Proofs of Space-Time and Rational Proofs of Storage

Work In Progress: Do Not Distribute

Anonymous Submission

Abstract. We introduce a new cryptographic primitive: Proofs of Space-Time (PoSTs) and construct a practical protocol for implementing these proofs. A PoST allows a prover to convince a verifier that she spent a "space-time" resource (storing data—space—over a period of time). Formally, we define the PoST resource as a trade-off between CPU work and space-time (under reasonable cost assumptions, a rational user will prefer to use the lower-cost space-time resource over CPU work).
Compared to a proof-of-work, a PoST requires less energy use, as the "difficulty" can be increased by extending the time period over which data is stored without increasing computation costs. Our definition is very similar to "Proofs of Space" [ePrint 2013/796, 2013/805] but, unlike the previous definitions, takes into account amortization attacks and storage duration. Moreover, our protocol uses a very different (and simpler) technique, making use of the fact that we explicitly allow a space-time tradeoff, and doesn't require any non-standard assumptions (beyond random oracles). As a consequence, our protocol supports a market-based adjustment of its parameters in a crypto-currency context, similar in spirit to the difficulty adjustment for PoW protocols.

1 Introduction
(Version Id)

A major problem in designing secure decentralized protocols for the internet is a lack of identity verification. It is often easy for an attacker to create many "fake" identities that cannot be distinguished from the real thing. Several strategies have been suggested for defending against such attacks (often referred to as "sybil attacks"); one of the most popular is to force users of the system to spend resources in order to participate. Creating multiple identities would require an attacker to spend a correspondingly larger amount of resources, making this attack much more expensive.

Any bounded resource can be used as the "payment"; one of the more common is computing resources, since they do not require any additional infrastructure beyond that already needed to access the Internet. In order to ensure that users actually do spend the appropriate resource payment, the users must employ a "proof of work".

Proofs of work have been used for reducing spam [6], for defending against denial-of-service attacks [15] and fairly recently, as the underlying mechanism

for implementing a decentralized bulletin-board—this is the technical heart of the Bitcoin protocol [11].

While effective, proofs-of-work have a significant drawback; they require energy in direct proportion to the resource used (i.e., the amount of electricity required to run the CPU during the proof of work generally depends linearly on the amount of work being performed). This is especially problematic in the context of the Bitcoin protocol, since the security of the system relies on all honest parties *constantly* performing proofs of work. In addition to having an environmental impact, this also sets a lower bound on transaction fees (since rational parties would only participate in the protocol if their reward exceeds their energy cost). Motivated in large part by the need to replace proofs-of-work as a basis for crypto-currencies, two (very similar) proposals for *Proofs of Space* (PoS) have been published [7,2]. Park et al. also designed an alternative crypto-currency that is based on Proofs of Space [12].

A PoS is a two-phase protocol[1]: it consists of an initialization phase and (sometime later) an execution phase. In an (N_0, N_1, T)-PoS the prover shows that she either (1) had access to at least N_0 storage between the initialization and execution phases and at least N_1 space during the execution phase, or (2) used more than T time during the execution phase.

At first glance, this definition might seem sufficient as a replacement for proof-of-work. However, in contrast to work, space can be reused. Using the PoS definition as a "resource payment" scheme thus violates two properties we would like any such scheme to satisfy:

1. **Amortization-Resistance**: A prover with access to $\max(N_0, N_1)$ space can, without violating the formal PoS security guarantee, generate an arbitrary number of different (N_0, N_1, T)-PoS proofs while using the same amount of resources as an honest prover generating a single proof; thus, the *amortized* cost per proof can be arbitrarily low.
2. **Rationally Stored Proofs**: Loosely speaking, in a *rationally stored proof* a verifier is convinced that a rational prover has expended a space resource over a period of time. There may exist a successful adversarial strategy that does not require the adversary to expend space over time, but this strategy will be more costly than the honest one. If we are interested in designing a crypto-currency that replaces CPU work with a space-based resource, our proof of resource consumption must be a rationally stored proof, otherwise rational parties will prefer to use the adversarial strategy, and we can no longer claim that the crypto-currency is energy-efficient.
   The *cost* of storage is proportional to the product of the storage space and the time it is used (e.g., in most cloud storage services, it costs the same to store 10TB for two months or 20TB for one month[2]). Under the PoS definition, a

[1] We use the formal definitions of [7], which are more general than those in [2]

[2] Of course, this is also true for a local disk; during the interval in which we are using the disk to store data A, we can't use it to store anything else, so our "cost" is the utility we could have gained over the same period (e.g., by renting out the disk to a cloud-storage company).

prover can pay an arbitrarily small amount by discarding almost all stored data after the initialization phase and rerunning the initialization in the execution phase (the prover only needs to store the communication from the verifier in the initialization phase). More generally, a *rational* prover will prefer to use computation over storage whenever the cost of storing the data between the phases is greater than the cost of rerunning the initialization; when this occurs the PoS basically devolves into a standard proof-of-work in terms of energy usage.
Even if we ignore energy use, this is a problem if the PoS is used in a protocol where the prover must generate many proofs, but only some will be verified: the dishonest prover will not have to expend resources on the unverified proofs in this case.

Although the protocols of [7,2] are not completely undermined by these attacks, they are more than just a definitional problem. In particular, in the suggested PoS protocols based on graph pebbling, the *work* performed by the honest prover in the initialization phase is proportional to the work required to access the graph (i.e., $O(N_0)$). It's not clear how to increase the initialization costs without increasing either the memory size or verification cost linearly. This strongly bounds the time that can be allowed between the initialization and execution phases if we want rational provers to use space resources rather than CPU work. In the Spacemint protocol, for example, the authors suggest running the proofs every minute or so [12]. If one wanted to run a proof only once a month, a rational miner might prefer to rerun the initialization phase each time.

1.1 Our Contributions

"Fixed" Definition. In this paper, we define a new proof-of-resource-payment scheme: a "Proof of Spacetime" (PoST), that we believe is better suited as a scalable energy-efficient replacement for proof-of-work. Our definition is similar to a Proof of Space, but addresses both amortization and rationality of storage.

In a PoST, we consider two different "spendable" resources: one is CPU work (i.e., as in previous proofs-of-work), and the second is "spacetime": filling a specified amount of storage for a specified period of time (during which it cannot be used for anything else); we believe spacetime is the "correct" space-based analog to work (which is a measure of CPU power over time). Like work, spacetime is directly convertible to cost.

Rational Storage vs. Space Rather than require the prover to show exactly which resource was spent in the execution phase, we allow the prover to choose arbitrarily the division between the two, as long as the total amount of resources spent is enough.

That is, the prover convinces a verifier that she *either* spent a certain amount of CPU work, *or* reserved a certain amount of storage space for some specified period of time or spent some linear combination of the two. However, by setting parameters correctly, we can ensure that *rational* provers will prefer to use space-time over work; when this is the case we say that a PoST is *Rationally Stored*

(we give a formal definition in Section 2.2). In situations where it is reasonable to assume rational adversaries (such as in crypto-currencies), our definition opens the door to new constructions that might not satisfy the PoS requirements. For example, the PoS definition essentially requires a memory-hard function, while our construction is rationally stored but is *not* memory-hard!

Novel Construction. We construct a PoST based on *incompressible* proofs-of-work (IPoW); a variant of proofs-of-work for which we can lower-bound the storage required for the proof itself. We give a candidate construction based on the standard "hash preimage" PoW. Our protocols and proofs use a very different technique than the Proofs of Space protocols, and we believe they are simpler to implement.

Market-Based Parameter Adjustment. One advantage of our different technique is that we can support market-based adjustment for our protocol parameters. In particular, by using a modified version of our PoST protocol, we can detect whether rational users are recomputing or storing data (see Section 5 for a sketch of the solution). This allows us to build protocols that automatically increase the difficulty when the price of storage rises (in which case we'd expect to see more users choosing computation over storage).

Standard Assumptions. Our constructions (and proofs) are in the random oracle model (like most previous work on memory-hard functions and proofs of space). Unlike the existing PoS constructions, our analysis does not require any non-standard assumptions (we use an information-theoretic argument to reduce breaking the soundness to compressing a random string).

Different Parameter Regimes. In comparison with [7], we think of the time between the initialization and proof phases as *weeks* rather than minutes (this could enable, for example, a crypto-currency in which the "miners" could be completely powered off for weeks at a time). One can think of our constructions as complementary to the existing PoS constructions for different parameter regimes—On the one hand, the proof phase of our PoST protocol is less efficient (it requires access to the entire storage, so a proof might take minutes rather than seconds, as is the case for the pebbling-based constructions.[3] This means it is not as well suited to very short periods between proofs). On the other hand—unlike the existing PoS constructions—the computational difficulty of our initialization phase is tunable *independently* of the amount of space, so it is possible to use it to prove reasonable storage size over long periods (e.g., weeks or months). In this parameter regime, a proof that takes several minutes would be reasonable.

Compared to pebbling-based constructions, the big loss of efficiency is on the *prover*'s side. In our construction, the prover must read (more or less) the entire table in order to generate a valid response to a challenge. This is indeed much worse asymptotically. Of course this is a drawback of our construction, and improving this is certainly a worthwhile goal. In practical terms, however, our

[3] To put things in perspective: a fast consumer SSD today has random-access throughput of around 400MB/s; this means reading through a 100GB table in about 4 minutes, which is reasonable even if challenges occur every few hours, much less every few weeks

efficiency doesn't preclude the use-cases we describe (e.g., a fast consumer SSD today has random-access throughput of 400MB/s; this means reading through a 100GB table in about 4 minutes, which is reasonable even if challenges occur every few hours, much less every few weeks).

