Simple Proofs of Space-Time and Rational Proofs of Storage

Tal Moran* Ilan Orlov†

Abstract

We introduce a new cryptographic primitive: Proofs of Space-Time (PoSTs) and construct an extremely simple, 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 much 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). Unlike previous constructions, our protocol allows incremental difficulty adjustment, which can gracefully handle increases in the price of storage compared to CPU work. In addition, we show how, in a crypto-currency context, the parameters of the scheme can be adjusted using a market-based mechanism, similar in spirit to the difficulty adjustment for PoW protocols.

1 Introduction

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 [9], for defending against denial-of-service attacks [23] and fairly recently, as the underlying mechanism for implementing a decentralized bulletin-board—this is the technical heart of the Bitcoin protocol [17].

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 [10, 5]. Park et al. also designed an alternative crypto-currency that is based on Proofs of Space [18], and several new crypto-currency companies are also basing their protocols on similar ideas [15, 1, 2].

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.

*IDC Herzliya. Email: talm@idc.ac.il. Supported by ISF grant no. 1790/13 and the Bar-Ilan Cyber-center

†Outbrain

[1]For the purposes of this paper, we use the formal definitions of [10]

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 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.

We note that though the definition of a PoS is insufficient to guarantee rational storage, the existing PoS *constructions* actually do achieve this under some parameters. However, this is more than just a definitional problem. In particular, in the existing PoS protocols [10, 5, 22, 3, 12, 21], 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 [18]. 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 spacetime over work; when

[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).

this is the case we say that a PoST is *Rationally Stored* (we give a formal definition in Section 2.2.4). 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!

Simple, 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 two simple candidate constructions based on the standard "hash preimage" PoW and on storing part of a single hash output. Our protocols and proofs use a very different technique than existing proofs of space, and are much simpler to implement. (We note that although the constructions are extremely simple, proving their security is non-trivial.)

Incremental Difficulty Adjustment. Since the relative price of CPU and storage may change over time, use of a PoST (or PoS) protocol in a crypto-currency setting could require adjusting the parameters (in particular, if the relative price of storage increases, it may no longer be rational to use storage as the preferred resource). In existing PoS constructions, this appears to require rerunning the entire initialization protocol. In contrast, our PoST construction supports simple *incremental* difficulty adjustment—that is, users only have to pay the marginal work cost between difficulty levels.

Market-Based Parameter Adjustment. A related issue when designing a crypto-currency based on PoST (or PoS) is deciding when and how to adjust the initialization difficulty. We show how to do this automatically via a *market-based* mechanism (similar in spirit to the difficulty adjustment in PoW-based crypto-currencies). The idea is to incentivize users to honestly report whether they are recomputing or storing data (see Section 7 for details), allowing us to build protocols that automatically increase the difficulty when the price of storage rises sufficiently (in which case we'd expect to see more users choosing computation over storage). The detection technique is general, and may be of independent interest—it can be applied to existing PoS constructions as well.

Different Parameter Regimes. In comparison with existing PoS constructions, 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. 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 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 efficiency doesn't preclude the use-cases we describe (e.g., even on a mid-range consumer HDD, sequential throughput is about 150MB/s; this means reading through a 100GB table in about 10 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 8)

1.2 Related Work

Random-Function-Inversion PoS A recent work by Abusalah et al. [3] shows how to construct a PoS protocol based on inverting a random function. This construction is significantly simpler than the pebbling-based constructions (although still more complex than our construction). However, the initialization difficulty is also fixed, and it does not seem trivial to increase initialization difficulty without at the same time increasing verification difficulty linearly, and it does not appear to support incremental difficulty adjustment. Hence it does not appear suitable for long intervals between proofs.

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 [13, 7, 6, 14, 20]. 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.

Proofs of Replication In a Proof of Replication [11], a party would like to prove that they are storing multiple redundant copies of a file. The PoRep definitions combine a PoS and a Proof of Retrievability. Similarly to the PoST definition, PoReps don't (and can't) guarantee that the prover actually stores redundant copies of the data, but instead make it an ε-Nash equilibrium (so a rational prover does not lose much by doing so). The existing constructions of PoReps depend on depth-robust graphs for the PoS and on sequential timing assumptions (the prover must respond to a challenge quickly, and the timing assumptions ensure that the prover cannot recompute its data in that time)

Memory-Hard Functions Loosely speaking, a memory-hard function is a function that requires a large amount of memory to evaluate [19, 4]. 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).

Filecoin Filecoin [15] is a crypto-currency protocol based on Proofs of Replication, whose underlying idea is to base the consensus algorithm resource on "useful" space. The Filecoin whitepaper also defines a "Proof of Spacetime",[3] however in their definitions the proof must include a proof of the elapsed time (requiring assumptions such sequential work timing assumptions). Moreover, their constructions make use of very heavy cryptographic machinery (such as zkSNARKS).

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 [16]. 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

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.

[3] We note that the our PoST definitions precede theirs.

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 $H^{(\text{work})}$.

Initialization Phase Both parties receive as input an id string $id \in \{0,1\}^*$. 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):

$$(\sigma_P, \sigma_V) \leftarrow \left\langle P_{\text{init}}^{H^{(\text{work})}}(id), V_{\text{init}}^{H^{(\text{work})}}(id) \right\rangle .$$

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:

$$(\cdot, out_V) \leftarrow \left\langle P_{\text{exec}}^{H^{(\text{work})}}(id, \sigma_P), V_{\text{exec}}^{H^{(\text{work})}}(id, \sigma_V) \right\rangle .$$

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.

2.2.1 PoST Parameters

A PoST has three parameters: w, the *Honest Initialization Work*, m, the *Honest Storage Space*, and f, the *Soundness Bound*,

Honest Initialization Work (w) This is 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 2.9 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.

[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 Storage Space (m) 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).

Definition 2.1 (PoST)**.** A protocol (P, V) as defined above is a (w, m, ε, f)-PoST if it satisfies the properties of completeness and f-soundness defined below.

2.2.2 Completeness

Definition 2.2 (PoST η-Completeness)**.** We say that a PoST is η-complete if for every $id \in \{0,1\}^{\text{poly}(k)}$ and every oracle $H^{(\text{work})}$,

$$\Pr\Big[out_V = 1 : (\sigma_P, \sigma_V) \leftarrow \Big\langle P_{\text{init}}^{H^{(\text{work})}}(id), V_{\text{init}}^{H^{(\text{work})}}(id)\Big\rangle, \\ (\cdot, out_V) \leftarrow \Big\langle P_{\text{exec}}^{H^{(\text{work})}}(id, \sigma_P), V_{\text{exec}}^{H^{(\text{work})}}(id, \sigma_V)\Big\rangle\Big] \geq \eta\,.$$

When $\varepsilon = 1$ completeness is perfect (in this case we sometimes omit the η).

