# Optimal Proofs of Sequential Work

Krzysztof Pietrzak
ISTA
pietrzak@ista.ac.at

Leonid Reyzin
Boston University
reyzin@bu.edu

October 20, 2026

**Abstract**

In a proof of sequential work (PoSW), introduced by Mahmoody, Moran and Vadhan [MMV'13], a verifier can be convinced that a prover performed a computation of $T$ sequential steps. Such a proof is $(\delta, \lambda)$-sound if a malicious prover making at most $(1-\delta)T$ sequential steps can produce a valid proof only with probability $2^{-\lambda}$, even when using massive parallelism.

Besides time-lock puzzles (TLPs) and verifiable delay functions (VDFs), PoSW are one of the central primitives in timed cryptography. Unlike TLPs or VDFs, PoSW can be constructed just from hash functions, with a security proof in the random oracle model. Existing PoSW are based on graph labeling: the prover must sequentially compute a labeling of a graph of $T$ nodes, where each node's label is the hash of its parents' labels.

The [MMV'13] construction uses node-depth-robust graphs. A simple and much more efficient construction was later given by Cohen and Pietrzak [CP'18]. In the [CP'18] PoSW, getting $(\delta, \lambda)$ soundness requires transmitting $\Omega((\lambda \log T)/\delta)$ hash outputs. The underlying graph has average in-degree $\Theta(\log T)$, so labeling takes $\Theta(T \log T)$ total hash inputs.

Our first result is a lower bound: any efficient graph-labeling-based PoSW that is $(\delta, \lambda)$-sound requires the prover to send $\Omega((\lambda \log T)/\delta)$ labels. To prove this result, we relate PoSW soundness to a cut-depth game and bound this game using a refinement of Valiant's depth-reduction argument. This shows that [CP'18] achieves asymptotically optimal communication and verifier hash-input complexity, while leaving open whether its $\Theta(\log T)$ average indegree (and thus $\Theta(T \log T)$ prover hash-input complexity) is necessary.

Our second result is a new PoSW that matches these lower bounds up to constant factors and has maximum indegree two. In particular, it reduces the honest prover's total hash-input complexity from $\Theta(T \log T)$ to $\Theta(T)$ while retaining $O((\lambda \log T)/\delta)$ communication and verifier hash-input complexity.

# 1 Introduction

**Proofs of Work.** At Crypto 1992, Dwork and Naor [DN93] presented *Pricing via processing or combatting junk mail*, which introduced proofs of work (PoW) as a system to deter junk mail and denial of service attacks. The general idea is to require a client requesting some service (or a sender of an email) to perform a moderately expensive computation in order to solve a puzzle, while the recipient should be able to verify the solution efficiently, thereby making access to the service somewhat costly.

Back [Bac02] proposed a particularly simple construction, Hashcash, whose approach was later used in Bitcoin. The puzzle is simply a random string $\chi$, while the proof is a nonce $\pi$ such that $H(\chi, \pi)$ has some special form, say, starting with $d$ zeros. If $H$ is modeled as a random oracle, finding such a $\pi$ requires $2^d$ hash evaluations in expectation, while a solution can be verified with a single hash query.

**Timed Cryptography.** In a 1993 Cypherpunks mailing-list post, Timothy May envisioned cryptographic mechanisms for "sending information into the future." This vision was later realized by Rivest, Shamir and Wagner [RSW96], who proposed a new primitive called time-lock puzzles (TLPs).

In timed cryptography, time is captured by an *inherently sequential* computation, which is a computation performed in some $T$ steps, where each step is fairly simple and can be implemented on basic hardware, say a single CPU. Here, *inherently sequential* means that even when given more powerful hardware, in particular when allowing for high parallelism, the computation still basically requires $T$ sequential steps.

**Proofs of Sequential Work.** Mahmoody, Moran and Vadhan [MMV13] define and construct a *publicly verifiable proof of sequential work* (PoSW), which, using ideas of timed cryptography, is defined as a PoW with the additional constraint that the work must be done sequentially.

