# Reversible Proofs of Sequential Work

Hamza Abusalah[1(✉)], Chethan Kamath[2], Karen Klein[2], Krzysztof Pietrzak[2], and Michael Walter[2]

[1] SBA Research, Vienna, Austria
habusalah@sba-research.org
[2] IST Austria, Am Campus 1, 3400 Klosterneuburg, Austria
{ckamath,kklein,pietrzak,mwalter}@ist.ac.at

**Abstract.** Proofs of sequential work (PoSW) are proof systems where a prover, upon receiving a statement $\chi$ and a time parameter $T$ computes a proof $\phi(\chi, T)$ which is efficiently and publicly verifiable. The proof can be computed in $T$ sequential steps, but not much less, even by a malicious party having large parallelism. A PoSW thus serves as a proof that $T$ units of time have passed since $\chi$ was received.

PoSW were introduced by Mahmoody, Moran and Vadhan [MMV11], a simple and practical construction was only recently proposed by Cohen and Pietrzak [CP18].

In this work we construct a new simple PoSW in the random permutation model which is almost as simple and efficient as [CP18] but conceptually very different. Whereas the structure underlying [CP18] is a hash tree, our construction is based on skip lists and has the interesting property that computing the PoSW is a reversible computation.

The fact that the construction is reversible can potentially be used for new applications like constructing *proofs of replication*. We also show how to "embed" the sloth function of Lenstra and Weselowski [LW17] into our PoSW to get a PoSW where one additionally can verify correctness of the output much more efficiently than recomputing it (though recent constructions of "verifiable delay functions" subsume most of the applications this construction was aiming at).

## 1 Introduction

*Timed-release cryptography* was envisioned by May [May93] and realised by Rivest, Shamir and Wagner [RSW00] in the form of a "time-lock puzzle". For a time parameter $T$, such a puzzle can be efficiently sampled together with a solution. However, solving it requires $T$ sequential computational steps, and this holds even for parties aided with massive parallelism. In other words, there are no "shortcuts" to the solution. The application envisioned in [RSW00] was "sending a message to the future": generate a puzzle, derive a symmetric key from the

H. Abusalah—Partially supported by FFG COMET under SBA-K1 grant.
C. Kamath, K. Klein, K. Pietrzak and M. Walter—Supported by the European Research Council, ERC consolidator grant (682815 - TOCNeT).

Y. Ishai and V. Rijmen (Eds.): EUROCRYPT 2019, LNCS 11477, pp. 277–291, 2019.
https://doi.org/10.1007/978-3-030-17656-3_10

solution, encrypt your message using that key, then release the ciphertext and the puzzle. Now everyone can decrypt by solving the puzzle which requires $T$ sequential steps.

The construction put forward in [RSW00] is in the RSA setting: the puzzle is a tuple $(N, x, T)$, where $N = p \cdot q$ is an RSA modulus and $x \in Z_N^*$ a group element, and the solution to the puzzle is $x^{2^T} \bmod N$. Although the solution can be computed efficiently *if* the factorisation of $N$ is known, it is conjectured to require $T$ sequential squarings given only $N$.

The assumption that underlies the soundness of the [RSW00] time-lock puzzle is rather non-standard (which is basically that the puzzle is sound, i.e., there's no shortcut in computing the solution) and it's an open problem to come up with constructions under more standard assumptions. In a negative result, Mahmoody, Moran and Vadhan [MMV11] show that there's no black-box construction of a time-lock puzzle in the random oracle model. In subsequent work the same authors [MMV13] propose and construct proofs of sequential work (PoSW). This is a proof system wherein a prover $\mathcal{P}$ can convince a verifier $\mathcal{V}$ that it spent $T$ sequential time steps upon receiving some challenge $\chi$. Even though PoSW seem related to time-lock puzzles, they are not directly comparable. In particular, a PoSW does not require that one can sample the solution together with an instance. On the other hand, a PoSW must be publicly verifiable[1] and sampling a challenge must be public-coin[2] so it can be made non-interactive by the Fiat-Shamir heuristic. [MMV13] construct a PoSW in the random oracle model (or under a standard model assumption on hash functions called "sequentiality").

As possible applications for PoSW [MMV13] suggest universally verifiable CPU benchmarks and non-interactive time-stamping. The construction given in [MMV13] is not practical as a prover needs not only $T$ sequential time steps but also linear in $T$ space to compute a proof. Cohen and Pietrzak [CP18] resolved this issue by constructing a PoSW where the prover requires just $\log(T)$ space.

More recently there has been renewed interest in time-delayed cryptography as it found applications in decentralized systems, including public randomness beacons (cf. discussion in [BBBF18]), blockchain designs like chia.net or proofs of replication [Fis19]. For the first two applications the PoSW need to be *unique*, which means it should not be possible (or at least computationally hard) to compute more than one accepting proof for the same challenge. This notion was introduced in [CP18] but constructing such a PoSW was left as an open problem.

**Our Contribution.** Constructions of hash-based PoSW start with some underlying graph structure, which in [MMV13] is a depth-robust graph and in [CP18] a binary tree with some extra edges. In this paper we construct a new PoSW which

[1] So everybody, not just the party who generated the challenge, can efficiently verify correctness. Note that in the RSW time-lock puzzle only the party who generated the challenge (which is called a puzzle in this context) and thus knows the factorization can verify the proof efficiently.

[2] This basically means that the challenge is just a uniformly random string. Note that the RSW time-lock puzzle is not public-coin as the coins used to sample the RSA modulus $N$ must remain secret.

is as simple and almost as efficient as [CP18] with the underlying graph being a skip list. Our construction can be instantiated with permutations – instead of hash functions – and is "reversible".