2.2.3 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 2.3 (PoST (n, s, T_1, T_2)-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 $|\sigma_{\mathcal{A}}| \leq s$, $\mathcal{A}_1$ makes at most T_1 queries to the oracle $H^{(\text{work})}$, $\mathcal{A}_2$ makes at most T_2 queries to the oracle and all of the verifiers output 1 (we denote this event $\mathbf{Succ}_{n,s,T_1,T_2}$)

Definition 2.4 (PoST f-Soundness)**.** We say a PoST protocol is ε, f*-sound* if for all $T_1, T_2, s, \geq 0$ and all $n \geq 1$, for every adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ must satisfy the following conditions in the PoST security game:

1. *Rational Storage*: If $\mathcal{A}_1$ made less than $\varepsilon \cdot w$ queries to the work oracle, then the probability of success is negligible (in the security parameter).
2. *Space-Time Trade-Off*: $\Pr[\mathbf{Succ}_{n,s,T_1,T_2}] \leq f(n, s, T_1, T_2)$

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. Ideally using anything less we'd like the adversary to fail with overwhelming probability. However, this is impossible to achieve, even if it might be true for an individual PoST. This is because the adversary can always "forget" the entire data for a subset of the n instances, and rerun the initialization phase for those instances in the proof phase.

[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.

2.2.4 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 *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 crypto-currency setting, where profit is the main motive for participating).

Definition 2.5 ((γ, ε')-Rationally-Stored PoST)**.** We say a PoST is (γ, ε')-*rationally stored* if, when the real-world cost of a space unit is less than γ, then for any given resource budget C, the optimal execution strategy (maximizing the expected number of successful PoST proofs for that budget) requires that at least an ε'-fraction of the budget be used for storage).

We don't count the initialization cost in Definition 2.5. This is because it is only incurred once, while the cost of the execution phase is incurred repeatedly.

We can identify a sufficient condition for a PoST to be rationally stored:

Lemma 2.6. *If a (w, m, ε, f)-PoST is η-complete, and for all $C > 0$, $s < \varepsilon' \cdot C/\gamma$ it holds that*

$$\sum_{i=1}^{\infty} f(i, s, C - \gamma \cdot s) < \eta \cdot C/(\gamma \cdot m)$$

then it is (γ, ε')-Rationally-Stored.

(Note that we assume $f(i, s, T) \leq 1$ for all values if i,s and T — otherwise we use instead $f^*(i, s, T) = \min\{1, f(i, s, T)\}$.)

Proof. Denote $\#G$ the random variable for the number of successful PoST proofs produced by the adversary. Then

$$\begin{aligned}\mathbb{E}[\#G] &= \sum_{i=1}^{\infty} i \cdot \Pr[\#G = i] = \sum_{i=1}^{\infty} i \cdot \Big(\Pr[\#G \geq i] - \Pr[\#G \geq i+1]\Big) \\ &= \sum_{i=1}^{\infty} \Pr[\#G \geq i] \,.\end{aligned}$$

By the definition of f-soundness, for an adversary using s space and $C - \gamma \cdot s$ oracle queries, the expectation is thus bounded by

$$\mathbb{E}[\#G] \leq \sum_{i=1}^{\infty} f(i, s, C - \gamma \cdot s)$$

On the other hand, using the honest proof strategy, and allocating the entire C budget to space will give $C/(\gamma \cdot m)$ proofs, each successful with probability at least η, hence the expected number of successful proofs for the honest space-only strategy is $\eta \cdot C/(\gamma \cdot m)$.

Thus, the honest proof strategy generates, in expectation, more successful proofs (i.e., higher reward) than any adversarial strategy that spends less than an ε' fraction of its budget on storage space. □

Note that the adversary can always rerun the initialization phase instead of storing data, so for any η-complete, (w, m, ε, f)-PoST we must have $f(i, 0, i \cdot w) \geq \eta$, hence if $\gamma \cdot m > w$ the condition of Lemma 2.6 cannot be satisfied.

2.2.5 Comparison with the PoS definition

As we remarked in the introduction, an (N_0, N_1, T)-PoS does not give any formal 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 optimally-sound (w, m, f)-PoST can't guarantee a (x, x, w)-PoS, for any $x \in (0, w)$, since we don't

place any lower bound on the space required to generate a proof—the adversary can always trade space for polynomial work. Thus, the parameters are not truly comparable.

Note that even if we did add a space lower bound, similar to the PoS definition, in order to make use of it in practice one would have to add additional non-standard assumptions (such as timing assumptions); this is because the adversary can perform the space-time tradeoff at the level of entire PoS instances (e.g., generate n instances, but use space for only a single instance at a time).

Thus, 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 high storage costs and requires frequent proof phases (to prevent a space/time tradeoff), while the PoST definition does allow long periods of elapsed time between proofs (with a suitably hard initialization step), but relies on the rationality of the adversary to enforce use of storage rather than work.

2.2.6 Non-Interactive Proofs of SpaceTime (NIPSTs)

Sigma-PoST A *Sigma-PoST* is a PoST scheme that has the form of a Sigma-protocol: $P_{\text{init}}(id)$ sends a single commitment message to the verifier; V_{init} responds with a random challenge string, after which P_{init} sends a single response message. For the execution phase, the commitment message is the same as the initialization commitment (hence does not need to be resent); V_{exec} sends a random challenge string, and P_{exec} responds in turn with a single message.

We note that our PoST construction is a Sigma-PoST.

Making Sigma-PoSTs Non-Interactive The initialization phase of a Sigma-PoST can be made non-interactive in the random oracle model by using the Fiat-Shamir heuristic (replacing the verifier's response with a hash of the commitment message). However, interaction cannot be removed entirely—the execution phase requires a challenge that cannot be predicted by the prover at initialization time—hence, under standard assumptions it cannot be solely a function of the prover's inputs.

Using Proofs of Sequential Work By introducing a sequential timing assumption, we *can* make the proof entirely non-interactive; the idea is to use the output of the initialization phase (or the previous execution phase if we're running multiple times) as the input to a publicly-verifiable *proof of sequential work* (PoSW). We can then use (a hash of) the output of the PoSW as the challenge to the execution phase. If we assume a lower bound on the elapsed time for an adversary to perform a given amount of sequential work, this construction ensures that the adversary must have used sufficient spacetime resources between the initialization and execution phases.

This NIPST construction appears to violate our main goal—it requires continuous CPU work even for an honest user. The trick is that a *single* PoSW instance can be shared between an arbitrary number of provers, so the *amortized* CPU cost vanishes as the number of users grows. Instead of using the previous proof directly as the input to the PoSW, we create a Merkle tree whose leaves are the inputs from each prover, and use the root of the tree as the input to the single, shared PoSW.

The full NIPST consists of (1) the initialization phase output, (2) a Merkle path from the output to the root of the tree, (3) a PoSW whose input is the Merkle root and (4) the execution phase proof, with the PoSW as the challenge.