Improvements to Spacemint. Finally, we propose a modification to the Spacemint crypto-currency protocol that removes some restrictions on the types of PoS protocols it can use—allowing it to use PoSTs rather than the specific PoS constructions it is currently based on (see Section 6)

1.2 Related Work

Memory-Bound Functions. One of the inspirations for our protocol is the memory-bound function defined by Dwork, Goldberg and Naor [5]. The goal in designing a memory-bound function is to make the performance bottleneck for function evaluation the memory latency rather than CPU speed. The DGN design uses a "pointer-chasing" technique in a random table to prove that an adversary who computes the function must touch many entries. Our protocol is based on a similar idea. However, since our security goals are different (e.g., we do not need to ensure the protocol actually uses a large amount of space, but do need to enforce a tradeoff between space and work) our protocol (and proof techniques) are different.

Memory-Hard Functions. Loosely speaking, a memory-hard function is a function that requires a large amount of memory to evaluate [13,1]. One of the main motivations for constructing such functions is to construct proofs-of-work that are "ASIC-resistant" (based on the assumption that the large memory requirement would make such chips prohibitively expensive). Note that the proposed memory-hard functions are still proofs-of-*work*; the prover must constantly utilize her CPU in order to produce additional proofs. PoSTs, on the other hand, allow the prover to "rest" (e.g., by turning off her computer) while still expending space-time (since expending this resource only requires that storage be filled with data for a period of time).

Proofs of Storage/Retrievability. In a proof-of-storage/retrievability a prover convinces a verifier that she is correctly storing a file previously provided by the verifier [8,4,3,9,14]. The main motivation behind these protocols is verifiable cloud storage; they are not suitable for use in a PoST protocol due to high communication requirements (the verifier must send the entire file to the server in the first phase), and because they are not publicly verifiable. That is, if the prover colludes with the owner of the file, she could use a very small amount of storage space and still be able to prove that she can retrieve a large amount of pseudorandom data.

Permacoin. Miller, Juels, Shi, Parno and Katz proposed the Permacoin protocol, a cryptocurrency that includes, in addition to the standard PoWs, a special, distributed, proof of retrievability that allows the cryptocurrency to serve as a

distributed backup for *useful* data [10]. In strict contrast to PoSTs, the Permacoin construction is amortizable *by design*—an adversary who stores the entire dataset can reuse it for as many clients as it wishes. Thus, Permacoin still requires regular PoWs, and cannot be used to replace them entirely with a storage-based resource. Also by design, clients require a large amount of communication to retrieve the data they must store, in contrast to PoSs and PoSTs in which clients trade computation for communication.

2 Proofs of Spacetime
(Version Id)

A PoST deals in two types of resources: one is processing power and the other is storage. All our constructions are in the random oracle model—we model processing power by counting the number of queries to the random oracle.

Modeling storage is a bit trickier. Our purpose is to allow an *energy-efficient* proof-of-resource-consumption for rational parties, where we assume that the prover is rewarded for each successful proof (this is, roughly speaking, the case in Bitcoin). Thus, simply proving that you used a lot of space in a computation is insufficient; otherwise it would be rational to perform computations without pause (reusing the same space). Instead, we measure spacetime—a unit of space "reserved" for a unit of time (and unusable for anything else during that time). To model this, we separate the computation into two phases; we think of the first phase as occurring at time $t = 0$ and the second at time $t = 1$ (after a unit of time has passed). After executing the first phase, the prover outputs a state $\sigma \in \{0,1\}^*$ to be transferred to the second phase; this is the only information that can be passed between phases. The size of the state $|\sigma|$ (in bits) measures the space used by the prover over the time period between phases.

Informally, the soundness guarantee of a PoST is that the *total* number of resource units used by the adversary is lower bounded by some specified value—the adversary can decide how to divide them between processing units and spacetime units.

We give the formal definition of a PoST in Section 2.2, in Section 3 we present a simple construction of a PoST, and in Section 3.1 we prove its security.

2.1 Units and Notation

Our basic units of measurement are CPU throughput, Space and Time. These can correspond to arbitrary real-world units (e.g., 2^{30} hash computations per minute, one Gigabyte and one minute, respectively). We define the rest of our units in terms of the basics:

- Work: CPU×time; A unit of CPU effort expended (e.g., 2^{30} hash computations).
- Spacetime: space×time; A space unit that is "reserved" for a unit of time (and unusable for anything else during that time).

In our definitions, and in particular when talking about the behavior of *rational* adversaries, we would like to measure the total cost incurred by the prover, regardless of the type of resource expended. To do this, we need to specify the conversion ratio between work and spacetime:

Real-world Cost. We define γ to be the work-per-spacetime cost ratio in terms of real-world prices. That is, in the real-world one spacetime unit costs as much as γ work units (the value of γ may change over time, and depends on the relative real-world costs of storage space and processing power).

We define the corresponding cost function, the *real-world cost* of a PoST to be a normalized cost in work units: a PoST that uses $|\sigma|$ spacetime units and x work units has real-world cost $c = \gamma|\sigma| + x$.

2.2 Defining a PoST Scheme

A PoST scheme consists of two phases, each of which is an interactive protocol between a prover $P = (P_{\text{init}}, P_{\text{exec}})$ and a verifier $V = (V_{\text{init}}, V_{\text{exec}})$.[4] (for brevity, we drop the *init* and *exec* subscripts when they are clear from the context.) Both parties have access to a random oracle H.

Initialization Phase Both parties receive as input an id string $id \in \{0,1\}^k$. At the conclusion of this phase, both the prover and the verifier output state strings ($\sigma_P \in \{0,1\}^*$ and $\sigma_V \in \{0,1\}^*$, respectively):

$$(\sigma_P, \sigma_V) \leftarrow \left\langle P^H_{\text{init}}(id), V^H_{\text{init}}(id) \right\rangle \ .$$

Execution Phase Both parties receive the id and their corresponding state from the initialization phase. At the end of this phase, the verifier either accepts or rejects ($out_V \in \{0,1\}$, where 1 is interpreted as "accept"). The prover has no output:

$$(\cdot, out_V) \leftarrow \left\langle P^H_{\text{exec}}(id, \sigma_P), V^H_{\text{exec}}(id, \sigma_V) \right\rangle \ .$$

The execution phase can be repeated multiple times without rerunning the initialization phase. This is critical, since the initialization phase requires work, while the execution phase is energy-efficient. Thus, although a single execution of the PoST does not give any advantage over proof-of-work, the amortized work per execution can be made arbitrary low.

PoST Parameters A PoST has three parameters: w, the *Honest Initialization Work*, m, the *Honest Storage Space*, and f, the *Space-time Tradeoff Function*,

[4] Although the definition allows general interaction, in our construction the first phase is non-interactive (the prover sends a single message) and the second consists of a single round.

Honest Initialization Work. We denote w the expected work performed by the *honest* prover in the initialization phase. This should be "tunable" to ensure that storing the output remains the rational choice rather than recomputing the initialization as the space-time to work cost ratio changes.

If the cost of the initialization phase is too low, the adversary can generate a proof more cheaply than an honest prover by deleting all data after initialization, then rerunning the initialization just before the proof phase. In this case, the adversary does not store any data between phases, so does not pay *any* space-time cost. We formalize this in Definition 8 as a *rationality* attack. **Note that this is a general attack that also applies to PoS schemes**—hence they must also have a lower bound on the work required for initialization.

Honest Storage Space. This is the amount of storage the honest prover must expend during the period between the initialization and execution phases (and between successive execution phases).

Space-time Tradeoff Function. We define f to be the work/spacetime tradeoff bound for a PoST proof. $f(s)$ lower-bounds the expected work the prover needs to succeed in the execution phase if it stores up to s bits. Note that f need not be linear. In fact, the optimal f is exponential: $f(s) = w/2^s$. This is because $f(0) \leq w$ (since we can always just run the initialization phase again), and $2f(i+1) \leq f(i)$—we can always do with one bit less of storage, and try both options.

Definition 1 (PoST). *A protocol (P, V) as defined above is a (w, m, ε, f)-PoST if it satisfies the properties of completeness and (ε, f)-soundness defined below.*

Completeness

Definition 2 (PoST Completeness). *We say that a PoST is (perfectly) complete if for every $id \in \{0,1\}^{poly(k)}$ and every oracle H,*

$$\Pr\left[out_V = 1 : (\sigma_P, \sigma_V) \leftarrow \left\langle P^H_{init}(id), V^H_{init}(id)\right\rangle, (\cdot, out_V) \leftarrow \left\langle P^H_{exec}(id, \sigma_P), V^H_{exec}(id, \sigma_V)\right\rangle\right] = 1 \,.$$

Note that the probability is exactly 1 and hence the completeness is perfect.

Soundness We define a security game with two phases; each phase has a corresponding adversary. We denote the adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, where $\mathcal{A}_1$ and $\mathcal{A}_2$ correspond to the first and the second phases of the game. $\mathcal{A}_1$ and $\mathcal{A}_2$ can coordinate arbitrarily before the beginning of the game, but cannot communicate during the game itself (or between phases).