Until recently the sloth hash function [LW17] was the closest we had to a *unique* PoSW. It's not a PoSW because the computation required for verification is linear in the time parameter $T$ (albeit around a 1000 times faster). The fact that our PoSW is reversible allows us to "embed" sloth into our PoSW, this way we get a PoSW where verification (of the claim that $T$ sequential time steps were spent) is very efficient (logarithmic in $T$), while verifying correctness can be done as efficiently as in sloth (in time $O(T)$ with a very small hidden constant). We outline this construction in more detail in Sect. 1.1 below.

For applications of unique PoSW our construction is by now mostly subsumed by very recent constructions of verifiable delay functions [BBBF18] (VDF). A VDF is defined almost like a PoSW, but the (non-interactive) proof does not only certify that $T$ sequential time has been used to compute some value, but the stronger property that this value is actually the correct value. A VDF is thus basically a unique PoSW (the only reason it's not exactly a unique PoSW is that the proof itself could be malleable, but this doesn't matter for any of the applications). The notion of a VDF has been introduced by Boneh et al. in [BBBF18] who also construct a VDF using rather heavy machinery like incrementally verifiable computation. Subsequently two extremely simple and efficient VDFs have been proposed [Wes19, Pie19b], both papers basically show how to make the RSW time-lock puzzle [RSW00] publicly verifiable, that is, they give proof systems for showing that a given tuple $(x, y, T)$ satisfies $x^{2^T} = y$ in a group of unknown order (e.g. $Z_N^*$ as used in [RSW00]). These constructions are clearly favourable to ours as correctness (which here means that the output is correctly computed) can be verified much more efficiently, though as they are not post-quantum secure, ours is arguably still the best option in a post-quantum setting for some applications. This hopefully will change in the near future as research on post-quantum VDFs is ongoing [FMPS19].

The fact that our PoSW is reversible seems also useful in the context of proofs of replication [Fis19, Fis18, Pie19a] for similar reasons that "decodable" VDFs are useful in this context as discussed in [BBBF18], we are currently working towards constructing simple proofs of replication based on the skip list based PoSW presented in this paper.

## 1.1 Hash Chains and the Sloth Function

A simple construction which is not quite a PoSW is a hash chain, where on input $x = x_0$ one outputs as proof $x_T$ which is recursively computed as $x_i = hash(x_{i-1})$. If *hash* is a bijection and can be efficiently evaluated in both directions (i.e., a permutation), then from a given state $x_i$, one can compute the previous state $x_{i-1}$, we call such a construction *reversible*.

Verifying that $x_T$ has been correctly computed requires $T$ hashes (so it's no a PoSW), but at least one can parallelize verification by additionally outputting some $q$ intermediate values $x_0, x_{T/q}, x_{2T/q}, \ldots, x_T$ (then the proof can be verified in $T/q$ time assuming one can evaluate $q$ instantiations of *hash* in parallel: for every $i \in [q]$, verify that $T/q$ times hashing $x_{(i-1)T/q}$ gives $x_{iT/q}$).

Lenstra and Wesolowski [LW17] suggest a construction called "sloth", which basically is a hash chain but with the additional property that it can be verified with a few hundred times less computation than what is required to compute it. The construction is based on the assumption that computing square roots in a field $\mathbb{F}p$ of size $p$ is around $\log(p)$ times slower than the inverse operation, which is just squaring. A typical value would be $\log(p) \approx 1000$, going much higher is problematic as then fast multiplication methods (e.g., Karatsuba, Schönhage-Strassen) can be applied.

Their idea is to simply use a hash chain where the hash function is some permutation $\pi : \mathbb{F}p \to \mathbb{F}p$, where $\mathbb{F}p$ is a finite field of size $p$, followed by taking a square root: that is $x_i = \sqrt{\pi(x_{i-1})}$. Verification goes as for a standard hash chain, but one computes backwards, checking $x_{i-1} = \pi^{-1}(x_i^2)$, which – assuming computing $\pi, \pi^{-1}$ is cheap compared to squaring, and squaring is $\log(p)$ times faster than taking square roots – gives the claimed speedup of $\approx \log(p)$ compared to a simple hash chain.

In Sect. 4 we show how sloth can be embedded into our skip list based PoSW to get a construction such that it remains a good PoSW, while correctness of the output can be verified as efficiently as in sloth, the constructions discussed are summarized in the table below.

## 2 Construction

### 2.1 Notation

Throughout we denote the time parameter of our construction by $N = 2^n$ with $n \in \mathbb{N}$ and assume it's a power of 2. We reserve $w, t \in \mathbb{N}$ to denote two statistical security parameters, $w$ is the block size (say $w = 256$) and $t$ denotes the number of challenges: a cheating prover who only makes $N(1-\epsilon)$ sequential steps (instead $N$) will pass verification with probability $(1-\epsilon)^t$. For integers $m, m'$ we denote with $[m, m'] = \{m, m+1, \ldots, m'\}$, $[m], [m]_0$ are short for $[1, m]$ and $[0, m]$.

We define $\tilde{0} = n + 1$ and for $i \geq 1$ we denote with $\tilde{i}$ the number of trailing zeros in the binary representation of $i$, plus 1

$$\tilde{0}, \tilde{1}, \tilde{2}, \tilde{3}, \tilde{4}, \tilde{5}, \tilde{6}, \tilde{7}, \tilde{8}, \tilde{9}, \ldots \;=\; n+1, 1, 2, 1, 3, 1, 2, 1, 4, 1, \ldots$$

For $\sigma \in \{0,1\}^{w \cdot i}$ we denote with $\sigma^{(j)}$ the $j$th $w$-bit block of $\sigma$, so that $\sigma = \sigma^{(1)} \| \ldots \| \sigma^{(i)}$. $\sigma^{(i \ldots j)}$ is short for $\sigma^{(i)} \| \ldots \| \sigma^{(j)}$.