We note that some PoSW constructions (such as that of Cohen and Pietrzak [8]) don't have *unique* proofs; an adversary can generate multiple different proofs for the same input that will all be accepted by a verifier. When used in a NIPST, this means the PoST execution-phase challenges come from a distribution that can be biased by the adversary. However, our PoST construction can handle this as long as the distribution has enough min-entropy (which must be the case, since otherwise an adversary could solve the PoSW by trying to guess the result and running the verifier to check—this can be done in parallel, so would violate the sequential work security of the PoSW).

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 recomputing 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 τ 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.

A Simple Solution Our construction overcomes this problem by forcing the prover to commit to the entire table *at the time of the challenge*, and only then learn the random entries to be sent back (this is made non-interactive using the Fiat-Shamir heuristic). Intuitively, the prover is forced to either reconstruct a large fraction of the table (in which case it must either store many IPoWs, or recompute them), or spend a lot of computational work trying to find a commitment that will produce a "good" challenge. By setting the parameters correctly, we can ensure that in the second case the amount of work the prover must do is more than the initialization cost (see Section 3 for details).

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).

2.4.1 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 2.7 ((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 2.8) and f-sound (c.f. Definition 2.9)

Definition 2.8 (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 2.9 (IPoW $f(n, s, T)$-Soundness)**.** We say $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ is an $[n, s, T]$-adversary if $\mathcal{A}_1$ outputs a string σ with length $|\sigma| \leq s$, while $\mathcal{A}_2$ gets σ as input, makes at most T queries to the random oracle and outputs n pairs $(ch_1, \pi_1, \ldots, ch_n, \pi_n)$.

Denote **Succ** the event (over the randomness of $\mathcal{A}$ 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 $n \geq 1$, $s \geq 0$ and $T \geq 0$

$$\Pr[\mathbf{Succ}] < f(n, s, T)$$

Note that we don't restrict the number of queries $\mathcal{A}_1$ makes to the oracle.

As in the PoST definition, this condition bounds the trade-off between space-time and work for the IPoW adversary. Ideally, we'd like f to be negligible when $s < n \cdot m$ and $T < n \cdot w'$ (this implies that the adversary must either store the same amount as the honest prover, or do enough work to reconstruct the proof from scratch). Unfortunately, we can't hope to achieve this; for example, for any $i \in (0, n)$. if an adversary stores only i IPoWs, and reconstructs the remaining $n - i$, it will have overwhelming probability of success while storing $i \cdot m$ bits and doing $(n - i) \cdot w'$ work. Moreover, the adversary can always "forget" j bits of storage and guess them correctly with 2^{-j} probability. Thus, in any f-sound IPoW, we must have, for all $j \geq 0$ and $i \in (0, n)$ that $f(n, i \cdot m - j, (n - i) \cdot w', T) \geq 2^{-j} - \varepsilon$ for some negligible ε (that depends on the completeness of the protocol).

3 Our Simple PoST Construction: The Details

Formally, we describe the protocol in the presence of two types of random oracles, a "work" oracle $H^{(\text{work})}$ and "Merkle" oracles H_i (for $i \neq j$, H_i and H_j are independent random oracles).[6]. We assume the work oracle has a much higher cost than the calls to the Merkle oracles (in implementation, we can think of the Merkle oracles as a single iteration of a fast hash function, while the work oracle can be implemented by a slower hash function or multiple sequential iterations). In the analysis, we track the number of calls separately, using T to denote the number of calls to $H^{(\text{work})}$ and T^* the number of calls to the Merkle oracles.

The formal PoST protocol description appears as Protocol 1. To construct it, we use a (w', m, f)-IPoW. (We construct two oracle-based IPoW schemes in Section 4.)

The soundness of our Simple PoST protocol is summarized in the following theorem. (For our construction we allow $\mathcal{A}_1$ unbounded access to the work oracle, so don't include a T_1 parameter.)

Theorem 3.1 (PoST Soundness)**.** *Let k_{ch} be the min-entropy of the distribution from which PoST challenges are sampled. The Simple PoST protocol, instantiated with an f'-sound IPoW, is f-sound for*

$$\begin{aligned} f(n, s, T_1^*, (T, T^*)) = \min\{ & \\ & \min_{\varepsilon \in (0,1)} \left\{ f'(\varepsilon \cdot n \cdot \tau, s + n \cdot (k_H + \log T^* + \tau), T) + T^* \cdot \varepsilon^k \right\} + (T^*)^2 \cdot 2^{-k_H} + T_1^* \cdot 2^{-k_{ch}} , \\ & 2^{-k_H \cdot (n - \max\{T_1^*, T^*\})} \} \end{aligned}$$

(Note that the second term in the outer min is relevant only when $n > \max\left\{T_1^*, T^*\right\}$.)

Corollary 3.2. *When instantiated with the m-Partial-Hash IPoW, the Simple PoST is f-sound for*

$$\begin{aligned} f(n, s, T_1^*, (T, T^*)) = \min_{\varepsilon \in (0,1)} \left\{ 2^{-(\varepsilon \cdot n \cdot \tau \cdot m - (T \cdot m + s + n \cdot (k_H + \log T^* + \tau))} + T^* \cdot \varepsilon^k \right\} + \\ (T^*)^2 \cdot 2^{-k_H} + T_1^* \cdot 2^{-k_{ch}} . \end{aligned}$$

3.1 Security Proof

Proof Proof of Theorem 3.1. Let $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ be an adversary that wins the $(n, s, T_1^*, (T, T^*))$-PoST security game with probability p. For every $\varepsilon < 1$, we can use $\mathcal{A}$ to construct an IPoW adversary $\mathcal{A}^{(IPoW)} = (\mathcal{A}_1^{(IPoW)}, \mathcal{A}_2^{(IPoW)})$ as follows:

Let $ch_1, \ldots, ch_n$ be random challenges independently selected from a distribution with min-entropy k_{ch} [7]

IPoW Adversary Initialization ($\mathcal{A}_1^{(IPoW)}$):

1. Execute $\mathcal{A}_1$, recording the n ids $id_1, \ldots, id_n$ and storing its output σ.

[6] 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., $\mathrm{H}_i(x) = \mathrm{H}(i \| x)$.)

[7] These can be chosen by hardwiring a seed in the code of both $\mathcal{A}_1^{(IPoW)}$ and $\mathcal{A}_2^{(IPoW)}$, and computing ch_i using the Merkle oracle, which is not counted against the query budget of $\mathcal{A}_2^{(IPoW)}$.

Protocol 1 SIMPLE-POST

Public Parameters: k_H: hash output size, k: security parameter, τ: table size and IPoW (ch) is a (w', m, f)-IPoW.

Storing Phase: (Performed by the prover P)

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

1. Generate an array G of size τ as follows:
   For each $0 \leq i < \tau$, set $G[i] \stackrel{\text{def}}{=} \text{IPoW}(id \| i)$ (where the IPoW is given access to $H^{(\text{work})}$ as its underlying work oracle).
2. Run the proof phase with fixed challenge 0.
3. Publish the string id and the initial proof.