Concretely, on input a salt $\chi$ and time parameter $T$, the prover must compute an efficiently verifiable proof $\pi$. For soundness, we require that for some sequentiality gap $\delta$ and some $\lambda$, a malicious prover who makes at most $T(1-\delta)$ *sequential* steps will fail to produce a valid proof except with probability $2^{-\lambda}$. The authors of [MMV13] motivated their notion of PoSW with applications like non-interactive timestamping and verifiable benchmarks.

**Verifiable Delay Functions.** Let us also mention the most recent timed-cryptography primitive, namely verifiable delay functions (VDFs) [BBBF18]. They are defined similarly to PoSW, but with the additional requirement that there's some unique output: a VDF on input $(\chi, T)$ outputs a tuple $(y, \pi)$, where $\pi$ certifies that $y$ is the correct output. This uniqueness property is crucial in some applications like randomness beacons [Rab83, BBBF18], lotteries [GS98], or blockchain consensus [CP19]. While PoSW have security proofs in the random oracle model, all VDFs (and also TLP) constructions need more structured assumptions, and this seems inherent as shown by [MSW20, GRY25, AAK+25].

**PoSW Based on Graph Labeling.** Constructions [MMV13, CP18, AKK+19] of PoSW are based on *graph labeling*. The construction is specified by a DAG $G = (V, E)$, with $|V| = T$, a single sink $\tau \in V$, and a hash function $H$.[1] For every node $v$, $\mathsf{pred}(v)$ denotes the immediate predecessors of $v$, i.e., nodes $u$ such that $(u, v) \in E$.

- The verifier sends a random salt $\chi$ to the prover.
- The prover sequentially computes the labeling of the nodes of $V$ using the salted hash $H_\chi(\cdot) = H(\chi, \cdot)$, where the label $\ell_v$ of a node $v$ is the hash of the labels of its predecessors (arranged in a fixed order)

  $$\ell_v = H_\chi(v, (\ell_u)_{u \in \mathsf{pred}(v)}) .$$

  The prover sends the label $\phi = \ell_\tau$ of the sink to the verifier. Think of $\phi$ as a commitment to the labels. [2]
- The verifier samples and sends to the prover a challenge $\gamma \leftarrow \mathcal{D}$ (according to some distribution $\mathcal{D}$), which tells the prover which nodes to open.
- The prover returns purported labels $z_v$ for all $v$ in a set $\mathsf{ProverSet}(\gamma)$. One can think of these labels as an opening of the labels in $\gamma$; if the prover is honest, $z_v = \ell_v$ (while a dishonest prover may not even know the correct $\ell_v$).
- Using those labels, the verifier recomputes purported labels $z_v$ for all $v$ in some set $\mathsf{VerifierSet}(\gamma)$. This set must contain the sink $\tau$; it must also satisfy that

  $$\forall v \in \mathsf{VerifierSet}(\gamma), \mathsf{pred}(v) \subseteq \mathsf{ProverSet}(\gamma) \cup \mathsf{VerifierSet}(\gamma)$$

  so that computation is possible, and that $\mathsf{VerifierSet}(\gamma) \cap \mathsf{ProverSet}(\gamma) = \emptyset$, so that computation is unambiguous. The verifier accepts if the recomputed sink label $z_\tau$ matches the commitment $\phi$.

The verifier can always simply recompute the labels for the entire graph and check whether $\phi = \ell_\tau$, which suffices to verify that the prover almost certainly must have made $T$ (assuming $G$ has depth $|V| = T$) sequential queries. So in principle $\mathsf{ProverSet}(\gamma)$ can be empty. However, by requiring the prover to send some labels to help the verifier and allowing for a sequentiality gap $\delta > 0$ we can dramatically decrease verifier time, achieving $|\mathsf{ProverSet}(\gamma)|$ and $|\mathsf{VerifierSet}(\gamma)|$ in the order of $\log(T) \cdot \lambda/\delta$.