For a permutation $\pi$ over $\ell$ bit strings, we denote with $\dot{\pi}$ the function over bit stings of length $\geq \ell$ which simply applies $\pi$ to the $\ell$ bit prefix of the input, and leaves the rest untouched.

| construction (using time parameter $T$ and statistical security parameter $\lambda$) | # of steps[d] to verify sequential computation $O(\cdot)$ of | # of steps[d] to verify if output is correct (uniqueness) $O(\cdot)$ of | assumption | step | post quantum | reversible |
|---|---|---|---|---|---|---|
| hash chain | $T$ | $T$ | random oracle[a] | RO call | yes | yes[b] |
| sloth [LW17] | $T/\log(p)$ | $T/\log(p)$ | $\log(p)$ gap computing $\sqrt{x}$ vs $x^2$ and random permutation[a,c] | $x \to \sqrt{x}$ & RP call | yes | yes |
| PoSW [CP18] | $\lambda \cdot \log(T)$ | $T$ | random oracle[a] | RO call | yes | no |
| PoSW Sect. 2 | $\lambda \cdot \log^2(T)$[f] | $T$ | random permutation[a,c] | RP call | yes | yes |
| Combined Sect. 4 | $\lambda \cdot \log^2(T)$[f] | $T/\log(p)$ | like sloth | $x \to \sqrt{x}$ & RP call | yes | yes |
| [Pie19b] VDF | $\lambda \cdot \log(T)$ | $\lambda \cdot \log(T)$ | $(x,T) \to x^{2^T}$ requires $T$ sequential squarings[e] | $x \to x^2$ | no | no |
| [Wes19] VDF | $\lambda$ | $\lambda$ | as above plus "root assumption" | $x \to x^2$ | no | no |

[a]Or a standard model assumption called "sequential hash function".
[b]If the function used is an efficiently invertible permutation.
[c]The random permutation model is equivalent to the random oracle model.
[d]What a step is depends on the construction, but evaluating the function is always assumed to require $T$ sequential steps.
[e]This assumption can only hold in groups of unknown order.
[f]Strictly speaking, the number of oracle calls required to verify is just $\lambda \cdot \log T$ (as in [CP18]), but in our constructions the input consists of up to $\log T + 1$ blocks (unlike [CP18], where it is 2 blocks) and therefore to make a fairer comparison, we count the cost of an oracle call on an input of length $k$ blocks as $k$ calls.

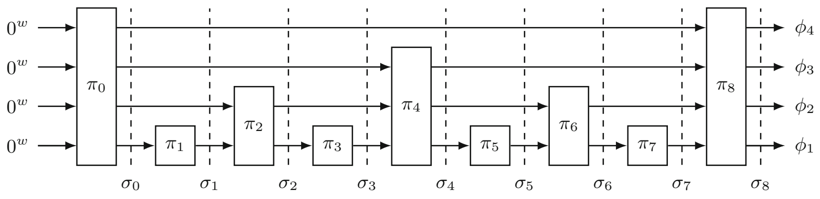

**Fig. 1.** Illustration of the computation of $\sigma_\Pi = (\sigma_0, \ldots, \sigma_N)$ with $n = 3, N = 2^n = 8$. The blocks represent the permutations, whereas the dashed vertical lines represent the states. Note that the structure of the graph is the same as a skip list with four layers, where a pointer in layer $i$, $i \in \{0, 1, 2, 3\}$, points to the $2^i$-th element to its right on the list.

## 2.2 The Sequence $\sigma_\Pi$

At the core of our construction is a mapping based on the skip list data structure (see Fig. 1). It is built from a set of permutations $\Pi = \{\pi_i\}_{i \in [N]_0}$, where each $\pi_i$ is over $\{0,1\}^{w \cdot \tilde{i}}$, and defines a sequence of states $\sigma_\Pi = \sigma_0, \ldots, \sigma_N$, $\sigma_i \in \{0,1\}^{(n+1)\cdot w}$, recursively as

$$\sigma_0 = \dot{\pi}_0(0^{w\cdot(n+1)}) \text{ and for } i > 0 \ : \ \sigma_i = \dot{\pi}_i(\sigma_{i-1}) \left( = \pi_i(\sigma_{i-1}^{(1\ldots\tilde{i})}) \| \sigma_{i-1}^{((\tilde{i}+1)\ldots(n+1))} \right).$$

### 2.3 The DAG $G_N$

It will be convenient to consider the directed acyclic graph (DAG)

$$G_N = (V, E)\ ,\ V = [N]_0\ ,\ E = \{(i,j) \in V^2\ : \exists k \geq 0\ : j - i = 2^k, 2^k | i\}$$

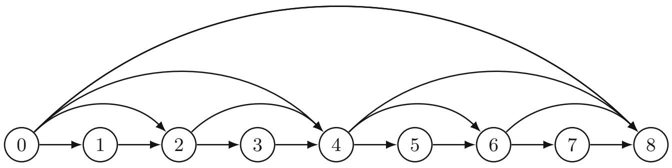

**Fig. 2.** The graph $G_8$ that corresponds to the computation of $\sigma_\Pi$ with $n = 3$.

that is derived from the computation of $\sigma_\Pi$ as follows: identify the permutation $\pi_i$ with the node $i$ and add a directed edge $(i, j)$ if in the computation of $\sigma_\Pi$ part of the output of $\pi_i$ is piped through directly to $\pi_j$ (see Fig. 2).

For $i \in [N-1]$ we denote with $\mathsf{path}(i) \subseteq V$ the subgraph of $V$ induced by the nodes on the shortest path in $G_N$ which starts at 0, ends at $N$ and passes through node $i$. For example, in Fig. 1,

$$\mathsf{path}(5) = (\{0,4,5,6,8\}, \{(0,4),(0,8),(4,5),(4,6),(4,8),(5,6),(6,8)\}).$$

It's not hard to check that the number of vertices in $\mathsf{path}(i)$ is $n + 3 - \tilde{i}$, and in particular is never more than $n + 2$.