Proof Phase: (Performed by the prover P)

Upon receiving a challenge ch from the verifier V:

1: 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 H_{ch} on the concatenation of its children's labels. Let com be the root label.
2: parse $\text{H}_{ch}(com)$ as a set of k indices $(i_1, \ldots, i_k) \in \{0, \tau - 1\}^k$.
3: Let π_j be the Merkle path from the table entry $G[i_j]$ to the root com // The first element of π_j is the table entry itself
4: Output $com, \pi_1, \ldots, \pi_k$. // This can be made more communication efficient by eliminating common labels

The honest prover does not need any calls to $H^{(\text{work})}$, but needs up to 2τ calls to the Merkle oracle (or temporary space to store the Merkle tree).

Proof Phase: (Performed by the verifier V)

Generate a random challenge ch and send it to the prover. Wait to receive the list $com, \pi_1 \ldots, \pi_k$

1: parse $\text{H}_{ch}(com)$ as a set of k indices $(i_1, \ldots, i_k) \in \{0, \tau - 1\}^k$.
2: **for all** $j \in \{1, \ldots, k\}$ **do**
3: Verify that $G[i_j]$ (the first element of π_j) is a valid IPoW for the challenge $id \| i_j$ (using the oracle $H^{(\text{work})}$).
4: Verify that π_j is a valid Merkle path from the leaf i_j to the root com (using the oracle H_{ch}).
5: **end for**

2. Execute $\mathcal{A}_2$, with input $id_1, \ldots, id_n, \sigma$ and challenges $ch_1, \ldots, ch_n$. While executing, keep track of all calls to H_{ch_i}. Denote $com_1, \ldots, com_n$ the first elements of each $\mathcal{A}_2$ proof (which, for an honest prover, would each correspond to the root of a merkle tree).

3. For all $i \in [n]$:

   (a) Denote Q_i the set of queries to H_{ch_i}.

   (b) For every $q \in Q_i$, attempt to reconstruct a merkle tree with root $\mathrm{H}_{ch_i}(q)$. Obviously, this may not be possible for every query q, and even when possible may not result in a full tree. We will say a leaf (i, j) *exists* for q if some subset of Q_i comprises a valid Merkle path from the leaf j to the root $\mathrm{H}_{ch_i}(q)$. (Note that the reconstruction doesn't make any *additional* calls to the Merkle oracle, it just uses the stored results.)

   (c) For all q, and every existing leaf (i, j) for q, run the IPoW verifier with challenge $id_i \| j$ to check if the leaf is a valid IPoW proof. In this case, we say the leaf (i, j) is *valid* for q.

   (d) We say a query q is ε-*good* if there exist $\varepsilon \cdot \tau$ different leaves that are valid for q.

   (e) If there does not exist an ε-good query in Q_i, output $\perp$ and abort. Otherwise, denote g_i the index of the first ε-good query in Q_i, and let v_i be a bit-vector indicating the valid leaves ($v_{i,j} = 1$ iff (i, j) is a valid leaf for q_{g_i}).

4. Output $(\sigma, id_1, \ldots, id_n, g_1, \ldots, g_n, v_1, \ldots, v_n)$.

Note that the output length for $\mathcal{A}_1^{(IPoW)}$ is $s + n \cdot k_H + n \cdot \log T^* + n \cdot \tau = s + n \cdot (k_H + \log T^* + \tau)$ bits (since for all i, $|Q_i| < T^*$, and assuming, w.l.o.g, that the id size is k_H—we can always use a hash of the id if its larger).

IPoW Adversary Prover ($\mathcal{A}_2^{(IPoW)}$):

1. Run Steps 2 and 3 from the execution of $\mathcal{A}_1^{(IPoW)}$.

2. For each $i \in [n]$, reconstruct the Merkle tree rooted at $\mathrm{H}_{ch_i}(q_{g_i})$ and for every valid leaf (i, j), as indicated by v_i, output $id_i \| j$ as an IPoW challenge and leaf (i, j) as the corresponding proof.

Note that $\mathcal{A}_2^{(IPoW)}$ makes at most T calls to the work oracle and T^* calls to the Merkle oracle, since it executes $\mathcal{A}_2$ exactly once.

When $\mathcal{A}_2^{(IPoW)}$ succeeds, we're guaranteed that for each of the n challenges it can extract an ε-fraction of valid leaves, hence it outputs at least $\varepsilon \cdot n \cdot \tau$ valid IPoW proofs.

The storage space it requires is at most $s + n \cdot (k_H + \log T^* + \tau)$ bits. Thus, $\mathcal{A}^{(IPoW)}$ is an $(\varepsilon \cdot n \cdot \tau, s + n \cdot (k_H + \log T^* + \tau), T)$-IPoW adversary.

IPoW Adversary Success Probability: To bound the probability of success, we first rule out two "catastrophic" events:

- $\mathcal{A}_1$ makes a query to H_{ch_i}. Since $\mathcal{A}_1$ makes at most T_1^* queries in total to the Merkle oracles, and ch_i is chosen from a distribution with min-entropy k_{ch}, the probability of this occurring for challenge ch_i is at most $T_1^* \cdot 2^{-k_{ch}}$.
- $\mathcal{A}_2$ finds a collision in H_{ch_i} for some i. Since the Merkle oracle has output length k_H, and $\mathcal{A}_2$ makes at most T^* queries to H_{ch_i}, by the Birthday Bound the probability of finding any collision is less than $(T^*)^2 \cdot 2^{-k_H}$.

Now, consider instance i of the PoST proofs generated by $\mathcal{A}_2$. We claim that unless $p < 2^{-k_H}$, com_i must be the result of a query $\mathcal{A}_2$ makes to H_{ch_i}. To see this, recall that we assume $\mathcal{A}_1$ did not query H_{ch_i} on any input. Thus, if $\mathcal{A}_2$ did not receive com_i as the result of an oracle query, the probability that it can generate a valid Merkle path that terminates at com_i is at most 2^{-k_H}.

Since $\mathcal{A}_2$ can make at most T^* Merkle queries, each execution of $\mathcal{A}_2$ can have at most T^* potential Merkle roots for instance i.

Denote $\mathbf{Bad}_i$ the event that there are no ε-good queries in Q_i. Denote $\mathbf{Succ}$ the event that $\mathcal{A}_2$ is successful (for all n instances). We claim that for all $i, T_1^*, \ldots, T_n^*$ such that $\Pr\left[\mathbf{Bad}_i \wedge_{j=1}^{n} |Q_j| = T_j^*\right] > 0$, it holds that

$$\Pr\left[\mathbf{Succ} \,\middle|\, \mathbf{Bad}_i \bigwedge_{j=1}^{n} |Q_j| = T_j^*\right] < T_i^* \cdot \varepsilon^k \,.$$

To see this, consider an execution of $\mathcal{A}_2$. In order for $\mathcal{A}_2$ to be successful, it must output a good PoST proof for instance id_i. This means it must output a Merkle root com_i and the k merkle paths from valid leaves that are selected by $\mathrm{H}_{ch_i}(com_i)$.