Let us illustrate efficiency on (a toy example with 31 nodes of) the construction from [CP18] as depicted in Figure 1. This graph has a source with index 0000 and a sink with

[1] [AKK+19] is somewhat different as there $H$ is an invertible permutation, modeled as an ideal permutation in the proof, but the proof goes through even if it's a normal length-preserving hash function. The invertibility is just a special feature that enables recomputation of the input from a correctly computed output.

[2] The fact that the prover sends exactly the label of the sink might seem somewhat restrictive. We use this definition as it captures the existing constructions considered here and our new construction. As we'll discuss in the "Cut-Depth vs. PoSW" paragraph below, the lower bound we prove applies to more general constructions.

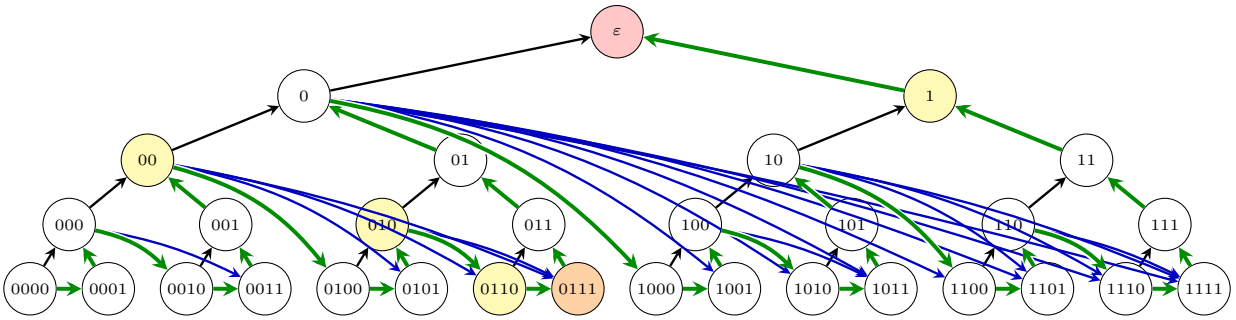



Figure 1: The graph underlying the PoSW of [CP18]. The green edges form its Hamiltonian path from 0000 to the root. To open the orange node 0111, the prover supplies the labels of the yellow co-path nodes $0110, 010, 00, 1$, and the verifier recomputes the labels $0111, 011, 01, 0, \varepsilon$ on the path from the leaf to the root.

index $\epsilon$, and a path (shown in green) connecting the source to the sink while visiting all nodes. If challenged to open the label of, say, node 0111 (highlighted in orange), the prover would provide the purported labels on the co-path from 0111 to the root (highlighted in yellow), and the verifier would recompute the labels on the path from 0111 to the sink, and check if the recomputed sink matches the commitment.

An honest prover will compute the $T$ (in the toy example $T = 31$) labels sequentially. In [CP18] it's shown that in the random oracle model, if the prover only makes $T(1-\delta)$ sequential query rounds to the random oracle (each round may contain many parallel queries, and each query may contain many inputs), then it will be able to open a random leaf label with probability at most $(1-\delta)$ (apart from an additive term that captures a bad event which happens if we see a random oracle collision or a random oracle output is guessed, i.e., it's used as input before it was received as an output). To get soundness $2^{-\lambda}$ (plus this error), it therefore suffices to open $t$ independently sampled leaves with $(1-\delta)^t \leq 2^{-\lambda}$. For $0 < \delta < 1$, we may take $t = \lceil (\lambda \ln 2)/(-\ln(1-\delta)) \rceil$; the simpler choice $t = \lceil (\lambda \ln 2)/\delta \rceil$ also suffices.