### 2.4 Consistent States/Paths

By construction, the $\sigma_i \in \sigma_\Pi$ satisfy $\sigma_i = \dot{\pi}_{i+1}^{-1}(\sigma_{i+1})$, and more generally, for every edge $(i, j) \in E$ and $d = \min(\tilde{i}, \tilde{j})$

$$\sigma_i^{(d\ldots n+1)} = (\dot{\pi}_j^{-1}(\sigma_j))^{(d\ldots n+1)}.$$

We say two strings are consistent for $(i, j)$ if they satisfy this condition.

**Definition 1 (Consistent States/Path).** *$\alpha_i, \alpha_j \in \{0,1\}^{(n+1)\cdot w}$ are consistent for edge $(i,j) \in E$ if with $d = \min(\tilde{i}, \tilde{j})$*

$$\alpha_i^{(d\ldots n+1)} = (\dot{\pi}_j^{-1}(\alpha_j))^{(d\ldots n+1)}\ .$$

*We say $\alpha'_i \in \{0,1\}^{\tilde{i}\cdot w}, \alpha'_j \in \{0,1\}^{\tilde{j}\cdot w}$ are consistent if they can be "padded" to consistent $\alpha_i, \alpha_j$ as above, which is the case if*

$$\alpha'^{(d)}_i = \pi^{-1}(\alpha'_j)^{(d)}.$$

*We say $\{\alpha_i\}_{i \in \mathsf{path}(k)}$ are consistent with $\mathsf{path}(k)$ if $\alpha_i, \alpha_j$ are consistent for every edge $(i,j) \in \mathsf{path}(k)$.*

Note that if $\alpha_j$ is computed from $\alpha_i$ by applying $\dot{\pi}_{i+1}, \ldots, \dot{\pi}_j$ to $\alpha_i$, then those $\alpha_i, \alpha_j$ will be consistent with $(i, j)$, but the converse is not true (except if $j = i + 1$).

### 2.5 PoSW Construction

The protocol between $\mathcal{P}, \mathcal{V}$ on common input $T = N = 2^n, w, t$ goes as follows

1. $\mathcal{V}$ samples $\chi \leftarrow \{0,1\}^{w \cdot n}$ and sends it to $\mathcal{P}$. This $\chi$ defines a fresh set of random permutations $\Pi$ (cf. Remark 1 below).
2. $\mathcal{P}$ computes $\sigma_0, \ldots, \sigma_N$ and sends $\phi = \sigma_N$ to $\mathcal{V}$.
3. $\mathcal{V}$ samples $t$ challenges $\gamma = (\gamma_1, \ldots, \gamma_t) \leftarrow [N-1]^t$ and sends them to $\mathcal{P}$.
4. $\mathcal{P}$ sends $\{\sigma_i\}_{i \in \mathsf{path}(j), j \in \gamma}$ to $\mathcal{V}$ (cf. Remark 2 below).
5. $\mathcal{V}$ verifies for every $j \in \gamma$ that $\{\sigma_i\}_{i \in \mathsf{path}(j)}$ is consistent as in Definition 1. If any check fails output $\mathsf{reject}$, output $\mathsf{accept}$ otherwise.

*Remark 1 (Seeding Random Oracles/Permutations).* Ideal permutations can be constructed from random oracles [CPS08,HKT11,DSKT16] (formally, the ideal permutation model is indifferentiable from the random oracle model), so we can realize $\Pi$ in the standard random oracle model.[3] Consider a fixed random oracle $\mathcal{H}(\cdot)$ about which a potential adversary has some auxiliary input (i.e., it has queried it on many inputs before, and stored some information $\mathsf{aux}$). If one samples a random seed $\chi$ and uses it as a prefix to define the function $\mathcal{H}_\chi(x) = \mathcal{H}(\chi \| x)$, this $\mathcal{H}_\chi$ – from the adversaries' perspective – is a fresh random oracle as long as this seed is just a bit longer that $\log(|\mathsf{aux}|)$ [DGK17]. Thus, we can also sample a fresh $\Pi$ by just sending a seed $\chi$.

*Remark 2 ($\mathcal{P}'s$ Space Requirement).* To avoid any extra computation in step 4., $\mathcal{P}$ would need to store the entire $\sigma_\Pi = \{\sigma_i\}_{i \in [N]_0}$. By using a bit of extra computation, one can reduce the space requirement (we remark that a similar trade off comes up in [CP18]). Concretely, for some $K = 2^k$, we let $\mathcal{P}$ only store $\sigma_i$ where $2^k | i$, thus storing only $N/K$ states. From this, every state $\sigma_i$ can be computed making at most $K/2$ invocations to $\Pi$ (and $K/2$ not $K$ as we can also compute backwards).

## 3 Security Proof

**Theorem 1.** *Consider a malicious prover $\tilde{\mathcal{P}}$ which*

1. *makes at most $N - \Delta$* sequential *queries to permutations in $\Pi$ before sending $\phi = \sigma_N$ (in step 3 of the protocol); and*
2. *queries the permutations in $\Pi$ on at most $q$ inputs in total during execution of the protocol.*

*Then $\tilde{\mathcal{P}}$ will win (i.e., make $\mathcal{V}$ output accept) with probability at most*

$$\mathsf{Pr}\left(\tilde{\mathcal{P}} \text{ wins}\right) \leq \frac{2q^2(n+3)^2}{2^w} + \left(\frac{N - \Delta}{N}\right)^t. \tag{1}$$

[3] In practice, one could e.g. use $\chi$ to sample $N+1$ AES keys $k_0, \ldots, k_N$, and then use $AES(k_i, \cdot) : \{0,1\}^{256} \rightarrow \{0,1\}^{256}$ – i.e., AES with a fixed public key – to construct $\pi_i$, where for $\tilde{i} > 1$ one would use domain extension for random permutations to extend the domain to $256 \cdot \tilde{i}$ bits.

The proof of Theorem 1 mainly follows the intuition that sending $\phi$ "commits" the prover to a set of challenges it can respond to. We prove this fact formally in Lemma 7. If this set is a large fraction of the possible challenges, it implies the existence of a long sequence (as defined below) that necessarily requires many sequential steps.