For every new query q_i made by $\mathcal{A}_2$, conditioned on $\mathbf{Bad}_i$ the probability that $\mathrm{H}_{ch_i}(q_i)$ selects k valid leaves in *any* Merkle tree in $\mathcal{A}_2$'s view is at most ε^k; this is because, conditioning on $\mathbf{Bad}_i$, no Merkle tree in $\mathcal{A}_2$'s view has more than an ε-fraction of valid leaves. Since q_i has not been previously queried, $\mathrm{H}_{ch_i}(q_i)$ is independent of the view up to that point, hence the probability that k random indices are all valid is at most ε^k. Since there are exactly T_i^* queries to H_{ch_i}, the claim follows by the union bound.

Denote $\mathbf{Bad} = \mathbf{Bad}_1 \vee \cdots \vee \mathbf{Bad}_n$ the event that for some i there did not exist an ε-good query. Since $\mathcal{A}$ is bounded by T^* queries to the Merkle oracles, it must hold that $\sum_{i=1}^{n} |Q_i| \leq T^*$. Thus,

$$
\begin{aligned}
&\Pr[\mathbf{Succ} \wedge \mathbf{Bad}] \\
&= \Pr\left[\mathbf{Succ} \wedge \mathbf{Bad} \wedge \sum_{i=1}^{n} |Q_i| \leq T^*\right] \\
&= \Pr\left[\mathbf{Succ} \wedge (\mathbf{Bad}_1 \vee \cdots \vee \mathbf{Bad}_n) \wedge \sum_{i=1}^{n} |Q_i| \leq T^*\right] \\
&= \sum_{T_1^*,\ldots,T_n^*} \Pr\left[\bigwedge_{i=1}^{n} |Q_i| = T_i^*\right] \cdot \\
&\qquad\qquad \Pr\left[\mathbf{Succ} \wedge (\mathbf{Bad}_1 \vee \cdots \vee \mathbf{Bad}_n) \wedge \sum_{i=1}^{n} |Q_i| \leq T^* \,\middle|\, \bigwedge_{i=1}^{n} |Q_i| = T_i^*\right] \\
&= \sum_{\substack{T_1^*,\ldots,T_n^* | \\ \sum_{i=1}^{n} T_i^* \leq T^*}} \Pr\left[\bigwedge_{i=1}^{n} |Q_i| = T_i^*\right] \Pr\left[\mathbf{Succ} \wedge (\mathbf{Bad}_1 \vee \cdots \vee \mathbf{Bad}_n) \,\middle|\, \bigwedge_{i=1}^{n} |Q_i| = T_i^*\right]
\end{aligned}
$$

By the union bound,

$$
\leq \sum_{\substack{T_1^*,\ldots,T_n^* | \\ \sum_{i=1}^{n} T_i^* \leq T^*}} \Pr\left[\bigwedge_{i=1}^{n} |Q_i| = T_i^*\right] \sum_{i=1}^{n} \Pr\left[\mathbf{Succ} \wedge \mathbf{Bad}_i \,\middle|\, \bigwedge_{j=1}^{n} |Q_j| = T_j^*\right]
$$

By the definition of conditional probability,

$$
\begin{aligned}
&= \sum_{\substack{T_1^*,\ldots,T_n^* | \\ \sum_{i=1}^{n} T_i^* \leq T^*}} \Pr\left[\bigwedge_{i=1}^{n} |Q_i| = T_i^*\right] \cdot \\
&\qquad\qquad \sum_{i=1}^{n} \left(\Pr\left[\mathbf{Bad}_i \,\middle|\, \bigwedge_{j=1}^{n} |Q_j| = T_j^*\right] \Pr\left[\mathbf{Succ} \,\middle|\, \mathbf{Bad}_i \bigwedge_{j=1}^{n} |Q_j| = T_j^*\right]\right)
\end{aligned}
$$

Since $\Pr[\mathbf{Bad}_i] \leq 1$,

$$
\leq \sum_{\substack{T_1^*,\ldots,T_n^* | \\ \sum_{i=1}^{n} T_i^* \leq T^*}} \Pr\left[\bigwedge_{i=1}^{n} |Q_i| = T_i^*\right] \sum_{i=1}^{n} \Pr\left[\mathbf{Succ} \,\middle|\, \mathbf{Bad}_i \bigwedge_{j=1}^{n} |Q_j| = T_j^*\right]
$$

By our bound on $\Pr[\mathbf{Succ}|\mathbf{Bad}_i]$ above,

$$\begin{aligned}
&\leq \sum_{\substack{T_1^*,\ldots,T_n^* | \\ \sum_{i=1}^n T_i^* \leq T^*}} \Pr\left[\bigwedge_{i=1}^n |Q_i| = T_i^*\right] \sum_{i=1}^n \left(T_i^* \cdot \varepsilon^k\right) \\
&= \varepsilon^k \sum_{\substack{T_1^*,\ldots,T_n^* | \\ \sum_{i=1}^n T_i^* \leq T^*}} \Pr\left[\bigwedge_{i=1}^n |Q_i| = T_i^*\right] \sum_{i=1}^n T_i^* \\
&\leq \varepsilon^k \sum_{\substack{T_1^*,\ldots,T_n^* | \\ \sum_{i=1}^n T_i^* \leq T^*}} \Pr\left[\bigwedge_{i=1}^n |Q_i| = T_i^*\right] T^* \\
&\leq T^* \cdot \varepsilon^k
\end{aligned}$$

Therefore

$$\Pr[\neg\mathbf{Bad}] \geq \Pr[\mathbf{Succ} \wedge \neg\mathbf{Bad}] = \Pr[\mathbf{Succ}] - \Pr[\mathbf{Succ} \wedge \mathbf{Bad}] \geq p - T^* \cdot \varepsilon^k$$

Note that if the event **Bad** did not occur, and neither catastrophic event occurred, then $\mathcal{A}_1^{(IPoW)}$ does not abort and $\mathcal{A}_2^{(IPoW)}$ is guaranteed to be successful.

Since $\mathcal{A}^{(IPoW)}$ is a $(\varepsilon \cdot n \cdot \tau, s + n \cdot (k_H + \log T^* + \tau), T)$-IPoW adversary that succeeds with probability $p - T^* \cdot \varepsilon^k$, by the f'-soundness of the IPoW it follows that $p < f'(\varepsilon \cdot n \cdot \tau, s + n \cdot (k_H + \log T^* + \tau), T) + T^* \cdot \varepsilon^k + (T^*)^2 \cdot 2^{-k_H} + T_1^* \cdot 2^{-k_{ch}}$.

Finally, note that if $n > \max\left\{T_1^*, T^*\right\}$, then there are at least $n - \max\left\{T_1^*, T^*\right\}$ challenges which the adversary did not query at all; in this case its success probability is bounded by $2^{-k_H \cdot (n - \max\{T_1^*, T^*\})}$

□

4 Hash-Preimage IPoW

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

$$H^{(\text{work})}(ch \| x) < p \tag{4.1}$$

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).[8]

Formally,

[8] 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.