Definition 3 (PoST n-Security Game). *Each phase of the security game corresponds to a PoST phase:*

1. Initialization. $\mathcal{A}_1$ *chooses a set of ids* $\{id_1, \ldots, id_n\}$ *where* $id_i \in \{0,1\}^*$. *It then interacts in parallel with* n *independent (honest) verifiers executing the initialization phase of the PoST protocol, where verifier* i *is given* id_i *as input. Let* $\sigma_{\mathcal{A}}$ *be the output of* $\mathcal{A}_1$ *after this interaction and* $(\sigma_{V_1}, \ldots, \sigma_{V_n})$ *be the outputs of the verifiers.*
2. Execution. *The adversary* $\mathcal{A}_2(id_1, \ldots, id_n, \sigma_{\mathcal{A}})$ *interacts with* n *independent verifiers executing the execution phase of the PoST protocol, where verifier* i *is given* (id_i, σ_{V_i}) *as input.*[5]

We say the adversary has succeeded if all of the verifiers output 1 *(we denote this event* $\mathbf{Succ}_n$*)*

For a party P, let $q_P^{\#}$ be the number of queries P makes to the oracle H.

Definition 4 (PoST (ε, f)-Soundness). *We say a PoST protocol is* (ε, f)-sound *if for all* $s > 0$ *and all* $n \geq 1$*, every adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *that uses at most* s *storage must satisfy the following conditions in the PoST security game:*

1. Rational Storage*:* $\mathbb{E}\left[q_{\mathcal{A}_1}^{\#} | \mathbf{Succ}_n\right] \geq \epsilon \cdot w \cdot n$.
2. Space-Time Trade-Off*: If* $s < n \cdot m$ *then* $\mathbb{E}\left[q_{\mathcal{A}_2}^{\#} | \mathbf{Succ}_n\right] \geq n \cdot f(s/n)$.

The first condition checks that the adversary spends at least an ε fraction of the honest work in the initialization phase. This prevents the adversary from launching a "rationality attack": if the initialization phase requires very little computational effort, the prover can "throw out" the stored data from the initialization phase and rerun the phase to regenerate any needed data during the execution phase. This would make its total space-time cost negligible (since the "time" component vanishes).

The second condition bounds the trade-off between space-time and work. Intuitively, an PoST satisfying this definition forces an adversary to trade space for queries. The use of n ids rather than just one prevents an amortization attack, wherein the adversary reuses the same space for different proofs. Naïvely, to generate n proofs the prover would require n times the queries, splitting the storage equally between them, hence the $n \cdot f(s/n)$. Ideally f would meet these bounds exactly, However, we allow imperfect soundness, in which case the bound implied by f can be worse than those given by the naïve approach.

Rationally Stored Proofs of Work Our high-level goal in this paper is to construct *energy-efficient* proofs, by forcing provers to use storage rather than work. Unfortunately, our definitions (and contructions) don't allow a prover to *prove* they used storage (this is actually impossible if the adversary can simulate the initialization phase without a lot of storage—which is always the case unless communication in the initialization phase is proportional to storage or we use non-standard assumptions). However, we can still give conditions under which a

[5] Each of the verifiers runs a copy of the honest verifier code with independent random coins; $\mathcal{A}_2$, however, can correlate its sessions with the verifiers.

rational prover (whose goal is to minimize expected total cost) would prefer to use storage. As long as these conditions are met, it seems reasonable to assume that real-world users would choose storage over work (especially in a cryptocurrency setting, where profit is the main motive for participating).

Definition 5 ((γ, ε')-Rationally-Stored PoST). *We say a (w, m, ε, f)-PoST is* (γ, ε')-rationally stored *if for all $n \geq 1$, the optimal execution strategy requires storing an ε'-fraction of the honest storage:*

$$s^* = \arg\min_s (n \cdot f(s/n) + \gamma s) \geq \varepsilon' \cdot n \cdot m$$

In a protocol satisfying this definition, when the real-world cost of a space unit is less than γ, the optimal strategy for reducing real-world cost is to store at least ε' compared to the honest storage cost. Note that we omit the initialization cost in Definition 5. This is because it is only incurred once, while the cost of the execution phase is incurred repeatedly. If we run the PoST execution phase k times, the total cost to the adversary would be lower-bounded by $k(n \cdot f(s^*/n) + \gamma s^*) + \varepsilon \cdot n \cdot w \geq k \cdot \varepsilon' \cdot n \cdot m + \varepsilon \cdot n \cdot w$, while the honest cost would be $k \cdot n \cdot m + n \cdot w$; thus, for any constants ε, w and m, the ratio is lower-bounded by

$$\frac{k \cdot \varepsilon' \cdot n \cdot m + \varepsilon \cdot n \cdot w}{k \cdot n \cdot m + n \cdot w} = \frac{\varepsilon' \cdot m}{m + \frac{1}{k} \cdot w} + \frac{1}{k} \frac{\varepsilon \cdot w}{m + k \cdot w} \xrightarrow{k \to \infty} \varepsilon'$$

Note that this definition implies:

$$n \cdot f(0/n) + \gamma \cdot 0 = n \cdot f(0) \geq n \cdot f(s^*/n) + \gamma s^* \geq \gamma s^* \geq \gamma \cdot \varepsilon' \cdot n \cdot m \ .$$

Thus, for a PoST to be (γ, ε')-rationally stored, it must hold that $\gamma \leq \frac{f(0)}{\varepsilon' \cdot m}$

Comparison with the PoS definition As we remarked in the introduction, an (N_0, N_1, T)-PoS does not give any security guarantees with respect to the PoST definition (even if we ignore amortization), since it does not address rationality attacks at all. In the other direction, even an optimal $(w, m, 1, w/2^{-s})$-PoST can't guarantee a (x, x, w)-PoS, for any $x \in (0, w)$ (e.g., our construction allows an adversary to easily trade space for work). Thus, the parameters are not truly comparable.

One can think of the two definitions as being targeted at different "regimes": a PoS forces the prover to use a lot of space, but is not well suited to long periods between the initialization and execution phases, while the PoST definition does allow long periods of elapsed time (with a suitably hard initialization step), but relies on the rationality of the adversary to enforce use of storage rather than work.

2.3 Constructing a PoST: High-Level Overview

Our proof of spacetime has each prover generate the data they must store on their own. To ensure that this data is cheaper to store than to generate (and

to allow public verifiability), we require the stored data to be a proof-of-work. We construct our protocol using the abstract notion of an incompressible-proof-of-work (IPoW); this is a proof-of-work (PoW) that is non-compressible in the sense that storing n different IPoWs requires n times the space compared to storing one IPoW (we define them more formally below; see Section 2.4).

As long as the cost of storing an IPoW proof is less than the cost of re-computing it, the prover will prefer to store it. However, this solution is very inefficient: it requires the prover to send its entire storage to the verifier. In order to verify the proof with low communication, instead of one large proof of work, we generate a table containing T entries; each entry in the table is a proof of work that can be independently verified.

Why the Naïve Construction Fails. At first glance, it would seem that there is an easy solution for verifying that the prover stored a large fraction of the table:

1. In the initialization phase: the prover commits to the table contents (using a Merkle tree whose leaves are the table entries)
2. In the execution phase: the verifier sends a random set of indices to the prover, who must then respond with the corresponding table entries and commitment openings (merkle paths to the root of the tree).

Unfortunately, this doesn't work: the prover can discard the entire table and reconstruct only those entries requested by the verifier during the execution phase.

Instead of challenging the prover on random table entries, we use a trick from Dwork et al.'s memory-bound function construction [5]: the challenge provided by the verifier defines *multiple sets* of entries. The prover must search through the table to find a set of entries that satisfies a "probing criterion". We set the difficulty of the probing criterion to ensure the prover must read a large fraction of the table in order to find a satisfying set; thus, the prover must either have stored or recomputed a large fraction of the table.

In a little more detail: for a challenge ch, every nonce $nonce$ selected by the prover will define the set of k entries $H(nonce||ch||1), \ldots, H(nonce||ch||k)$, where the output of the random oracle is interpreted as a pointer to the table. We use the random oracle for the probing criterion too; setting $p^* = \Theta(k/T)$ to be the probability of success, for each set of entries $(X_1, \ldots, X_k)$ the prover must compute $H(X_1, \ldots, X_k)$, interpret the output of the oracle as a real fraction in $[0, 1)$ and check if it is less than p^*.

Intuitively, the only thing the prover can do is try many different nonces until one succeeds (in expectation $1/p^* = \Omega(T/k)$ attempts), but because the entries defined by each nonce are random and independent, with high probability the attempts will cover most of the table. Essentially, by setting p^* low enough, we force the prover to read the most of the table, even though the final output consists of only k entries and their Merkle paths to the root of the tree used to commit the table (k can be polylogarithmic in the table size).

2.4 Incompressible Proofs of Work

The standard definitions of PoWs do not rule out an adversary that can store a small amount of data and can use it to regenerate an entire table of proofs with very low computational overhead. Thus, to ensure the adversary must indeed store the entire table we need a more restrictive definition:

An *Incompressible Proof of Work* (IPoW) can be described as a protocol between a verifier V and a prover P:

1. The prover P is given a challenge ch as input, and outputs a "proof" π:
2. The verifier receives (ch, π) and outputs 1 (accept) or 0 (reject).

For simplicity, we denote IPoW (ch) the output of the honest prover on challenge ch (this is a random variable that depends on the random oracle and the prover's coins).

Defining an IPoW Let $q_P^{\#}$ denote the number of oracle calls made by P in the protocol (this is a random variable that depends on ch and the random coins of P).