To aid the proof we begin with a couple of definitions. The first one is merely for notational convenience.

**Definition 2.** *We use $\sim$ to denote that two strings (composed of $w$-bit blocks) contain an identical block*

$$\alpha \sim \alpha' \iff \exists i, j : \alpha^{(i)} = \alpha'^{(j)}.$$

*We then say that $\alpha$ and $\alpha'$* collide*.*

The next definition characterizes the property that paths through our skiplist construction satisfy and that we rely on for the proof.

**Definition 3 ($\Pi$-Sequence).** *For a family of $N$ permutations $\Pi = \{\pi_i\}_i$ a $\Pi$-*sequence *of length $N' < N$ is an $N'$-tuple of pairs of strings $((x_j, y_j))_j$ together with an $N'$-tuple of strictly increasing integers $(i_j)_j$ such that for all $j$*

$$\pi_{i_j}(x_j) = y_j \text{ and } y_j \sim x_{j+1}.$$

Below we show that $\Pi$-sequences are inherently sequential (cf. Lemma 6 and Corollary 1), but that requires a few technical lemmas, so we defer the details and proceed to the main proof. We now show how Lemma 7 and Corollary 1 imply Theorem 1.

*Proof (of Theorem 1).* Consider a malicious prover $\tilde{\mathcal{P}}$ that convinces the verifier on a random challenge with probability $\geq \frac{N-\Delta}{N}$. Since the correct response to any challenge is a distinct $\Pi$-sequence from 0 to $\phi$, Lemma 7 implies that $\tilde{P}$ must be able to respond to a fraction $\geq \frac{N-\Delta}{N}$ of the challenges. This is because the set of $\Pi$-sequences from 0 to $\phi$ that $\tilde{\mathcal{P}}$ can compute is essentially fixed after sending $\phi$ and thus independent of the choice of challenges. This means there must exist a set of $N - \Delta$ responses which can be pieced together to a $\Pi$-sequence of length $N - \Delta$ from 0 to $\phi$ (details below). Note that all responses can be obtained by sending all the challenges using rewinding (which does not increase the number of queries). Then the result follows from Corollary 1.

It remains to establish the fact that the responses can be merged to a long sequence. To see this, first assume that whenever two paths contain the same node, the corresponding responses have the same state at this node. If this is the case then merging the responses to $k$ distinct challenges is easy: simply take the "union" of the responses, which will be a $\Pi$-sequence of length at least $k$.

Finally, we show that different verifying reponses must have the same state at intersecting nodes. The proof of this fact is recursive: consider the node $N/2$ and assume for contradiction that there are two paths that both verify and each contains a state $\sigma_{N/2}$ and $\sigma'_{N/2}$, respectively, with $\sigma_{N/2} \neq \sigma'_{N/2}$. First note that the states $\sigma_0$ and $\sigma'_0$ must both be equal to $\pi_0(0)$, so they are equal to each other. Similarly, $\sigma_N$ and $\sigma'_N$ must both be equal to $\phi$ in order to both verify. Furthermore, verification ensures that $\sigma_{N/2} \sim \sigma'_{N/2}$. Specifically, they are equal in block $n-1$, where they must both be equal to $\pi_N^{-1}(\sigma_N)^{(n-1)}$, since verification checks the edge $(N/2, N)$ for consistency (cf. Definition 1). Analogously, verification ensures that $\pi_{N/2}^{-1}(\sigma_{N/2}) \sim \pi_{N/2}^{-1}(\sigma'_{N/2})$, since this corresponds to the edge $(0, N/2)$. Note that the latter pair of values could be extracted from the prover by sending the appropriate challenges. By Lemma 5 (proved below) this can only happen with probability $\leq \frac{2q^2(n+3)^2}{2^w}$. We conclude that $\sigma_{N/2}$ is equal among all valid responses with overwhelming probability. This allows to recurse on the node $N/4$ and $3N/4$, etc. $\square$

We now establish the remaining lemmas used in the main proof. Throughout the rest of this section, w.l.o.g. we only consider algorithms that do not make redundant queries. In all results in this section pertaining to random permutations the probabilities are taken over the choice of the permutations.

First, we need a version of a PRP/PRF switching lemma that allows the adversary oracle access to the permutation and its inverse. We have not seen such a version in the literature so we prove it in the appendix.

**Lemma 1.** *Let $\pi : \{0,1\}^w \mapsto \{0,1\}^w$ be a random permutation and consider an algorithm $\mathsf{A}^{\pi,\pi^{-1}}$ with oracle access to $\pi$ and $\pi^{-1}$ that makes exactly $q$ queries in total. Assume that $\mathsf{A}$ does not repeat any queries to $\pi$ nor any queries to $\pi^{-1}$, and that if it queries $\pi$ at $x$, it does not query $\pi^{-1}$ at $\pi(x)$ and vice versa. Let $F_1, F_2 : \{0,1\}^w \mapsto \{0,1\}^w$ be independent random functions. Then for any event $E$ over the output of $\mathsf{A}$, we have $\mathsf{Pr}\left(\mathsf{A}^{\pi,\pi^{-1}} \in E\right) \leq \mathsf{Pr}\left(\mathsf{A}^{F_1,F_2} \in E\right) + \frac{q(q-1)}{2^w}$, where the first probability is over the choice of $\pi$ and the second over the choice of $F_1, F_2$.*

The Lemma shows that in the analysis we can replace the random permutation and its inverse oracle with random functions. Note that by a simple hybrid argument, Lemma 1 also holds for families of permutations, where $q$ is the sum over all queries and $w$ is the minimal input/output length over all permutations.

We now show that Lemma 1 implies a few restrictions on what an algorithm can achieve when querying random permutations. Namely, we first show that input/output pairs are hard to guess (cf., Lemma 2), that preimages are hard to find without using the inverse oracle (cf., Lemma 3), and that it is hard to find queries that result in collisions with earlier queries (cf., Lemma 4).