**Efficiency of CP'18.** Let us discuss the main efficiency properties of [CP18]; they are also illustrated in Table 1. During the labeling phase, the prover must compute $T$ sequential hash queries, which is clearly optimal. But not only the number of hash queries is relevant; so is the length of their inputs, which ideally should be as small as possible. The input length depends on the indegree (as we must hash the labels of all parents), and the average indegree of the graph is[3] $\Theta(\log T)$.

As mentioned above, the prover supplies $t = \Theta(\lambda/\delta)$ openings, each requiring $\Theta(\log T)$ transmitted labels and $\Theta(\log T)$ predecessor labels hashed by the verifier. Thus communication and verifier hash-input complexity are $\Theta((\lambda \log T)/\delta)$.

It is also important to consider prover memory during the computation. As explained

[3]For $N = 2^d$ leaves and $T = 2N - 1$ nodes, the graph has $2(N-1)$ tree edges and $Nd/2$ additional edges. Its average indegree is therefore $(2(N-1) + Nd/2)/(2N-1) = d/4 + O(1)$.

| | Lower bound | CP'18 [CP18] | Diamond |
|---|---|---|---|
| Honest-prover hash evaluations | $\Omega(T)$ | $\Theta(T)$ | $\Theta(T)$ |
| Honest-prover hash-input complexity | $\Omega(T)$ | $\Theta(T \log T)$ | $\Theta(T)$ |
| Maximum indegree | — | $\Theta(\log T)$ | 2 |
| Verifier hash-input complexity | $\Omega((\lambda \log T)/\delta)^\dagger$ | $O((\lambda \log T)/\delta)$ | $O((\lambda \log T)/\delta)$ |
| Communication complexity | $\Omega((\lambda \log T)/\delta)^\dagger$ | $O((\lambda \log T)/\delta)$ | $O((\lambda \log T)/\delta)$ |

Table 1: Graph-based PoSW with soundness error $2^{-\lambda}$ against provers using at most $(1-\delta)T$ sequential random oracle query rounds, suppressing the additive random-oracle error for the constructions. Hash-input complexity counts predecessor-label occurrences hashed, and communication counts prover-to-verifier hash outputs. The bounds marked † hold in the efficient-verification regime specified in the text.

in [CP18, Section 5.3], ideally the prover should not have to store all the labels waiting to answer the verifier's query. The construction allows the prover to store $O(\sqrt{T})$ labels during the computation and answer the queries by performing $O(t\sqrt{T})$ computation, which is tiny compared to the $T$ computation needed for the PoSW itself.

**Our Contributions.** In this work we prove a lower bound showing that, for any graph-based PoSW in the regime specified below, communication and verifier hash-input complexity are $\Omega((\lambda \log T)/\delta)$ to achieve $2^{-\lambda}$ security against a prover with a $\delta$ sequentiality gap, i.e., making at most $(1-\delta)T$ sequential random oracle query rounds before commitment. Our lower bound holds for $0 < \delta \leq 1/8$ in what we call the "efficient verification regime": a sufficient condition (see Corollary 1) is that for some $\epsilon > 0$, we have $m \cdot s \leq T^{1-\epsilon}$ and $s/\delta \leq T^{1-\epsilon}$. This is a very mild restriction as we're typically interested in verifiers with a complexity that is at most polylogarithmic in $T$, which falls in this regime.

With respect to the parameters discussed above, the only remaining question is whether the $\Theta(\log T)$ average indegree (and thus prover hash-input complexity) is necessary or can be improved. As our second contribution we show that the indegree can indeed be reduced. We propose a new construction called Diamond (referring to the shape of the underlying graph as illustrated in Fig. 2) which is based on a DAG with maximum indegree 2 (which is the best possible), and in the other parameters above (communication and verifier computation) is as efficient as [CP18]. In particular, the honest prover hashes $\frac{5}{3}T$ predecessor labels in total, compared with $\Theta(T \log T)$ in [CP18].