Definition 4.1 (w'-Hash-Preimage IPoW)**.** The honest prover and verifier are defined as follows: Set $p = 1/w'$.

Prover Given a challenge y, calls $H^{(\text{work})}$ 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 $H^{(\text{work})}(y\|x) < p$.

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 $H^{(\text{work})}(y\|z\|\pi) < p$ (where π is zero-padded to the maximum length).

The security of the Hash-Preimage IPoW is summarized in the following theorem:

Theorem 4.2. *The w'-hash-preimage protocol is a $(w', \log w', f)$-IPoW for $f(n, s, T) = 2^{-(n\log w' - s - n(2+\log\lceil T/n\rceil))}$.*

(The proof appears in Section 6.1.)

5 Partial Hash IPoW

Our choice of parameters for the IPoW is constrained by several real-world variables:

- The maximal time period between proofs that we would like to support
- The amount of storage we would like to fill
- The cost of storage per time period
- The cost of a hash invocation
- The maximum cost we can tolerate for PoST initialization.

For the Hash Preimage IPoW, given a maximum initialization cost and the cost of a hash invocation, we can upper bound the amount of storage we can fill: each hash invocation can "contribute" at most a single bit to the total storage (this is because the amount of space needed to store a single Hash-Preimage IPoW is logarithmic in the expected number of hash invocations needed to generate the proof; hence the largest space is taken when each proof requires on average only a single invocation).

If we would like to fill more space without increasing our initialization cost, we need to use a different IPoW.

The Partial Hash IPoW is a simple solution that can fill up to k bits per hash invocation (but *at least* one bit per invocation). In this case, the amount of work per IPoW is always a single hash invocation, as is the verification cost. We parameterize with the amount of space required to store an IPoW.

Formally,

Definition 5.1 (m-Partial-Hash IPoW)**.** The honest prover and verifier are defined as follows (where m is the space required to store an IPoW for the honest user):

Prover Given a challenge y, calls $H^{(\text{work})}(y)$ and returns as the proof the m least-significant bits of $H^{(\text{work})}(y)$.

Verifier Given challenge y and proof π, verifies that π consists of the m least-significant bits of $H^{(\text{work})}(y)$.

The security of the Partial-Hash IPoW is summarized in the following theorem:

Theorem 5.2. *The m-partial-hash IPoW protocol is a $(1, m, f(n, s, T) = 2^{-(n\cdot m-(T\cdot m+s))})$-IPoW.*

(The proof appears in Section 6.2.)

6 IPoW Security Analysis

In the proofs of security for both of our IPoW schemes, we use the following simple claim bounding the probability to compress a random string.

Let $(Compress, Decompress)$ be an arbitrary pair of probabilistic algorithms (possibly computationally unbounded), such that $Compress : \{0,1\}^k \mapsto \left\{\{0,1\}^{k-m}, bot\right\}$ and $Decompress : \{0,1\}^{k-m} \mapsto \{0,1\}^k$, and for all $(x,y) \in \{0,1\}^k \times \{0,1\}^{k-m}$, if $y \in \Im(Compress(x))$ then $Decompress(y) = x$. (That is, if $Compress$ "succeeds" then decompression is perfect).

Claim 6.1. *Let U_k be a uniformly selected from $\{0,1\}^k$. Then $\Pr\left[Compress(U_k) \neq \perp\right] \leq 2^{-m}$.*

Proof. Denote $Y = \Im(Compress) \setminus \{\perp\}$. Then $|Y| \leq 2^{k-m}$. Note that since decompression is perfect, for any $y \in Y$ there can be only a single pre-image (if we have $x_1 \neq x_2$ such that $Compress(x_1) = y = Compress(x_2)$, then for at least one of them $Decompress(y)$ will fail with non-zero probability). Let $X = \{x | Compress(x) \in Y\}$. Then $|X| \leq 2^{k-m}$. By the definitions of p and X, $p = \Pr\left[Compress(U_k) \neq \perp\right] = \Pr\left[U_k \in X\right]$, but since $|X| \leq 2^{k-m}$ and U_k is uniform, we have $p \leq 2^{-m}$ □

6.1 Proof of Theorem 4.2

Proof. The honest prover uses w' expected queries, by the setting of $p = 1/w'$ and stores $\log \frac{1}{p} = \log w'$ bits. Given an (n, s, T)-adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ that succeeds with probability p, we can construct a compression algorithm as described in Protocols 2 and 3.

Protocol 2 Hash-IPoW Decompression algorithm

1: **function** DECOMPRESS(Z)
2: Parse Z as $(\sigma, \Delta_1, \ldots, \Delta_{|X'|}, \Delta_{|X'|+1}, H(q_1), \ldots, H(q_{T \cdot n}), H|_{X \setminus Q}, H|_{\neg(X \cup Q)})$
3: 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 $T \cdot n$)
4: 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 $i \notin \hat{X}$ then reconstruct $H(q_i)$ by reading the k next bits

The execution will give Q and X as output.
5: Reconstruct $H|X \setminus Q$ by reading the next $(n - |X'|)(k - m)$ bits and treating them as $(n - |X'|)$ values
6: Reconstruct $H|_{\neg(X \cup Q)}$ by reading the next $(2^\ell - (n + T \cdot n - |X'|))k$ bits.
7: **end function**

This algorithm can, with probability p, compress a string of length $2^\ell \cdot k_H$ into a string of length $2^\ell k_H - (n \log w' - s - n(2 + \log \lceil T/n \rceil))$ (the analysis of the encoding length appears in the algorithm description). Thus, by Claim 6.1, we must have $p \leq 2^{-(n \log w' - s - n(2 + \log \lceil T/n \rceil))}$. □

6.2 Proof of Theorem 5.2

Proof. The space requirements for the honest prover follow by inspection, as does the perfect completeness.

To prove soundness, assume that there exists an (n, s, T)-adversary that succeeds in convincing a verifier with probability p. We construct a compression algorithm for random strings based on this adversary.