Definition 6 ((w', m, f)-IPoW). *A protocol is a (w', m, f)-IPoW if:*

1. $\mathbb{E}\left[q_P^{\#}\right] \leq w'$ *(the honest prover's expected work is bounded by w'),*
2. $|\pi| \leq m$ *(the honest prover's storage is bounded by m) and*
3. *The IPoW is complete (c.f. Definition 7) and f-sound (c.f. Definition 8)*

Definition 7 (IPoW Completeness). *An IPoW protocol is* complete *if, for every challenge ch, the probability that the verifier rejects is negligible in the security parameter (the probability is over the coins of the prover and the random oracle).*

Definition 8 (IPoW $f(x)$-Soundness). *Let $\mathcal{A}^{(n,s)} = (\mathcal{A}_1^{(n,s)}, \mathcal{A}_2^{(n)})$ be an adversary such that $\mathcal{A}_1^{(n,s)}$ outputs a string σ with length $|\sigma| \leq s$, while $\mathcal{A}_2^{(n,s)}$ gets σ as input and outputs n pairs $(ch_1, \pi_1, \ldots, ch_n, \pi_n)$. Denote* **Succ** *the event (over the randomness of $\mathcal{A}^{(n,s)}$ and the random oracle) that all the challenges are distinct and $\forall i \in [n] : V(ch_i, \pi_i) = 1$. An IPoW protocol is f-*sound *if for every adversary and all $n \geq 1, s \in [0, m)$,*

$$\mathbb{E}\left[q^{\#}_{\mathcal{A}_2^{(n,s)}} \mid \mathbf{Succ}\right] \geq n \cdot f(s/n)$$

As in the PoST definition, this condition bounds the trade-off between space-time and work for the IPoW adversary. Note that we don't restrict the number of queries $\mathcal{A}_1^{(n,s)}$ makes to the oracle.

Rationally Stored IPoWs As for rationally-stored PoSTs, we define a condition which implies that for small enough γ, a rational adversary's optimal strategy is storage. Since we aren't concerned with rationality attacks, the definition for IPoWs is a little simpler:

Definition 9 ((γ, ε)-Rationally-Stored IPoW). *We say a (w', m, f)-IPoW is* (γ, ε)-rationally stored *if for all $n \geq 1$*

$$\arg\min_{s}(n \cdot f(s/n) + \gamma s) \geq \varepsilon \cdot n \cdot m$$

3 Our PoST Construction: The Details
(Version Id)

Formally, we describe the protocol in the presence of several random oracles, denoted by H_i; for $i \neq j$, H_i and H_j are independent random oracles. This is just for convenience of notation, we can implement them all using a single oracle by assigning a unique prefix to the oracle queries (e.g., $\mathrm{H}_i(x) = H(i||x)$).

The formal PoST protocol description appears as Protocol 1. To construct it, we use a (w', m, f)-IPoW. Our soundness proof requires that the amount of work per IPoW verification is at most $\frac{1}{10} \cdot w'$. We construct such a hash-based IPoW scheme in Section 4.

3.1 Security Proof Sketch

Theorem 1 (Informal). *If the challenge ch is chosen with high min-entropy, Protocol 1 is a sound PoST protocol*

Proof. For simplicity of the proof, we assume that $\varepsilon = 1/4$ Recall that Definition 3 offers the adversary two ways to win in the security game; we treat each case separately.

1. Suppose there exists an adversary that wins with non-negligible probability under the first condition (i.e. a rationality attack). In this case, the adversary cannot have solved more than half of the table entries' proofs of work. Therefore, when given a new random challenge it will either have to find a path that exists entirely in the half of the table it did solve, or break the Merkle commitment. Note that searching for a path of length k that falls entirely in one half of the table is equivalent to flipping k coins until they all return 1—the choice of nonce determines the entire path (given the challenge and existing table entries), and the random oracle ensures that each index in the path is chosen randomly and independently of the previous links. Thus, the adversary would require a number of random oracle queries exponential in k to succeed (with high probability) (This argument is formalized in Lemma 1.)

Protocol 1 TABLE-POST (for $\gamma^* = w'/m$)

Public Parameters: k - security parameter, T - table size and IPoW (ch) is a (w', m, f)-IPoW. Denote $p^* = k/T$

Storing Phase: (Performed by the prover P)

Inputs: $id \in \{0,1\}^*$.

1. Generate an array G of size T as follows:
   For each $0 \leq i < T$, set $G[i] \stackrel{\text{def}}{=} \text{IPoW}\,(id||i)$
2. Generate a commitment com on G using a Merkle tree (i.e., construct a Merkle tree whose leaves are labeled with the entries of G, and each internal node's label is the output of the random oracle on the concatenation of its children's labels; com is the root label).
3. Publish the string id and the commitment com.

Proof Phase: (Performed by the prover P)

Upon receiving a challenge ch from the verifier V:

for all $j \in \{1, \ldots, k\}$ **do**
  $count \leftarrow 0$
  repeat
    $count \leftarrow count + 1$ // Increment counter
    Set $nonce_j \leftarrow count$
    for all $t \in \{1, \ldots, k\}$ **do**
      Compute $i_{j,t} = \text{H}_1\,(nonce_j||ch||j||t) \mod T$
    end for
    Let $\pi_{decommit(j)}$ be the Merkle paths from the table entries $\{G[i_{j,1}], \ldots, G[i_{j,k}]\}$.
    Compute $pathprobe \leftarrow \text{H}_2\,(i_{j,1}||G[i_{j,1}]|| \cdots ||i_{j,k}||G[i_{j,k}]||\pi_{decommit(j)})$
  until $pathprobe < p^*$ // happens with prob. p^*
end for
Output to V the list $(nonce_1, \ldots, nonce_k)$ and $\{\pi_{decommit(1)}, \ldots, \pi_{decommit(k)}\}$.

Proof Phase: (Performed by the verifier V)

Generate a random challenge ch and send it to the prover. Wait to receive the list $(nonce_1, \ldots, nonce_k)$ and $\{\pi_{decommit(1)}, \ldots, \pi_{decommit(k)}\}$.

for all $j \in \{1, \ldots, k\}$ **do**
  for all $t \in \{1, \ldots, k\}$ **do**
    Compute $i_{j,t} = \text{H}_1\,(nonce_j||ch||j||t) \mod T$
    Verify using $\pi_{decommit(j)}$ that $G[i_{j,t}]$ has a valid commitment opening.
    Verify that $G[i_{j,t}]$ is a valid IPoW for the challenge $id||i_{j,t}$.
  end for
  Compute $pathprobe \leftarrow \text{H}_2\,(i_{j,1}||G[i_{j,1}]|| \cdots ||i_{j,k}||G[i_{j,k}]||\pi_{decommit(j)})$
  Verify that $pathprobe < p^*$
end for

2. We analyze the second case (spacetime attack) by reducing from the incompressibility of IPoW; if there exists an adversary that can beat the work+space trade-off, we can use this adversary to violate the IPoW soundness guarantee. The idea is to use the adversary as a subroutine in a program that can reconstruct a large fraction of the table. To compress, we run the adversary's initialization phase. To decompress, we run the adversary's execution phase, simulating the oracle $\mathrm{H}_2(\cdot)$. Each path probe query (query to $\mathrm{H}_2(\cdot)$) should correspond to a set of k table entries and their commitment openings; we can use this query to "reconstruct" those table entries. (we can ignore path probe queries that don't have the right format—those are "useless", since they are independent of anything the verifier queries).
   Since the prover must output k good nonces and the oracle queries to $\mathrm{H}_2(\cdot)$ are independent for each nonce, with high probability it will have to make $\Omega(1/p^*)$ path probe queries. Moreover, since the table indices are chosen i.i.d. for each different nonce and challenge, if the adversary has previously asked about less than $1/4$ of the table entries, with high probability at least $1/2$ of the indices in each path probe query will be new. Thus, each path-probe query will add at least $1/2k$ new table entries. By setting $p^* = \Omega(k/T)$, we can ensure that with high probability at least one quarter of the table entries will be reconstructed.
   The total space we need to store is just the output of the adversary's initialization phase. Moreover, the number of work oracle queries we make while reconstructing the table is exactly the number of queries made by the adversary's execution phase. By our assumption about the adversary's winning strategy, the normalized total (of space+queries) is less than the total needed to honestly reconstruct $1/4$ of the entries. Thus, we violate IPoW soundness. (This argument is formalized in Lemma 2.)

□

Due to space considerations, the formal proof is deferred to Appendix A.1.

4 Hash-Preimage IPoW
(Version Id)

One of the most popular proofs of work is the hash-preimage PoW: given a challenge $ch \in \{0,1\}^k$, interpret the random oracle's output as a binary fraction in $[0,1]$ and find $x \in \{0,1\}^k$ s.t.

$$H(ch||x) < p \tag{1}$$

p is a parameter that sets the difficulty of the proof. For any adversary, the expected number of oracle calls to generate a proof-of-work of this form is at least $1/p$.

At first glance, this might seem to be an incompressible PoW already—after all, the random oracle entries are uniformly distributed and independent, so

compressing the output of a random oracle is information-theoretically impossible. Unfortunately, this intuition is misleading. The reason is that we need the proof to be incompressible *even with access to the random oracle*. However, given access to the oracle, it's enough to compress the *input* to the oracle. Indeed, the hash-preimage PoW is vulnerable to a very simple compression attack: Increment a counter x until the first valid solution is found, but don't store the zero prefix of the counter. Since the expected number of oracle calls until finding a valid x is only $1/p$, on average that means only $\log \frac{1}{p}$ bits need to be stored (rather than the full length of an oracle entry).