**Intuition for the Lower Bound.** We first reduce PoSW soundness to a *cut-depth game*. Before the verifier's challenge $\gamma$ is sampled, a cutter chooses edges whose removal reduces the graph's depth from $T$ to at most $(1-\delta)T$. It wins if the verifier recomputes no vertex with a deleted incoming edge. Such a cut gives a cheating prover: replace the labels on deleted dependencies by dummy values and compute the resulting labeling in the shallower graph. Whenever the cutter wins, all computations performed by the verifier agree with this

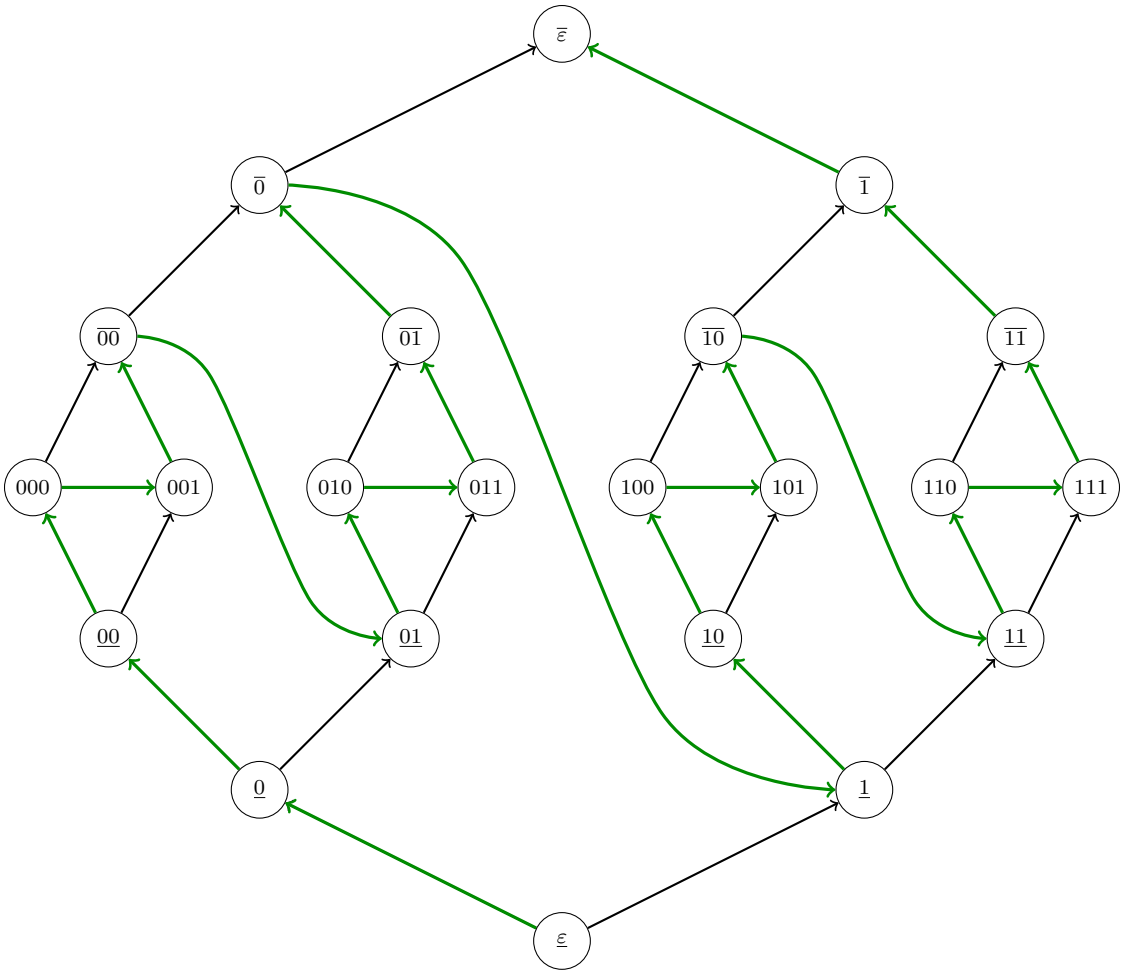



Figure 2: The graph $\mathsf{Diamond}_3$. The green edges form its unique Hamiltonian path from the source $\underline{\varepsilon}$ to the sink $\overline{\varepsilon}$.

labeling, so the verifier will accept.