The compression protocol appears in Protocol 4. When successful, it encodes a string of length $2^\ell \cdot k_H$ (parsed as

Protocol 3 Hash-IPoW Compression algorithm

function COMPRESS(H) // Treat $H \in \{0,1\}^{2^\ell \cdot k}$ as the truth table of a function: $H : \{0,1\}^\ell \mapsto \{0,1\}^k$
  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 $Q = (q_1, \ldots, q_T)$ be the set of oracle queries made by $\mathcal{A}_2$, sorted lexicographically. (We can assume w.l.o.g. that $|Q| = T$.)
  if $\forall i$, the $\log w'$ MSBs of $H(ch_i \| x_i)$ are all 0s **then** // the output of $\mathcal{A}_2$ verifies; occurs w.p. p
    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 $\Delta_{|X'|+1} = T - \sum_{j=1}^{|X'|-1} \Delta_j$ // $\sum_{j=1}^{|X'|} \Delta_j = T$
    return $(\sigma, \Delta_1, \ldots, \Delta_{|X'|}, \Delta_{|X'|+1}, H(q_1), \ldots, H(q_T), H|_{X \setminus Q}, H|_{\neg(X \cup Q)})$

- We will represent Δ_j in the following way:
  - $\left\lfloor \frac{\Delta_j}{\lceil T/n \rceil} \right\rfloor$ represented in unary (between 0 and $\lceil T/n \rceil$ one bits)
  - a zero bit.
  - Δ_j **mod** $(\lceil T/n \rceil)$ represented in binary ($\log \lceil T/n \rceil$ bits)

  Since $\sum_j \Delta_j \le T$, the total number of bits in the unary representations is at most n. Thus, in total we use at most $n + |X'|(1 + \log \lceil T/n \rceil)$ bits.
- We represent $H(q_i)$ as follows:
  - If $q_i \in X'$, we store the $k - \log w'$ LSBs of $H(q_i)$
  - Otherwise, we store the full k bits.

  In total, this uses $|X'|(k - \log w') + (|Q| - |X'|)k$ bits.
- We represent $H|_{X \setminus Q}$ by storing the $k - \log w'$ LSBs of each entry. The entries are stored consecutively without padding. This uses $(n - |X'|)(k - \log w')$ 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 $|X'| \le n$, the length of the encoding is at most

$$\begin{aligned} Z &= s + n + |X'|(1 + \log \lceil T/n \rceil) + |X'|(k - \log w') + \\ &\qquad (|Q| - |X'|)k + (n - |X'|)(k - \log w') + (2^\ell - n - |Q| + |X'|)k \\ &\le 2^\ell k - (n \log w' - s - n(2 + \log \lceil T/n \rceil)) \,. \end{aligned}$$

  else
    return $\perp$
  end if
end function

the truth function of a random oracle) to a string of length:

$$\begin{aligned} Y &= s + T \cdot k_H + |X_q \setminus Q| \cdot (k_H - m) + \left(2^\ell - T - |X_q \setminus Q|\right) k_H \\ &= s + T \cdot k_H + |X_q \setminus Q| \cdot k_H - |X_q \setminus Q| \cdot m + 2^\ell k_H - T \cdot k_H - |X_q \setminus Q| k_H \\ &= 2^\ell \cdot k_H + s - |X_q \setminus Q| \cdot m \\ &\leq 2^\ell \cdot k_H + s - (n - T) \cdot m \\ &= 2^\ell \cdot k_H - (n \cdot m - (T \cdot m + s)) \end{aligned}$$

(The analysis for the encoding length appears in the algorithm description.) The decoding is perfect (this can be verified by inspection). Thus, by Claim 6.1, this implies that $p < 2^{-n(m-(s+T\cdot m))}$.

Protocol 4 Partial Hash Compression algorithm

function COMPRESS(H) // Treat $H \in \{0,1\}^{2^\ell \cdot k_H}$ as the truth table of a function: $H : \{0,1\}^\ell \mapsto \{0,1\}^k_H$
  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 $Q = (q_1, \ldots, q_T)$ be the set of oracle queries made by $\mathcal{A}_2$, in the order the queries were made (we can assume w.l.o.g that $|Q| = T$, and that every query is unique).
  Let $X_q = (ch_1, \ldots, ch_n)$ be the output indices of $\mathcal{A}_2$, sorted lexicographically, and $X_\pi = (x_1, \ldots, x_n)$ be the corresponding proofs.
  if $\forall i$, the m LSBs of $H(ch_i)$ equal x_i **then** // the output of $\mathcal{A}_2$ verifies; this happens w.p. p
    return $(\sigma, H(q_1), \ldots, H(q_T), H|_{X_q \setminus Q}, H|_{\neg(X_q \cup Q)})$
    - We represent $H(q_i)$ by storing the full k_H-bit entries (this uses $T \cdot k_H$ bits).
    - We represent $H|_{X_q \setminus Q}$ by storing the $k - m$ MSBs of each entry (the LSBs will be reconstructed by $\mathcal{A}_2$). The entries are stored consecutively without padding. This uses $|X_q \setminus Q| \cdot (k_H - m)$ bits.
    - We will represent $H|_{\neg(X \cup Q)}$ by storing the full k_H-bit entries. The entries are stored consecutively without padding. This uses $(2^\ell - T - |X_q \setminus Q|)k_H$ bits.
  else
    return $\perp$.
  end if
end function
function DECOMPRESS(Z)
  Parse Z as $(\sigma, H(q_1), \ldots, H(q_T), H|_{X_q \setminus Q}, H|_{\neg(X_q \cup Q)})$
  Run $\mathcal{A}_2$ with σ as input.
  When $\mathcal{A}_2$ makes the i^{th} query to H, record q_i and simulate the response $H(q_i)$ using the value from Z.
  Let X_q, X_π be the output indices and proofs given by $\mathcal{A}_2$.
  // Since this execution of $\mathcal{A}_2$ was given identical input to its execution in COMPRESS, its output will be identical.
  We can fully reconstruct H by completing $H|_{X_q \setminus Q}$ using the corresponding X_π values, and inserting the values of $H|_{X_q \cup Q}$ into $H|_{\neg(X_q \cup Q)}$ in the correct places (the indices of Q and X_q).
end function

□

7 Market-Based Mechanisms for Difficulty Adjustment

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, by choosing an appropriate incentive scheme, it turns out that we *can* dynamically set the difficulty. The main idea is to give a prover two *identifiable* options for generating proofs: the standard, storage-based PoST, and an alternative that is computation-based. By giving a small "bonus" reward for solutions that use the computation-based proofs, we incentivize users to 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.

The challenge in instantiating such a scheme is that we must ensure that (1) the difficulty of the alternative proof is equivalent to the difficulty of recomputing the PoST proof and (2) that the work expended in the alternative proof is tied to a specific instance of the PoST proof phase (i.e., that it can't be amortized across multiple instances).

To solve both of these problems, we use the PoST initialization itself as the basis for the alternative proof. However, instead of allowing an arbitrary *id* string, we require the id for the proof to be a function of the original id and the challenge from the PoST proof phase.

7.1 PoSTs With Computation Bonus

More formally, we define a PoST with Computational Bonus to be a PoST scheme with an additional "computational" prover P_{bonus} and corresponding verifier V_{bonus}.

Definition 7.1 (PoST with Computational Bonus)**.** $P = (P_{\text{init}}, P_{\text{exec}}, P_{\text{bonus}})$, $V = (V_{\text{init}}, V_{\text{exec}}, V_{\text{bonus}})$ is a (w, m, ε, f)-PoST with a computational bonus if $P' = (P_{\text{init}}, P_{\text{exec}})$ and $V' = (V_{\text{init}}, V_{\text{exec}})$ comprise an (w, m, ε, f)-PoST and the prover P_{bonus} and verifier V_{bonus} comprise a w'-PoW such that $w' \leq w$

7.1.1 Computational Solvers Will Self-Identify

To receive the computational bonus, we will require the prover to send the proof for P_{bonus}. The expected cost to compute this proof is w', while the expected cost to recompute the PoST initialization is $w \geq w'$. Thus, the strategy of using P_{bonus} dominates the strategy of recomputing the PoST, meaning that rational computational solvers will self-identify.