We show that this is actually an optimal compression scheme. Therefore, to make this an *incompressible* PoW, we instruct the honest user to use this strategy, and store exactly the $\left\lceil \log \frac{1}{p} \right\rceil$ least significant bits of the counter. We note that $\frac{1}{p}$ is the *expected* number of attempts—in the worst case the prover may require more; thus, we allow the prover to search up to $\frac{k}{p}$ entries; the verifier will check k possible prefixes for the $\log \frac{1}{p}$ bits sent by the prover (with overwhelming probability, there will be a valid solution in this range). Thus, the verifier may have to make k oracle queries in the worst case in order to check a proof (however, in expectation it will be only slightly more than one).[6]

Formally,

Definition 10 (w'-Hash-Preimage IPoW). *The honest prover and verifier are defined as follows: Set $p = 1/w'$.*

Prover *Given a challenge y, calls H on the inputs $\{y||x\}_{x\in\{0,1\}^{\log \frac{k}{p}}}$ in lexicographic order, returning as the proof π the least significant $\log \frac{1}{p}$ bits of the first x for which $H(y||x) < p$.*

Verifier *Given challenge y and proof π, verifies that $|\pi| \leq \log \frac{1}{p}$ and that there exists a prefix z of length $\log k$ such that $H(y||z||\pi) < p$ (where π is zero-padded to the maximum length).*

Theorem 2. *The w'-hash-preimage protocol is a $(w', \log w', f(s) = \frac{1}{16} \cdot \frac{w'}{2^s})$-IPoW.*

Completeness follows by inspection. At a high level, our soundness proof works by showing that an adversary that "compresses" the IPoW can be used to compress a random string (violating Shannon's source-coding theorem). Due to space considerations, we defer the full proof to Appendix A.2.

4.1 Hash-Proof Rationality

Theorem 3. *For all $\varepsilon \in (0,1)$, the w'-Hash-Preimage IPoW is a $(2^{-5} \cdot (w')^{\varepsilon}, 1-\varepsilon)$ rationally-stored IPoW.*

[6] We note that this computation can be performed by the prover instead, but it will simplify our analysis to assume the verifier performs the checks.

This means that if $w' > \left(\gamma \cdot 2^5\right)^{(1/\varepsilon)}$ then the optimal storage strategy for the hash-preimage-based IPoW is to store at least $(1-\varepsilon)$ of the honest amount.

Proof. Suppose an adversary uses $s < n \log w'$ storage. Then by Corollary 2 the expected cost to the adversary is at least:

$$\gamma s + \mathbb{E}[q] \geq \gamma s + \frac{1}{4^{(1+1/n)}} \cdot nw' \cdot 2^{-s/n} \ .$$

Taking the derivative

$$\frac{\partial}{\partial s}\left(\gamma s + \frac{1}{4^{(1+1/n)}} \cdot nw' \cdot 2^{-s/n}\right) = \gamma - \frac{\ln 2}{4^{(1+1/n)}} \cdot w' \cdot 2^{-s/n}$$

which is monotone increasing as a function of s, and is negative when $s < n(\log w' + \log \ln 2 - \log \gamma - 2(1+1/n))$, hence an adversary minimizing cost will take $s = n(\log w' + \log \ln 2 - \log \gamma - 2(1+1/n)) \geq n(\log w' - \log \gamma - 5)$.

Since $\gamma < 2^{-5} \cdot (w')^{\varepsilon}$, it follows that a rational adversary will store at least

$$n(\log w' - \varepsilon \log w') = (1-\varepsilon) n \log w'$$

bits of storage, where an honest prover uses $n \log w'$ bits. □

5 A Market-Based Mechanism for Difficulty Adjustment (Version Id)

One of the very nice properties of PoW-based cryptocurrency schemes is that the tunable parameter of PoWs—their difficulty—can be set dynamically using a market-based solution: by counting the number of published PoW solutions, we can estimate the total computational power expended on producing PoWs, and thus update the difficulty accordingly.

A PoST scheme has two main tunable parameters—the amount of space it requires (m), and the computational cost of initialization, or difficulty parameter (w). The first parameter determines the cost of generating a good proof (since amortized over multiple proofs, the initialization cost becomes irrelevant). This parameter can be set dynamically in a similar fashion to the PoW-based schemes, by counting the total amount of space invested over a specified time period.

The difficulty parameter, on the other hand, determines the rationality of storage: the higher the cost of storage, the higher the difficulty parameter must be set in order to ensure that rational provers will prefer storage over recomputing the PoST. Unfortunately, the price of storage (relative to computation cost) can't readily be estimated simply by observing the PoST proofs (in particular, the proofs generated by recomputing the initialization are identical to "honest" proofs).

However, given a PoST with some special properties, it turns out that we *can* dynamically set the difficulty. The main idea is to construct a PoST protocol with a "computational bonus". Loosely speaking, we give a prover two *identifiable* options for generating proofs: the standard, storage-based PoST, and an

alternative that is computation-based. Our construction will ensure that when the cost of storage is low, the cost of the computation-based solution is high. However, when the cost of storage goes up, the computation-based solution will become more attractive. By giving a small "bonus" reward for solutions that use the computation-based proofs, we incentivize users to choose identify themselves as "computational solvers" when the price of storage is high enough to make computation a more attractive option. When we observe that the fraction of computational solvers changes, we can adjust the difficulty parameter to compensate.

5.1 PoSTs With Computation Bonus

Intuitively, we define a PoST with Computational Bonus to be a PoST scheme which has two modes of operation: a "large-storage" mode and a "small-storage" mode, such that the initialization procedure for the large-storage mode doesn't cost more than the that of small-storage mode. Moreover, we will require the stored data in the small-storage mode to be a prefix of the data for the large-storage mode; thus, rational provers can always run the large-storage initialization, and "forget" the extra data. If, at some point, the price of storage goes up enough to make recomputing the init phase less costly than storing the small-storage data, they will be able to use the large-storage prover, and get an extra bonus. Of course, we need to make sure that the bonus reward is small enough that when the price of storage is low, the extra cost of recomputing will be larger than the bonus.

More formally: Let $\text{prefix}_a(b)$ be the a-bit prefix of the string b.

Definition 11 (PoST with Computational Bonus). *$P = (P_{init}, P_{exec}, P_{bonus})$, $V = (V_{init}, V_{exec}, V_{bonus})$ is a (w, m, ε, f)-PoST with a computational bonus if the prover $P' = (prefix_m(P_{init}), P_{exec})$ and verifier $V' = (V_{init}, V_{exec})$ comprise an (w, m, ε, f)-PoST that is (γ, ε'_1)-rationally stored, and the prover $P'' = (P_{init}, P_{bonus})$ and verifier $V'' = (V_{init}, V_{bonus})$ comprise a $(w, m', \varepsilon_2, f_2)$-PoST that is (γ, ε'_2)-rationally stored, such that*

$$\varepsilon'_2 \cdot m' > m$$

To use get the computational bonus, we will require the prover to send a proof both for P' and P''.

Let r be the reward for a "regular" proof and r_{bonus} the reward for a bonus proof (both measured in work units).

Claim 1 (Detecting Computational Solvers). If $r_{\text{bonus}} - r > w - f(0)$ and γ' is large enough that for all $s > 0$, $f(0) < f(s) + \gamma' \cdot s$ (i.e., recomputing is an optimal strategy for P') then rational users will prefer to run the bonus prover.

Proof. Since (P'', V'') is a $(w, m', \varepsilon_2, f_2)$-PoST, it must hold that $f_2(0) \leq w$. The cost for running the regular prover is at least $f(0)$, while the cost of running the bonus prover is at most w. Thus, the expected utility for running the regular

prover is at most $r - f(0)$, and that for the bonus prover is at least $r_{\text{bonus}} - w$. As long as $r_{\text{bonus}} - r > w - f(0)$, the utility is maximized by choosing the bonus prover. □

Claim 2 (Storage Remains Rational). If $r_{\text{bonus}} - r < \gamma(\varepsilon_2' \cdot m' - m)$ and the price of storage is less than γ, rational users will prefer storage over computation.

Proof. Let s^* be the optimal storage for generating a bonus proof. Since the bonus PoST is (γ, ε_2')-rationally stored, it must hold that $s^* \geq \varepsilon_2' m$, hence the cost of generating a bonus proof is at least $\varepsilon_2' \cdot \gamma \cdot m' > \gamma \cdot m$ (the inequality follows from the property of the PoST with computational bonus). Since the cost of generating a regular proof is at most γm (by honestly storing the init data), it follows that it is rational to generate a regular proof as long as $r_{\text{bonus}} - r < \gamma(\varepsilon_2' \cdot m' - m)$. □

5.2 Constructing PoSTs with Computational Bonus (Sketch)

Our Table-PoST construction based on Hash IPoWs can be modified to support a computational bonus. The key insight is that when solving a Hash IPoW for difficulty p, on average we will see k solutions to the IPoW for difficulty $p \cdot k$. Our "extra data" will be these solutions (i.e., the bonus prover's IPoW will include one solution with difficulty p and $\alpha \cdot k$ solutions of difficulty $p \cdot k$, where α and k are parameters that control how much extra storage is required, and the probability that extra computation will be required). When recomputing an IPoW, the extra solutions are essentially free in terms of computation; however, even given a solution with difficulty p, finding $\alpha \cdot k$ additional solutions with difficulty $p \cdot k$ would require expected work close to α/p.

5.3 Incremental Difficulty Adjustment

Although in our analysis we treat the initialization phase as a one-time operation (and hence can amortize away its complexity), if we increase the difficulty, the data generated by a previous init phase will no longer be valid (since the IPoWs in our PoST table will not satisfy the new difficulty level).

However, a nice property of the hash-based Table-PoST is that we can incrementally increase the difficulty. If we increase difficulty from p to $p' < p$, then on average p/p' of the entries will already satisfy the new difficulty level. Moreover, for those that do not, since we stored the last index we reached in the search for a good solution, we can simply "continue" running the Hash-IPoW solver where it left off. Thus, the total work we expend (including the first initialization phase) will be only $1/p'$.