To bound the cutter's success probability, we use a proof similar to Valiant's bound on the edge depth-robustness [Val77]. Valiant showed how to partition the edges of a DAG of depth $T$ into $\log T$ bins, so that removing all the edges in one of the bins will reduce the depth of the DAG to $T/2$ (as high-level intuition, think of the bins as containing edges of decreasing lengths, with the $i$th bin containing edges of length up to $2^{\log(T)-i}$, more precisely, it contains edges $(u, v)$, where $u, v$ are at depth $p0a$ and $p1b$ for some $i$ bit prefix $p$. Within a bin $i$ we cluster the edges into classes $E_{i,p}$ depending on the prefix $p$). We can't just remove an entire bin, as by checking one edge in every bin the verifier would catch the cutter with probability 1. Instead, we delete, say a random half of the edge classes in one random bin, which will cut the depth to $\frac{3}{4}T$ (this is illustrated in Figure 3). Now, the verifier needs to check $\lambda$ random edges in each bin to catch the cutter with probability $1 - 2^{-\lambda}$. We extend

this argument from the number of edges checked by the verifier to the number of nodes sent by the prover.

For technical reasons, this argument cannot use the bins with the shortest prefixes, as they have too few edge classes for random selection to work, and this is the reason our bound only applies in the efficient verifier regime.

**Intuition for the New Construction.** The graph underlying our new Diamond construction, as illustrated in Figure 2, consists of two binary trees with their $N$ leaves identified, directed from the lower root towards the leaves and from the leaves towards the upper root. At every recursive split, a cross edge joins the sink of the left subgraph to the source of the right, creating a Hamiltonian path through all $T = 3N - 2$ vertices while keeping indegree at most two. To answer a random leaf challenge, the prover supplies the upper-tree co-path labels, omitting the leaf's sibling: the verifier recomputes both sibling leaves, together with the lower and upper paths. The same co-path labels supply the cross-edge inputs, so verification uses only $O(\log T)$ labels. For security, the commitment fixes canonical upper-tree labels and hence the lower-tree computation. Except for random-oracle collisions or lucky guesses, an accepted opening certifies that every recomputed node was consistently queried before commitment. If $B > 0$ leaves cannot be opened consistently, the recursive graph contains a consistent path of at least $3(N - B)$ vertices; if $B = 0$, it contains the full Hamiltonian path. A prover committing within $(1 - \delta)T$ rounds must therefore have $B/N \geq \delta$. Using $t$ independent challenges gives error at most $(1 - \delta)^t$, apart from the random-oracle error.

**Cut-Depth vs. PoSW.** A successful strategy in the cut-depth game yields an attack on the corresponding PoSW, but hardness of the cut-depth game alone does not establish PoSW soundness. In the cut-depth game, the cutter chooses dependencies to remove and loses whenever the verifier tests one of them. In a PoSW, detecting a violated dependency additionally requires the labels used in the test to be tied to the commitment. In [CP18] and our construction, this is achieved by Merkle-type openings that connect the tested computation to the committed sink label.

Despite this asymmetry, it turns out the cut-depth game is a good abstraction for labeling based PoSW as there actually exists a PoSW construction which matches the lower bound for the cut-depth graph. The intuition for this fortunate fact seems to be that when one asks for a (Merkle-type) opening of a leaf (in [CP18] or our construction), the $\log T$ labels provided verify an edge in each of the $\log T$ bins considered in the lower bound proof. Thus, even if only one of the bins contains a constant fraction of bad edges, to hit such a bad edge with probability $1 - 2^{-\lambda}$, we just need to open $\Theta(\lambda)$ leaves, where opening requires sending $\log T$ labels.