**Lemma 2.** *Let $\Pi = \{\pi_i\}_i$ be a family of random permutations. For any oracle algorithm outputting a pair $(x, y)$ and an integer $i$ and making $q$ queries to $\Pi$* except *$x$ in forward or $y$ in backward direction, the probability that $\pi_i(x) = y$ is $\leq \frac{q^2}{2^w}$.*

*Proof.* We are trying to bound the probability that the algorithm is able to guess the input/output pair of one of the permutations in $\Pi$ (after making at most $q$

queries). If the $\pi_i$ were random functions, this probability would be $\leq \frac{1}{2^w}$. By Lemma 1 the bound follows. $\square$

**Lemma 3.** *Let $\Pi = \{\pi_i\}_i$ be a family of random permutations. For any algorithm taking $y$ as input and making $q$ queries to $\Pi$* except querying $\pi_i^{-1}$ for $y$ *and outputting some $x$ and $i$, the probability that $\pi_i(x) = y$ is $\leq \frac{q^2}{2^w}$.*

*Proof.* If $\pi_i$ and $\pi_i^{-1}$ were random function, the probability of finding such an $x$ would be $\frac{1}{2^w}$. Lemma 1 completes the proof. $\square$

**Lemma 4.** *Let $\Pi = \{\pi_i\}_i$ be a family of random permutations. For any algorithm making $q$ queries to $\Pi$ the probability of a query to $\Pi$ either in forward or backward direction resulting in a response $z$ that collides (in the sense of $\sim$) with any of the previous queries (in either input or output) is $\leq \frac{2q^2(n+2)^2}{2^w}$.*

*Proof.* Assume we replace the permutations with random functions. The probability that the response to any query collides with a specific string is at most $(n+1)^2/2^w$, since there are at most $n+1$ blocks in each string. By union bound, the probability that a query collides with any of the previous queries is thus at most $2q(n+1)^2/2^w$, since there are two strings in each query (input and output). Applying a final union bound to all queries shows that the probability of this event is $2q^2(n+1)^2/2^w$. Lemma 1 now proves the result. $\square$

Using the basic lemmas above, we can make statements about certain cyclic structures that are hard to find in random permutations and about the sequentiality of random permutations.

**Lemma 5.** *Let $\Pi = \{\pi_i\}_i$ be a family of random permutations. For any algorithm making $q$ queries to $\Pi$ and outputting two distinct values, $x$ and $x'$, and an integer $i$, the probability that $x \sim x'$ and $\pi_i(x) \sim \pi_i(x')$ is $\leq \frac{2q^2(n+3)^2}{2^w}$.*

*Proof.* Obtaining two such pairs requires to guess one of the two input/output pairs or find a colliding query. Union bound over the two events (which are bounded by Lemmas 2 and 4, respectively) yields the bound. $\square$

**Lemma 6 ($\Pi$-Sequentiality of Random Permutations).** *Let $\Pi = \{\pi_i\}_i$ be a family of random permutations. For any algorithm $A$ taking as input $x$ and making a sequence $Q$ of $q$ queries to $\Pi$, and any $\Pi$-sequence $s$ starting at $x$, the probability of $A$ outputting $s$ and $Q$ not containing the pairs in $s$* in order and in forward direction *is $\leq \frac{2q^2(n+3)^2}{2^w}$.*

*Proof.* Producing a $\Pi$-sequence starting at a specific value without querying the pairs in order and in forward direction, requires to either guess some input/output pair (for some specific $\pi_i$) or find a colliding query, similarly to Lemma 5. $\square$

**Corollary 1.** *Let $\Pi = \{\pi_i\}_i$ be a family of random permutations. For any algorithm taking as input $x$ and making $q$ sequential queries to $\Pi$, the probability of outputting a $\Pi$-sequence of length longer than $q$ is $\leq \frac{2q^2(n+3)^2}{2^w}$.* $\square$

We use the above observations to show that nothing the prover does after sending its commitment $\phi$ will help responding to challenges.

**Lemma 7.** *Let $(\mathcal{P}_1^{\Pi}, \mathcal{P}_2^{\Pi})$ be a pair of algorithms such that*

- *$\mathcal{P}_1$, on input $x$, makes $q_1$ queries to $\Pi$, and outputs a state $s_1$ and some $y$*
- *$\mathcal{P}_2$, on input $s_1, x, y$, makes $q_2$ queries to $\Pi$ and outputs a $\Pi$-sequence $s$.*

*Let $Q$ be the set of queries (including responses) made by $\mathcal{P}_1$, and let $S$ be the set of $\Pi$-Sequences between $x$ and $y$ computable[4] from $Q$ without any further queries to $\Pi$. Then $s \in S$ except with probability $\leq \frac{2q^2(n+3)^2}{2^w}$, where $q = q_1 + q_2$.*

*Proof.* Assume $s \notin S$. Let $(x', y)$ be the last pair in $s$. First consider the case that the query $(x', y) \in Q$. Since $s$ is a new sequence not computable from $S$ it must contain a pair $(x_i, y_i) \notin Q$, so the queries in $s$ were not made in order and thus by Lemma 6 the probability of $\mathcal{P}_2$ outputting $s$ is $\leq \frac{2q^2(n+3)^2}{2^w}$.

Now consider the case $(x', y) \notin Q$. If $\mathcal{P}_2$ did not query $x'$ in forward direction, by Lemma 6 the probability of $\mathcal{P}_2$ outputting $s$ is $\leq \frac{2q^2(n+3)^2}{2^w}$. Finally, if $\mathcal{P}_2$ queried $x'$ in forward direction, it did not submit an equivalent query in the reverse direction by assumption. (Recall that we consider only algorithms that do not make redundant queries.) It follow from Lemma 3 that the probability of this event is $\leq \frac{q^2}{2^w}$. □

This completes the proof.

## 4 Embedding Sloth

As discussed in the introduction, we propose a reversible PoSW that is almost as efficient as the construction from [CP18] but achieves a larger time gap between the computation of the proof and the verification of correctness. To this aim, we embed the sloth hash function from [LW17] into construction 2.5.