6 Using PoSTs in Spacemint (Version Id)

Spacemint is a crypto-currency based on PoSs rather than PoWs [12]. Spacemint was designed to be used with the pebbling-based PoS constructions; our PoST

construction is not a drop-in replacement. However, we believe some simple modifications to Spacemint would allow it to be used with PoSTs as well (and thus provide an option for an even more "restful" crypto-currency). Below, we briefly sketch the main problem encountered in using the unmodified Spacemint with PoSTs, and how we overcome it. (We note that the Spacemint construction is fairly complex, and we do not include an in-depth description here. For more details on Spacemint, we refer the reader to [12].)

Like Bitcoin, Spacemint is based on a blockchain, in which blocks are generated by "lottery"; the winner of the lottery is allowed to add her block to the chain and claim the associated rewards. In Bitcoin, the winner is the first miner to solve a hash-puzzle. Thus, the probability of winning depends on the ratio between the miner's hashrate and that of the entire network. In Spacemint, the winner of the lottery is the miner whose answer (i.e., proof) to a PoS challenge has the best "quality". To prevent all miners from flooding the network with their proofs, miners first test their proof against a basic "quality threshold", and only if it passes do they post the entire proof. Like the hash difficulty, the quality threshold can be set so that the expected communication is constant, and does not depend on the total number of miners.

Unfortunately, this solution runs into a problem when replacing their PoS construction with our PoST: Unlike the pebbling-based PoS, our PoST construction allows many valid proofs for each challenge. Thus, rational users would "grind", wasting computational power on finding a good proof.

6.1 The Alternative Lottery Mechanism

Our alternative lottery mechanism uses two new ideas:

Two-phase challenge. We separate the lottery into two challenge phases: In the first challenge phase, an initial challenge is revealed, and every miner must generate a PoST proof using that challenge. The miners then publish a *commitment* to their proof (and must do so before the second phase). In the second challenge phase, a second challenge is revealed, and miners use this second challenge to test the quality of their proof (e.g., by hashing the proof together with the second challenge). As in the original Spacemint, the valid proof with the highest quality wins, and all miners with a proof that passes the quality threshold will publish their entire proofs.

Since we allow each miner only a single commitment, and miners must commit before learning the second challenge, griding is useless—there is no way to determine the quality of a proof when generating it.

Note that the challenges themselves can be generated in the same manner as Spacemint. Here we benefit from the fact that the challenge in Spacemint is produced ahead of the actual block generation time; this allows us to run the two-phase protocol without delaying block generation.

Initial quality filter. The two-phase challenge, by itself, still requires all miners to send a commitment, making the total communication at least linear in the

number of miners. To reduce the communication, we propose a further modification: a pre-filter that does not use the PoST at all—just the commitment to the stored data. The idea is that the first challenge will be used to select a subset of entries in the stored data table. Only if the hash of these entries is greater than an initial "quality" filter will the miner be eligible to generate a proof and participate in the full lottery (the miners will prove they are eligible by sending the relevant entries together with a Merkle path opening).

This reduces communication, and also greatly increases the time between PoST proofs (since miners who don't pass the initial filter will not have to run the PoST proof phase); here we make strong use of the fact that it is rational to store the PoST data for long periods rather than rerun the initialization phase.

7 Discussion and Open Questions
(Version Id)

Trading off Work and Space. One of the apparent drawbacks of our IPoW construction is that the tradeoff between work and space is fixed, and exponential. However, this can be overcome by defining a "composite" IPoW made up of several IPoWs in sequence (e.g., with challenges formed by appending an index to the original challenge). This allows us to create a $(t \cdot w', t \cdot m, t \cdot f(s/t))$-IPoW from a $(w', m, f(s))$-IPoW, giving much more flexibility in the space/time tradeoff, at the cost of increased verification time.

Improving Proving Complexity. Compared to PoS, our prover complexity (at least asymptotically) is much worse: the PoST prover has read the entire table in order to generate a proof. It might be possible to combine the PoS pebbling-based protocols with our IPoW construction to get "the best of both worlds"—fast proving time and finely-tunable difficulty with market-based adjustment—by having each pebble be an IPoW (whose challenge is given by the hash of its predecessor pebbles).[7] Proving the security of this construction appears to be non-trivial, however.

Constructing additional IPoW constructions using different techniques is also an interesting open question.

References

1. J. Alwen and V. Serbinenko. High parallel complexity graphs and memory-hard functions. In R. A. Servedio and R. Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 595–603. ACM, 2015.
2. G. Ateniese, I. Bonacina, A. Faonio, and N. Galesi. Proofs of space: When space is of the essence. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, pages 538–557, 2014.

[7] Thanks to the anonymous reviewer who suggested this idea!

3. G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. Song. Provable data possession at untrusted stores. *IACR Cryptology ePrint Archive*, 2007:202, 2007.
4. K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: theory and implementation. In R. Sion and D. Song, editors, *CCSW*, pages 43–54. ACM, 2009.
5. C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.
6. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In E. F. Brickell, editor, *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1992.
7. S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of space. In R. Gennaro and M. Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, volume 9216 of *Lecture Notes in Computer Science*, pages 585–605. Springer, 2015.
8. P. Golle, S. Jarecki, and I. Mironov. Cryptographic primitives enforcing communication and storage complexity. In M. Blaze, editor, *Financial Cryptography*, volume 2357 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2002.
9. A. Juels and B. S. K. Jr. Pors: proofs of retrievability for large files. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 584–597. ACM, 2007.
10. A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014*, pages 475–490. IEEE Computer Society, 2014.
11. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. webpage, 2008. `https://bitcoin.org/bitcoin.pdf`.
12. S. Park, K. Pietrzak, J. Alwen, G. Fuchsbauer, and P. Gazi. Spacemint: A cryptocurrency based on proofs of space. IACR Cryptology ePrint Archive, Report 2015/528, 2015. `http://eprint.iacr.org/2015/528`.
13. C. Percival. Stronger key derivation via sequential memory-hard functions. BSDCan 2009, 2009.
14. R. D. Pietro, L. V. Mancini, Y. W. Law, S. E., and P. J. M. Havinga. Lkhw: A directed diffusion-based secure multicast scheme for wireless sensor networks. In *ICPP Workshops*, pages 397–. IEEE Computer Society, 2003.
15. B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for dos resistance. In V. Atluri, B. Pfitzmann, and P. D. McDaniel, editors, *ACM Conference on Computer and Communications Security*, pages 246–256. ACM, 2004.

A Full Proofs
(Version Id)

A.1 Security Proof for Table-PoST

For simplicity of the proof, we assume that $\varepsilon = 1/4$. Recall that Definition 3 offers the adversary two ways to win in the security game; we treat each case separately.

First, assume that the adversary is deterministic (this is w.l.o.g. since the adversary is computationally unbounded). Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be an adversary in the PoST security game (defined in Definition 3) such that $\mathbb{E}\left[q^{\#}_{\mathcal{A}_1} | \mathbf{Succ}_n\right] < \varepsilon w n$.

Let $q^{\#}_{ver}$ be the number of oracle queries required to verify a single IPoW.

Lemma 1. *If the underlying IPoW is a (w', m, f)-IPoW and $q^{\#}_{ver} \leq \frac{1}{10} f(0)$ then for all $n \geq 1$ and every adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ that makes at most a polynomial number of queries, it holds that*

$$\mathbb{E}\left[q^{\#}_{\mathcal{A}_1} | \mathbf{Succ}_n\right] \geq \frac{8}{10} \cdot Tn \cdot f(0) \ .$$

Proof. We will construct an IPoW adversary, $\mathcal{A}^{(IPoW)} = (\mathcal{A}_1^{(IPoW)}, \mathcal{A}_2^{(IPow)})$, that makes the almost the same number of queries as $\mathcal{A}_1$, uses no space and outputs $\Omega(Tn)$ valid IPoWs.

1. $\mathcal{A}_1^{(IPoW)}$ does nothing.
2. $\mathcal{A}_2^{(IPoW)}$ runs $\mathcal{A}_1$, recording all the oracle queries.
3. When $\mathcal{A}_1$ sends $com_1, \ldots, com_n$, supposedly the roots of the Merkle trees, $\mathcal{A}_2^{(IPoW)}$ can reconstruct the complete trees by checking the recorded queries to see the preimage of each tree node.
4. $\mathcal{A}_2^{(IPoW)}$ then runs the IPoW verifier on each of the leaves.
5. $\mathcal{A}_2^{(IPoW)}$ outputs the valid IPoW leaves (note that for this attack, the communication between $\mathcal{A}_1$ and $\mathcal{A}_2$ is irrelevant, since both are executed by $\mathcal{A}_2^{(IPoW)}$).

Denote S the set of valid IPoW entries in the committed tables (i.e., the ones reconstructed by $\mathcal{A}_2^{(IPoW)}$). The list of table entries generated by $\mathcal{A}_2$ in a winning response to the challenge must be contained in S (except with negligible probability in the length of the Merkle oracle), otherwise either the IPoW verification would fail or the Merkle commitment verification would fail (in contradiction to the response being a "winning" response). We claim that this implies, in almost every instance in which $\mathcal{A}$ wins the PoST security game, that $|S| > \frac{9}{10} Tn$.

To see why, assume in contradiction that $|S| \leq \frac{9}{10} Tn$. Then there must be at least one table that does not have $\frac{9}{10} T$ valid entries. In particular, since all the verifiers accepted, there must be at least one accepting PoST whose corresponding table had less than $\frac{9}{10} T$. Let S' be the entries belonging to that table. The probability (over the choice of the random oracle) that for a given *nonce* all k entries $\mathrm{H}_1(nonce||ch||1), \ldots, \mathrm{H}_1(nonce||ch||k)$ are in S' is $(|S'|/T)^k \leq 2^{-k}$. Let $\mathbf{Succ}_{nonce}$ be the event that this test succeeded for *nonce*. Since entries of the random oracle are independent, the events $\mathbf{Succ}_{nonce}$ for different nonces are independent. Hence, the probability that an adversary can succeed with less than $2^k/k$ oracle queries to H_1 is bounded by a negligible function in k (using the Chernoff bound).