The fact that the lower bound holds for cut-depth game implies that we can't hope to get better constructions for a much broader set of PoSW constructions than the hash-based construction considered in this paper. For example, assume we let the prover commit to all the labels using a vector commitment with constant-size openings like KZG [KZG10] (which require algebraic assumptions, not just hash functions). So, to open an arbitrary label the

prover just needs to send one, and not $\log T$ values. As our cut-depth game also captures such constructions, we cannot hope to improve on e.g. the communication complexity when using such a construction.

# 2 Preliminaries

$H : \{0,1\}^* \to \{0,1\}^w$ denotes a random oracle; we consider that its inputs are tuples of bit strings, unambiguously encoded (rather than merely concatenated). For a salt $\chi$, write $H_\chi(x) := H(\chi, x)$. For a DAG $G = (V, E)$ and $v \in V$, let the *predecessors* of $v$ be the set of nodes $\mathsf{pred}(v) := \{u \in V : (u, v) \in E\}$. Let $T = |V|$.

For a salt $\chi$, the correct label $\ell_v$ of a node $v$ in this DAG is defined inductively (starting at the source(s)) as

$$\ell_v = H_\chi(v, (\ell_u)_{u \in \mathsf{pred}(v)}) ,$$

where $u \in \mathsf{pred}(v)$ are arranged in some canonical ordering.

We consider only single-sink DAGs; the label $\ell_\tau$ of the sink $\tau$ is what the prover sends to the verifier as $\phi$.

We work in the parallel random oracle query model, in which the running time of computation is measured in rounds, and each round can make as many random oracle queries in parallel as is desired. Of course, all queries in one round must be fixed before the answers to that round are returned, and a query depending on an answer from a previous round requires another sequential round. We bound the total number of queries (added up over all rounds of both the prover and the verifier) with a parameter $q$ (without such a bound, it would be possible to obtain all relevant information about $H_\chi$ in one step). We assume no queries to $H_\chi$ were issued before the salt $\chi$ was issued by the verifier.

**Graph-Based PoSW.** The structure of a PoSW protocol based on graph labeling (the only type of protocol considered in this paper) is described in Section 1. Recall that it is specified by a DAG $G$, the challenge distribution $\mathcal{D}$, and functions $\mathsf{ProverSet}$ and $\mathsf{VerifierSet}$.

The *soundness error* for a $\delta$-cheating prover is the probability that the verifier accepts if a malicious prover spends at most $(1 - \delta)T$ time (where time is measured in rounds, as described above, and $T = |V|$) before sending $\phi$. Note that the malicious prover can also issue random oracle queries after sending $\phi$, and in particular after seeing the verifier challenge $\gamma$.

We note that the protocol is described above as interactive, but, because it is public-coin, can be made noninteractive via the Fiat-Shamir heuristic, in which the prover and the verifier both use $H_\chi(v', \phi)$ (for some fixed value $v' \notin V$) to generate the challenge $\gamma$. (The soundness argument will need to be adjusted to account for the prover's ability to try multiple inputs to $H_\chi$.)

# 3 Lower Bounds for Graph-Based PoSWs

We first isolate the graph-theoretic property that is necessary for a graph-based proof of sequential work. Informally, suppose one can find a set of edges such that

- deleting this set substantially reduces the depth of the underlying DAG, and
- it is reasonably likely that the verifier will recompute no edge in this set,

then a malicious prover can simply ignore the dependencies indicated by these edges. The malicious prover will be able to compute node labels in parallel according to the shallower cut graph, with the knowledge that it is reasonably likely to not be caught.