The idea underlying sloth is to use the fact that the best known algorithms for computing modular square roots in a field $\mathbb{F}p$ takes $\approx \log(p)$ sequential squarings, whereas verification of the result only takes a single modular squaring. Thus, this gives a good candidate to build the *slow-timed hash* function *sloth*.

Let $p \equiv 3 \mod 4$ be a prime. We identify $x \in \mathbb{F}p^{\times}$ with its canonical representant in $[0, p-1]$. If $x \in \mathbb{F}p^{\times}$ is a quadratic residue, then there are two square roots $y, y' \in \mathbb{F}p^{\times}$, where $y' = p - y$, one of them being even, the other one odd. Let $\sqrt[+]{x}$, $\sqrt[-]{x}$ denote the (unique) even and odd square root of $x$, respectively. If $x \in \mathbb{F}p^{\times}$ is not a quadratic residue, then $-x$ is a quadratic residue, so it makes sense to define a permutation $\rho : \mathbb{F}p^{\times} \to \mathbb{F}p^{\times}$ as

$$\rho(x) = \begin{cases} \sqrt[+]{x}, & \text{if } x \text{ is a quadratic residue,} \\ \sqrt[-]{-x}, & \text{otherwise.} \end{cases}$$

[4] By "computable" we mean here that there exists an algorithm for which the output is correct with non-negligible probability.

Its inverse is defined by

$$\rho^{-1}(x) = \begin{cases} x^2, & \text{if } x \text{ is even,} \\ -x^2, & \text{otherwise.} \end{cases}$$

Unfortunately, one cannot directly build a hash chain by iterating $\rho$ since reducing modulo $p-1$ in the exponent would yield a much faster computation than sequentially computing $\rho$. Lenstra and Wesolowski [LW17] solve this problem by prepending an easily computable (in both directions) permutation $\pi$ on $\mathbb{F}p^\times$ to each iteration of the square rooting function $\rho$. Setting $\tau = \rho \circ \pi$, the sloth function is hence defined as $\tau^N$ for some appropriate chain length $N$. Verification can be done backwards by the computation $(\tau^N)^{-1} = (\sigma^{-1} \circ \rho^{-1})^N$, which is by a factor $\log p$ faster.

We now combine the ideas from [LW17] with our construction to achieve an efficient PoSW while preserving the fast verification of correctness obtained by the sloth construction. Let $\Pi = \{\pi_i\}_{i \in [N]_0}$ be a set of permutations where, for each $i \in [N]_0$, $\pi_i : \mathbb{F}p^\times \times \{0,1\}^{w \cdot (\tilde{i}-1)}$. We define the sequence $\bar{\sigma}_\Pi = \bar{\sigma}_0, \ldots, \bar{\sigma}_N$ with $\bar{\sigma}_i \in \mathbb{F}p^\times \times \{0,1\}^{n \cdot w}$ recursively as

$$\bar{\sigma}_0 = \dot{\rho} \circ \pi_0((0, 0^{w \cdot n})) \quad \Big(= \rho\big(\pi_0((0,0^{w\cdot n}))^{(1)}\big) \| \pi_0((0,0^{w\cdot n}))^{(2\ldots n+1)}\Big) \text{ and}$$

$$\text{for } i > 0 \; : \; \bar{\sigma}_i = \dot{\rho} \circ \dot{\pi}_i(\bar{\sigma}_{i-1}) \left(= \rho\big(\pi_i(\bar{\sigma}_{i-1}^{(1\ldots\tilde{i})})^{(1)}\big) \| \pi_i(\bar{\sigma}_{i-1}^{(1\ldots\tilde{i})})^{(2\ldots\tilde{i})} \| \bar{\sigma}_{i-1}^{(\tilde{i}+1\ldots n+1)}\right).$$

See Fig. 3 for an illustration of the computation of $\bar{\sigma}_\Pi$. Defining $\Pi' = \{\pi'_i\}_{i \in [N]_0}$ by $\pi'_i = \dot{\rho} \circ \pi_i$, it holds $\bar{\sigma}_\Pi = \sigma_{\Pi'}$. Thus, using $\bar{\sigma}_\Pi$ in our protocol results in a PoSW that is secure in the random permutation model, almost as efficient as the construction from [CP18], and at the same time achieves verification of correctness as efficient as in sloth. More formally, the efficiency of the combined scheme can be analysed as follows: First, consider the proof size:

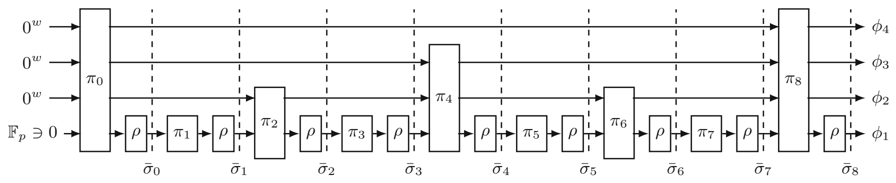

**Fig. 3.** Illustration of the computation of $\bar{\sigma}_\Pi = (\bar{\sigma}_0, \ldots, \bar{\sigma}_N)$ with $n = 3, N = 2^n = 8$.

$$|\chi| = w \cdot n, \quad |\phi| = \log(p) + w \cdot n, \quad |\gamma| = t \cdot n, \quad |\{\sigma_i\}_{i,j}| \le t \cdot n(\log(p) + w \cdot n).$$

Hence, compared to [CP18], the proofs are by a factor $n = \log(N) = \log(T)$ larger. Next, consider the prover efficiency. To compute $\phi$, the prover needs to sequentially evaluate $N+1$ permutations $\pi'_i = \dot{\rho} \circ \pi_i$, $i = 0, \ldots, N$. Storing only the $N/K$ states $\sigma_i$ with $K|i$ for some $K = 2^k$, the prover can answer the challenge $\gamma$ after $K$ parallel invocations to permutations $(\pi'_i)^{-1}$. Note, by construction computing $(\pi'_i)^{-1}$ is assumed to be by a factor $\log(p)$ faster than computing $\pi'_i$. The verifier, on the other hand, only needs $t \cdot n$ evaluations of $(\pi'_i)^{-1}$ to verify the PoSW. Also verification of correctness can be done in backwards direction by sequentially invocating $(\pi'_i)^{-1}$ for $i = N, \ldots, 0$, which is assumed to be by a factor $\log(p)$ faster than the computation of the prover.