So except with negligible probability, if $\mathcal{A}$ is successful $\mathcal{A}_2^{(IPoW)}$ outputs $\frac{9}{10} Tn$ valid IPoWs. $\mathcal{A}_2^{(IPoW)}$ runs $\mathcal{A}_1$ once, making exactly the same number of

queries. In addition it runs the verifier on all the n tables' entries. Thus, by the IPoW soundness,

$$\mathbb{E}\left[q^{\#}_{\mathcal{A}_1}|\mathbf{Succ}_n\right] = \mathbb{E}\left[q^{\#}_{\mathcal{A}^{(IPoW_2)}}|\mathbf{Succ}\right] - q^{\#}_{ver}\cdot T\cdot n \geq \frac{9}{10}Tn\cdot f(0) - q^{\#}_{ver}\cdot T\cdot n \geq \frac{8}{10}Tn\cdot f(0)\,.$$

□

Lemma 2. *If the underlying IPoW is a (w', m, f)-IPoW then for all $n \geq 1$ and every adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ that makes at most a polynomial number of queries and uses s bits of storage*

$$\mathbb{E}\left[q^{\#}_{\mathcal{A}_2}|\mathbf{Succ}_{n'}\right] \geq \frac{1}{4}n'\cdot T\cdot f(s/(n'\cdot T/4))$$

Proof. We construct an IPoW adversary that runs $\mathcal{A}$ once, uses the same amount of space and makes the same number of queries to the work oracle. The IPoW adversary, $\mathcal{A}^{(IPoW)} = (\mathcal{A}_1^{(IPoW)}, \mathcal{A}_2^{(IPow)})$, works as follows:

1. $\mathcal{A}_1^{(IPoW)}$ runs $\mathcal{A}_1$. For every Merkle oracle query corresponding to an invalid IPoW, $\mathcal{A}_1^{(IPoW)}$ intercepts the oracle call and replaces the answer with a random string (this will cause it to be invalid with high probability). If asked again, it answers consistently. ($\mathcal{A}_1^{(IPoW)}$ can ask the IPoW verification oracle, since in the IPoW soundness attack we don't care about the number of queries made by $\mathcal{A}_1^{(IPoW)}$). The output of $\mathcal{A}_1^{(IPoW)}$ is σ (where σ is the output of $\mathcal{A}_1$).
2. $\mathcal{A}_2^{(IPoW)}$ runs $\mathcal{A}_2(\sigma)$, keeping track of queries to the oracle.
3. For each path-probe query (to H_2) made by $\mathcal{A}_2$, $\mathcal{A}_2^{(IPoW)}$ parses the query as a set of k table entries and their commitment openings, and verifies the commitment openings. Note that $\mathcal{A}_1^{(IPoW)}$ ensured that any validly opened entries are also valid IPoWs.
4. $\mathcal{A}_2^{(IPoW)}$ outputs the set of valid IPoWs collected during the execution of $\mathcal{A}_2$.

$\mathcal{A}_2^{(IPoW)}$ makes the same number of queries to the work oracle as $\mathcal{A}_2$, and $\mathcal{A}^{(IpoW)}$ uses the same amount of storage (plus an additional PRF key of size k). It remains to show the the set of valid IPoWs it outputs is at least of size $\frac{1}{4}n\cdot T$.

To do this, we require that for every PoST table for which $\mathcal{A}_2^{(IPoW)}$ reconstructed less half of the entries, $\mathcal{A}_2$ will fail to convince a verifier with high probability. Thus, there must be at least n' tables with $\frac{1}{2}T$ "good" entries.

Let S be the entries of one of the tables output by $\mathcal{A}_2$, such that $\mathcal{A}_2^{(IPoW)}$ reconstructed less than $\frac{1}{2}|S|$ of the entries. Since $\mathcal{A}_2^{(IPoW)}$ will reconstruct any entry that was a valid IPoW and was included in a path-query, this means $\mathcal{A}_2$ did not make path-queries about at least half of the entries in the table. However, in order to convince the verifier, $\mathcal{A}_2$ must output k "good" nonces—that is, paths for which $\mathrm{H}_2(\cdot)$ returned value less than $p^* = k/T$ and all corresponding IPoW

entries were valid. Consider the array $nonce_1 \ldots, nonce_k$ of nonces output for this table by $\mathcal{A}_2$. For each nonce, the probability of finding a good path with less than $\frac{1}{2p^*}$ queries to $\mathrm{H}_2(\cdot)$ is at most $\frac{1}{2}$. Since $\mathrm{H}_2(\cdot)$ queries are independent for different nonces, if there are more than $\frac{1}{2}k$ of the nonces for which less than $\frac{1}{2p^*}$ queries were made, the probability that all of the nonces are good is at most $2^{-k/2}$. (Note that $\mathrm{H}_2(\cdot)$ is entirely independent of the view of $\mathcal{A}_1$ (and hence of σ, since we used an internally simulated oracle in $\mathcal{A}_2$. This is ok because the challenge was chosen with high min-entropy, so the probability that $\mathcal{A}_1$ could have queried $\mathrm{H}_2(\cdot)$ on the same challenge prefix is negligible.)

Hence, we can assume that at least $\frac{1}{2p^*} \cdot \frac{k}{2} = \frac{k}{4p^*}$ path-probes were made. Denote $P_1, \ldots, P_\ell$ the paths queries, where we can think of each P_i as a set of indices). Suppose $P_1, \ldots, P_i$ were already generated. Note that every set $S \subseteq [T]$ and nonce $nonce$, the events $\{H(nonce||i) \mod T \in S\}_{i \in [k]}$ are i.i.d and hold with probability $|S|/T$). By the Chernoff inequality,

$$\Pr\left[|\{H(nonce||1) \mod T, \ldots, H(nonce||k) \mod T\} \cap S| > \frac{2k|S|}{T}\right] < e^{-k\frac{|S|}{3T}}.$$

Since by our assumption $\left|\bigcup_{j=1}^{i} P_j\right| < T/4$, for any nonce, the probability that the nonce generates a path P_{i+1} such that $\left|P_{i+1} \setminus \bigcup_{j=1}^{i} P_j\right| < k/2$ is at most $e^{-k/12}$. Thus, since $\mathcal{A}_1$ runs in expected polynomial time, except with negligible probability it must be that for every $i \in [k]$, at least $k/2$ new indices are added. So, $\left|\bigcup_{j=1}^{\frac{k}{4p^*}} P_j\right| > \frac{k}{2} \cdot \frac{k}{4p^*} \geq T/4$, so $\mathcal{A}^{(IPoW)}$ outputs at least $n' \cdot T/4$ valid IPoWs.

By the soundness property of the IPoW, and since $\mathcal{A}^{(IPoW)}$ uses the same space and makes the same number of queries as $\mathcal{A}$, we have that for $\mathcal{A}$, $\mathbb{E}[q|Success] \geq n' f((s)/n')$. (We note that the queries to random oracles that are independent of H (such as H_1, H_2 and the Merkle oracle) don't matter, since the IPoW soundness should still hold). □

Theorem 4. *A Table-PoST based on an (w', m, f)-IPoW with $q^{\#}_{ver} < \frac{1}{10}w'$ is a $(w' \cdot T, m \cdot T, \varepsilon = \frac{8}{10}\frac{f(0)}{w'}, f'(s) = \frac{T}{4} \cdot f(\frac{4}{T} \cdot s))$-PoST*

Proof. The honest work and storage requirements follow from the protocol constructing a table of T IPoWs. The value of ε follows directly from Lemma 1, while the value of f' follows from Lemma 2. □

Corollary 1. *The Table-PoST protocol, used with our construction of the hash-preimage IPoW gives a $(w' \cdot T, m \cdot T, \varepsilon = \frac{8}{160}, f'(s) = \frac{T}{64} \cdot w' \cdot 2^{-\frac{4}{T} \cdot s})$-PoST protocol*

Proof. By Theorem 2, our construction gives a $(w', \log w', \frac{1}{16} \cdot w' \cdot 2^{-s})$-IPoW. Together with Theorem 4, this implies a $(w' \cdot T, m \cdot T, \varepsilon = \frac{8}{160}, f'(s) = \frac{T}{64} \cdot w' \cdot 2^{-\frac{4}{T} \cdot s})$-PoST (We note that the constants in the analysis are not tight; in fact both ε and f' give better bounds in practice.) □

A.2 Hash-IPoW Soundness Proof (Proof of Corollary 2)

To prove Theorem 2, we will first show that this IPoW is indeed incompressible. This is captured in Corollary 2: For an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, let denote $\mathbb{E}_{ch,H}[q|\mathbf{Succ}]$ the expected number of queries made by $\mathcal{A}_2$, conditioned on the verifier accepting (where the expectation is over the random oracle). The following corollary lower-bounds this expectation as a function of the storage used by the adversary and the total number of IPoWs:

Lemma 3. *For every adversary, $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, such that $\mathcal{A}_1$ outputs at most $s < n \cdot \log w$ bits and all $n \geq 1$,*

$$\mathop{\mathbb{E}}_{H}[q|\mathbf{Succ}] \geq \frac{n \cdot w}{8} \cdot 2^{-s/n} \cdot 2^{-2/(n \Pr[\mathbf{Succ}])}$$

Proof. To simplify the proof, we'll assume that the hardness parameter for the hash PoW is a power of two (i.e., $w = 1/p = 2^m$ for some $m \in \mathbb{N}$).

