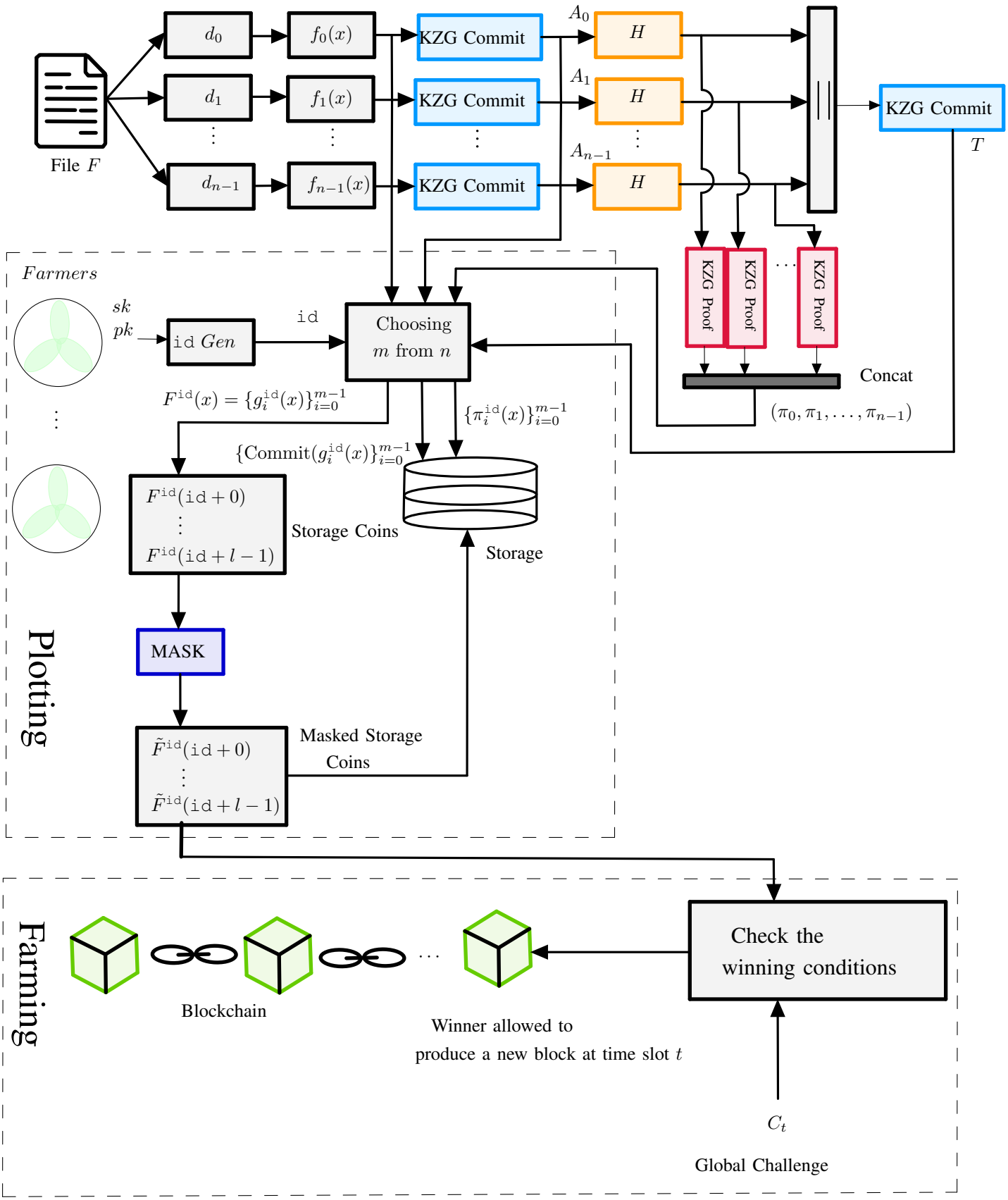
[10] Li J., and Li B. *Erasure coding for cloud storage systems: A survey* in Tsinghua Science and Technology, vol. 18, no. 3, pp. 259-272, June 2013.

[11] Moran T., and Orlov I. *Simple Proofs of Space-Time and Rational Proofs of Storage* Advances in Cryptology – CRYPTO 2019, pp. 381–409.

[12] Park S., Kwon A., Fuchsbauer G., Gaži P., Alwen J., and Pietrzak K. *SpaceMint: A Cryptocurrency Based on Proofs of Space* Cryptology ePrint Archive, Paper 2015/528, https://eprint.iacr.org/2015/528.

[13] Protocol Labs *Filecoin: A Decentralized Storage Network* https://filecoin.io/filecoin.pdf.

[14] Wagstaff J. *Subspace: A Solution to the Farmer's Dilemma* https://subspace.network/news/subspace-network-whitepaper,


File F
d_0
d_1
d_{n-1}
$f_0(x)$
$f_1(x)$
$f_{n-1}(x)$
KZG Commit
KZG Commit
KZG Commit
A_0
A_1
A_{n-1}
H
H
H
KZG Commit
T
KZG Proof
KZG Proof
KZG Proof
Concat
$(\pi_0, \pi_1, \ldots, \pi_{n-1})$
Farmers
sk
pk
id Gen
id
Choosing m from n
$F^{\mathrm{id}}(x) = \{g_i^{\mathrm{id}}(x)\}_{i=0}^{m-1}$
$\{\pi_i^{\mathrm{id}}(x)\}_{i=0}^{m-1}$
$\{\mathrm{Commit}(g_i^{\mathrm{id}}(x)\}_{i=0}^{m-1}$
$F^{\mathrm{id}}(\mathrm{id}+0)$
$F^{\mathrm{id}}(\mathrm{id}+l-1)$
Storage Coins
Storage
MASK
$\tilde{F}^{\mathrm{id}}(\mathrm{id}+0)$
$\tilde{F}^{\mathrm{id}}(\mathrm{id}+l-1)$
Masked Storage Coins
Plotting
Farming
Blockchain
Winner allowed to produce a new block at time slot t
Check the winning conditions
C_t
Global Challenge