When applied to a blockchain, our new PoSW allows extremely efficient rejection of wrong proofs while additionally providing sloth-like verification of correctness, which can be used whenever two or more distinct proofs pass the verification.

## A Proof of Lemma 1

*Proof (of Lemma 1).* Let $X = (X_1, \ldots, X_q)$ be the random variable corresponding to the responses to the queries of $\mathsf{A}^{\pi,\pi^{-1}}$ and $Y = (Y_1, \ldots, Y_q)$ the one corresponding to the responses to the queries of $\mathsf{A}^{F_1,F_2}$. We will show that $\Delta_{SD}(X,Y) \leq \frac{q(q-1)}{2^w}$. The lemma then follows from standard properties of $\Delta_{SD}$.

In the following, we will abbreviate the conditional distributions $(X_i|X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$ as $(X_i|(x_1, \ldots, x_{i-1}))$ and similarly for $Y$. From subadditivity for joint distributions (a property of $\Delta_{SD}$), we have

$$\Delta_{SD}(X,Y) \leq \sum_{i=1}^{q} \max_{x=(x_1,\ldots,x_{i-1})} \Delta_{SD}(X_i|x, Y_i|x).$$

For each particular $i$ we have

$$\Delta_{SD}(X_i|x, Y_i|x) = \frac{1}{2} \sum_{y \in \{0,1\}^w} |\mathsf{Pr}\,(X_i = y|x) - \mathsf{Pr}\,(Y_i = y|x)|.$$

From the definition of $F_1, F_2$, it is clear that $\mathsf{Pr}\,(Y_i = y|x) = 2^{-w}$ for all $y \in \{0,1\}^w$ and $x \in (\{0,1\}^w)^{i-1}$. For the other case, notice that any query to $\pi$ or $\pi^{-1}$ fixes a particular input/output pair. Accordingly, $X_i$ is uniform among the remaining $2^w - (i-1)$, no matter if $\pi$ or $\pi^{-1}$ was queried (recall that no input/output pair is repeated). It follows that

$$\begin{aligned}\Delta_{SD}(X_i|x, Y_i|x) &= \frac{1}{2}\left[\frac{i-1}{2^w} + (2^w - (i-1))\left(\frac{1}{2^w - (i-1)} - \frac{1}{2^w}\right)\right] \\ &= \frac{i-1}{2^w}\end{aligned}$$

for any $x$ (in particular, the maximum). Summing over all $i$ yields the final bound. □

## References

[BBBF18] Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part I. LNCS, vol. 10991, pp. 757–788. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_25

[CP18] Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 451–467. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_15

[CPS08] Coron, J.-S., Patarin, J., Seurin, Y.: The random oracle model and the ideal cipher model are equivalent. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 1–20. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85174-5_1

[DGK17] Dodis, Y., Guo, S., Katz, J.: Fixing cracks in the concrete: random oracles with auxiliary input, revisited. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 473–495. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_16

[DSKT16] Dachman-Soled, D., Katz, J., Thiruvengadam, A.: 10-round Feistel is indifferentiable from an ideal cipher. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 649–678. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_23

[Fis18] Fisch, B.: PoReps: proofs of space on useful data. IACR Cryptology ePrint Archive 2018/678 (2018)

[Fis19] Fisch, B.: Tight proofs of space and replication. In: Advances in Cryptology - EUROCRYPT 2019 (2019)

[FMPS19] De Feo, L., Masson, S., Petit, C., Sanso, A.: Verifiable delay functions from supersingular isogenies and pairings. Cryptology ePrint Archive, Report 2019/166, 2019. https://eprint.iacr.org/2019/166

[HKT11] Holenstein, T., Künzler, R., Tessaro, S.: The equivalence of the random oracle model and the ideal cipher model, revisited. In: Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing, STOC 2011, pp. 89–98, ACM, New York (2011)

[LW17] Lenstra, A.K., Wesolowski, B.: Trustworthy public randomness with sloth, unicorn, and trx. IJACT **3**(4), 330–343 (2017)

[May93] May, T.C.: Timed-release crypto (1993). http://www.hks.net/cpunks/cpunks-0/1460.html

[MMV11] Mahmoody, M., Moran, T., Vadhan, S.: Time-lock puzzles in the random oracle model. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 39–50. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_3

[MMV13] Mahmoody, M., Moran, T., Vadhan, S.: Publicly verifiable proofs of sequential work. In: Proceedings of the 4th Conference on Innovations in Theoretical Computer Science, ITCS 2013, pp. 373–388, ACM, New York (2013)

[Pie19a] Pietrzak, K.: Proofs of catalytic space. In: 10th Innovations in Theoretical Computer Science Conference, ITCS 2019, 10–12 January 2019, San Diego, California, USA, pp. 59:1–59:25 (2019)

[Pie19b] Pietrzak, K.: Simple verifiable delay functions. In: 10th Innovations in Theoretical Computer Science Conference, ITCS 2019, 10–12 January 2019, San Diego, California, USA, pp. 60:1–60:15 (2019). https://eprint.iacr.org/2018/627

[RSW00] Rivest, R.L., Shamir, A., Wagner, D.: Time-lock puzzles and timed-release crypto. Technical report MIT/LCS/TR-684, MIT, February 2000

[Wes19] Wesolowski, B.: Efficient verifiable delay functions. In: Advances in Cryptology - EUROCRYPT 2019 (2019)