The adversary is computationally unbounded, so we can assume w.l.o.g. that it is deterministic, and probabilities are only over the random string.

For the adversary to be successful, $\mathcal{A}_2$ must output n distinct challenges $ch_1, \ldots, ch_n$ and n outputs $x_1, \ldots, x_n$ such that $\forall i \in [n] : H(ch_i||x_i) < 2^{-m}$. Denote **Succ** the random variable indicating that this event occurred.

The compression algorithm is described in Protocol 2. Given a random string of length $2^\ell k$, it interprets the string as the random oracle, $H : \{0,1\}^\ell \mapsto \{0,1\}^k$ and uses an adversary that violates the corollary to compress the string. By observation, we can see that the corresponding decompression algorithm (Protocol 3) decodes perfectly with no errors.

Let Z be the storage size for the compressed data. Using the storage analysis in Protocol 2, when the adverary is successful and also makes few queries (i.e., $q \leq 2\,\mathbb{E}[q|\mathbf{Succ}] \wedge \mathbf{Succ}$ occurs), the data has compressed size at most $C = 2^\ell k + s + 1 - n(m - 2 - \log \frac{2\,\mathbb{E}[q|\mathbf{Succ}]}{n})$, while otherwise it's stored uncompressed with size $U = 2^\ell k + 1$.

Thus, the expected number of bits stored by the compression algorithm is at most

$$\begin{aligned}
\mathbb{E}[Z] &\leq \Pr\left[q \leq 2\,\mathbb{E}[q|\mathbf{Succ}] \wedge \mathbf{Succ}\right] \cdot C + (1 - \Pr\left[q \leq 2\,\mathbb{E}[q|\mathbf{Succ}] \wedge \mathbf{Succ}\right]) \cdot U \\
&= U + \Pr\left[q \leq 2\,\mathbb{E}[q|\mathbf{Succ}] \wedge \mathbf{Succ}\right] \cdot (C - U) \\
&= U - \Pr[\mathbf{Succ}] \Pr\left[q \leq 2\,\mathbb{E}[q|\mathbf{Succ}]|\mathbf{Succ}\right] \cdot (U - C)\ .
\end{aligned}$$

For every non-negative r.v. X, it holds that $\Pr[X > 2\,\mathbb{E}[X]] \leq \frac{1}{2}$. Since q is non-negative,

$$\Pr\left[q \leq 2\,\mathbb{E}[q|\mathbf{Succ}] \wedge \mathbf{Succ}|\mathbf{Succ}\right] = \Pr\left[q \leq 2\,\mathbb{E}[q|\mathbf{Succ}]|\mathbf{Succ}\right] \geq \frac{1}{2}\ .$$

Thus,

$$\begin{aligned}\mathbb{E}[Z] &= U - \Pr[\mathbf{Succ}] \Pr\left[q \leq 2\,\mathbb{E}[q|\mathbf{Succ}]|\mathbf{Succ}\right] \cdot (U - C) \\ &\leq U - \frac{1}{2}\Pr[\mathbf{Succ}] \cdot (U - C)\ .\end{aligned}$$

By Shannon's source-coding theorem, $\mathbb{E}[Z] \geq 2^{\ell}k = U - 1$. Therefore,

$$U - \frac{1}{2}\Pr[\mathbf{Succ}] \cdot (U - C) \geq U - 1$$

which implies

$$\frac{1}{2}\Pr[\mathbf{Succ}] \cdot (U - C) \leq 1 \Rightarrow U - C \leq 2/\Pr[\mathbf{Succ}]\ .$$

Expanding $U - C = n(m - 2 - \log \frac{2\,\mathbb{E}[q|\mathbf{Succ}]}{n}) - s$ we get

$$n(m - 2 - \log \frac{2\,\mathbb{E}[q \mid \mathbf{Succ}]}{n}) - s \leq 2/\Pr[\mathbf{Succ}]$$

which implies

$$\mathbb{E}[q \mid \mathbf{Succ}] \geq \frac{n \cdot w}{8} \cdot 2^{-s/n} \cdot 2^{-2/(n\Pr[\mathbf{Succ}])}$$

as required. □

Corollary 2. *For every adversary, $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, such that $\mathcal{A}_1$ succeeds with probability at least $1/2n$, outputs at most $s < n \cdot \log w$ bits and all $n \geq 1$,*

$$\mathop{\mathbb{E}}_{H}[q|\mathbf{Succ}] \geq \frac{n \cdot w}{16} \cdot 2^{-s/n}$$

We can now complete the proof:

Proof (of Theorem 2). The honest prover uses w' expected queries, by the setting of $p = 1/w'$ and stores $m = \log \frac{1}{p} = \log w'$ bits. By Corollary 2, this protocol is sound. □

Protocol 2 Compression algorithm

Run $\mathcal{A}_1$ to get σ. // Assume w.l.o.g that $|\sigma| = s$
Run $\mathcal{A}_2$ with σ and H as input.
Let $X = (ch_1||x_1, \ldots, ch_n||x_n)$ be the outputs of $\mathcal{A}_2$, sorted lexicographically.
Let $Q = (q_1, \ldots, q_{|Q|})$ be the set of oracle queries made by $\mathcal{A}_2$, sorted lexicographically.
if $|Q| \leq 2\,\mathbb{E}\,[q]$ and $\forall i$, the m MSBs of $H(ch_i||x_i)$ are all 0s **then** // the output of $\mathcal{A}_2$ verifies
    Let $X' = X \cap Q = (x'_1, \ldots, x'_{|X'|})$, the subset of outputs that were also queried.
    for all $j \in \{1, \ldots, |X'|\}$ **do**
        Denote $\mathbf{idx}(j)$ the index of x'_j in Q (i.e., $q_{\mathbf{idx}(j)} = x'_j$).
        Let $\Delta_j = \mathbf{idx}(j) - \mathbf{idx}(j-1)$ // we define $\mathbf{idx}(0) = 1$
    end for
    Let $\Delta_{|X'|+1} = 2\,\mathbb{E}\,[q] - \sum_{j=1}^{|X'|-1} \Delta_j$ // $\sum_{j=1}^{|X'|} \Delta_j = \mathbb{E}\,[q]$
    Output $(1, \sigma, \Delta_1, \ldots, \Delta_{|X'|}, \Delta_{|X'|+1}, H(q_1), \ldots, H(q_{|Q|}), H|_{X \setminus Q}, H|_{\neg(X \cup Q)})$
    - We will represent Δ_j in the following way:
        - $\left\lfloor \frac{\Delta_j}{2\,\mathbb{E}\,[q]/n} \right\rfloor$ represented in unary (between 0 and n one bits)
        - a zero bit.
        - Δ_j **mod** $(2\,\mathbb{E}\,[q]/n)$ represented in binary ($\log \frac{2\,\mathbb{E}\,[q]}{n}$ bits)

      Since $\sum_j \Delta_j \leq 2\,\mathbb{E}\,[q]$, the total number of bits in the unary representations is at most n. Thus, in total we use at most $n + |X'|(1 + \log \frac{2\,\mathbb{E}\,[q]}{n})$ bits.
    - We represent $H(q_i)$ as follows:
        - If $q_i \in X'$, we store the $k - m$ LSBs of $H(q_i)$
        - Otherwise, we store the full k bits.

      In total, this uses $|X'|(k-m) + (|Q| - |X'|)k$ bits.
    - We represent $H|_{X \setminus Q}$ by storing the $k - m$ LSBs of each entry. The entries are stored consecutively without padding. This uses $(n - |X'|)(k - m)$ bits.
    - We will represent $H|_{\neg(X \cup Q)}$ by storing the full entries. The entries are stored consecutively without padding. This uses $(2^\ell - n - |Q| + |X'|)k$ bits.

  All together, since $|X'| \leq n$, the length of the encoding is at most

$$\begin{aligned} Z &= 1 + s + n + |X'|(1 + \log \frac{2\,\mathbb{E}\,[q]}{n}) + |X'|(k-m) + \\ &\qquad (|Q| - |X'|)k + (n - |X'|)(k-m) + (2^\ell - n - |Q| + |X'|)k \\ &\leq 2^\ell k + s + 1 - n(m - 2 - \log \frac{2\,\mathbb{E}\,[q]}{n})\ . \end{aligned}$$

else
    Output $(0, H)$.
end if

Protocol 3 Decompression algorithm

1: **if** The stored data has the form $(0, H)$ **then**
2: Output H.
3: **else**
4: data has the form $(1, Bits)$. Treat $Bits$ as a stream.
5: Reconstruct $\hat{X} = \{i | q_i \in X'\}$ from $\Delta_1, \ldots, \Delta_{|X'|}$: $\hat{X}_i = \sum_j = 1^i \Delta_i$. (note: we know when we've reached $\Delta_{|X'|+1}$ when the sum is exactly $2\,\mathbb{E}\,[q]$)
6: Execute $\mathcal{A}_2$ with σ as input
– For the i^{th} query made by $\mathcal{A}_2(\sigma)$:
- If $i \in \hat{X}$ then reconstruct $H(q_i)$ by reading the $k-m$ next bits and treating them as a k-bit value with m zero MSBs
- If $i \in \hat{X}$ then reconstruct $H(q_i)$ by reading the k next bits

The execution will give Q and X as output.
7: Reconstruct $H_|X \setminus Q$ by reading the next $(n - |X'|)(k - m)$ bits and treating them as $(n - |X'|)$ values
8: Reconstruct $H|_{\neg(X \cup Q)}$ by reading the next $(2^\ell - (n + |Q| - |X'|))k$ bits.
9: **end if**