7.1.2 Rational Storage is Still Preferred

The adversary's expected cost for using the computational proof is w'. Thus, the expected number of successful proofs for a given budget C using the bonus proof method is C/w'. Denote β the bonus multiplier (i.e., a successful "standard" proof gets reward 1, while a computational bonus proof gets reward β).

Lemma 7.2. *If a PoST is (γ, ε')-rationally stored and $\beta < \frac{\eta \cdot w'}{\gamma \cdot m}$ then the PoST with a β computational bonus is (γ, ε')-rationally stored.*

Proof. Suppose the adversary uses $\alpha \cdot C$ of its budget for the standard PoST proofs (using an optimal adversarial strategy) and $(1 - \alpha) \cdot C$ for computational bonus proofs.

For every choice of α, if the adversary allocates less than $\varepsilon' \cdot \alpha \cdot C$ of the budget to storage, then the expected reward for the adversary is bounded by

$$\mathbb{E}\left[\#G\right] \leq \beta \cdot (1 - \alpha) \cdot C/w' + \max_{s \in [0, \varepsilon' \cdot \alpha \cdot C/\gamma)} \left\{ \sum_{i=1}^{\infty} f(i, s, \alpha \cdot C - \gamma s) \right\}$$

using the (γ, ε')-rational storage property:

$$< \beta \cdot (1 - \alpha) \cdot C/w' + \eta \cdot \alpha \cdot C/(\gamma \cdot m)$$

By our assumption about β

$$\begin{aligned} &: \leq \frac{\eta \cdot w'}{\gamma \cdot m} \cdot (1-\alpha) \cdot C/w' + \eta \cdot \alpha \cdot C/(\gamma \cdot m) \\ &= \eta \cdot (1-\alpha) \cdot C/(\gamma \cdot m) + \eta \cdot \alpha \cdot C/(\gamma \cdot m) = \eta \cdot C/(\gamma \cdot m) \,. \end{aligned}$$

In particular, this holds for $\alpha = 1$, which gives the desired result. □

7.2 Constructing PoSTs with Computational Bonus (Sketch)

Given any (w, m, ε, f)-PoST with a non-interactive initialization phase (e.g., as can be construction from a Sigma-PoST), we can extend it to a PoST with computational bonus by defining the following computational prover and verifier:

Let *id* be the id used in the PoST initialization phase. The computational prover/verifier are defined to be

$$P_{\text{bonus}}(id, ch) := P_{\text{init}}(id \| ch) \text{ and } V_{\text{bonus}}(id, ch) := V_{\text{init}}(id \| ch)$$

The security of this construction (with $w = w'$)) follows immediately from the rational storage condition of PoST soundness: this implies that PoST initialization is a proof of work. Moreover, since we use the same parameters as the underlying PoST (just with a different id), the cost is identical to initializing the PoST.

7.3 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 Simple-PoSTs is that we can incrementally increase the difficulty. For the Hash-Preimage IPoW, 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 $1/p'$. For the Partial-Hash IPoW, increasing the difficulty means reducing the number of bits stored per IPoW; this requires the prover to delete some data, and generate additional IPoWs (increasing the number of table entries) in order to maintain the same amount of space.

8 Using PoSTs in Spacemint

Spacemint is a crypto-currency based on PoSs rather than PoWs [18]. 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 [18].)

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.

8.1 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.

9 Discussion and Open Questions

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 both fast proving time and finely-tunable difficulty—by having each pebble be an IPoW (whose challenge is given by the hash of its predecessor pebbles).[9] Proving the security of this construction appears to be non-trivial, however.

Best-of-Both-Worlds? All the existing PoS constructions that don't require the prover to read its entire data don't support incremental difficulty adjustment. An interesting open question is whether it is possible to get a "best of both worlds" construction, combining low prover complexity with incremental difficulty adjustment.

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

References

[1] The chia network. URL: `https://chia.net/`.

[2] Spacemesh. URL: `https://spacemesh.io/`.

[3] H. Abusalah, J. Alwen, B. Cohen, D. Khilko, K. Pietrzak, and L. Reyzin. Beyond hellman's time-memory trade-offs with applications to proofs of space. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, volume 10625 of *Lecture Notes in Computer Science*, pages 357–379. Springer, 2017. `doi:10.1007/978-3-319-70697-9_13`.

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

[4] 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. URL: http://doi.acm.org/10.1145/2746539.2746622, doi:10.1145/2746539.2746622.

[5] 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. URL: http://dx.doi.org/10.1007/978-3-319-10879-7_31, doi:10.1007/978-3-319-10879-7_31.

[6] 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.

[7] 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.

[8] B. Cohen and K. Pietrzak. Simple proofs of sequential work. In J. B. Nielsen and V. Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 451–467. Springer, 2018. URL: https://doi.org/10.1007/978-3-319-78375-8_15, doi:10.1007/978-3-319-78375-8_15.

[9] 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.

[10] 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. URL: http://dx.doi.org/10.1007/978-3-662-48000-7_29, doi:10.1007/978-3-662-48000-7_29.

[11] B. Fisch. Poreps: Proofs of space on useful data. *IACR Cryptology ePrint Archive*, 2018:678, 2018. URL: https://eprint.iacr.org/2018/678.

[12] B. Fisch. Tight proofs of space and replication. Cryptology ePrint Archive, Report 2018/702, 2018. URL: https://eprint.iacr.org/2018/702.

[13] 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.

[14] 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.

[15] P. Labs. Filecoin: A decentralized storage network, 2017. URL: https://filecoin.io/filecoin.pdf.

[16] 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. URL: http://dx.doi.org/10.1109/SP.2014.37, doi:10.1109/SP.2014.37.

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. webpage, 2008. https://bitcoin.org/bitcoin.pdf.

[18] S. Park, A. Kwon, G. Fuchbauer, P. Gazi, J. Alwen, and K. Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC). Springer*, 2018. URL: http://fc18.ifca.ai/preproceedings/78.pdf.

[19] C. Percival. Stronger key derivation via sequential memory-hard functions. BSDCan 2009, 2009.

[20] 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.

[21] K. Pietrzak. Proofs of catalytic space. In A. Blum, editor, *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, volume 124 of *LIPIcs*, pages 59:1–59:25. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ITCS.2019.59.

[22] L. Ren and S. Devadas. Proof of space from stacked expanders. In M. Hirt and A. D. Smith, editors, *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part I*, volume 9985 of *Lecture Notes in Computer Science*, pages 262–285, 2016. URL: https://doi.org/10.1007/978-3-662-53641-4_11, doi:10.1007/978-3-662-53641-4_11.

[23] 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.