We assume that the predecessors of every vertex are given in a fixed canonical order. The depth $\mathsf{depth}(G)$ is the maximum number of vertices on a directed path in $G$. This convention is convenient because a graph labeling can be computed in exactly $\mathsf{depth}(G)$ parallel random-oracle rounds if the source labels are also obtained by random-oracle queries. (One could change to the convention that path length counts edges rather than nodes, but it is less relevant here, because the cost of hash computation is incurred at the nodes; in any case, it would affect parameters by just 1.)

## 3.1 The cut-depth game

The malicious prover will play the following combinatorial game against the verifier. For this game, the prover will be the *cutter* and the verifier will be the *challenger*. For a set $R \subseteq V$ of vertices, define $\mathsf{Inc}(R)$ as the set of all edges that end in $R$, i.e.,

$$\mathsf{Inc}(R) = \{(u, v) \in E \ : \ v \in R\}$$

**Definition 1** (Cut-depth game)**.** Recall that $G = (V, E)$ has $|V| = T$, $\mathcal{D}$ is a distribution over challenges $\gamma$, and $\mathsf{VerifierSet}$ maps $\gamma$ to a subset of $V$. Let $\delta \in (0, 1)$. The $\delta$-cut-depth game $\mathsf{Cut}_{G,\mathcal{D},\delta}$ is defined as follows.

1. The cutter $\mathcal{A}$ outputs a set $F \subseteq E$ satisfying

$$\mathsf{depth}(G - F) \leq (1 - \delta) \cdot \mathsf{depth}(G).$$

2. Independently, the challenger samples $\gamma \leftarrow \mathcal{D}$.
3. The cutter wins if $F$ contains no incoming edges for any element of $\mathsf{VerifierSet}(\gamma)$, i.e.,

$$\mathsf{Inc}(\mathsf{VerifierSet}(\gamma)) \cap F = \emptyset\,.$$

Note that while the challenge $\gamma$ must be chosen at random as otherwise the cutter could easily win, an optimal strategy for the cutter is to choose a fixed cut $F$ that maximizes its winning probability.

## 3.2 From cuts to PoSW attacks

The next lemma shows how to transform a cutter who succeeds with probability $\epsilon$ to a $\delta$-cheating prover with the same success probability.

**Lemma 1** (Cutting attacks graph-based PoSWs)**.** *Let $\Pi$ be a graph-based PoSW based on $(G, \mathcal{D})$. For every cutter $\mathcal{A}$ in $\mathsf{Cut}_{G,\mathcal{D},\delta}$ who wins with probability $\epsilon$, there is a malicious prover $\widetilde{P}^{\mathcal{A}}$ that computes its commitment in at most $(1-\delta)\cdot\mathsf{depth}(G)$ sequential random-oracle rounds using $T$ total random oracle queries, and causes the verifier to accept with probability at least $\epsilon$.*

*Proof.* The malicious prover first runs $\mathcal{A}$ to obtain a cut $F \subseteq E$ such that

$$\mathsf{depth}(G-F) \leq (1-\delta)\cdot\mathsf{depth}(G).$$

For every cut edge $e = (u, v) \in F$, it samples an independent dummy value $r_e \leftarrow \{0,1\}^w$. The prover now computes a "cut label" $\widetilde{\ell}_v$ for every node $v \in V$ as

$$\widetilde{\ell}_v := H_\chi\big(v, (x_{u,v})_{u\in\mathsf{pred}(v)}\big) \tag{1}$$

where

$$x_{u,v} := \begin{cases} \widetilde{\ell}_u, & (u,v) \notin F, \\ r_{(u,v)}, & (u,v) \in F. \end{cases}$$

The only dependencies between queries in Equation (1) are the uncut edges $E \setminus F$. Consequently the queries can be scheduled according to the layers of $G - F$, and hence the entire cut labeling, in particular $\widetilde{\ell}_\tau$, can be computed in at most $\mathsf{depth}(G-F) \leq (1-\delta)\cdot\mathsf{depth}(G)$ sequential rounds. The prover sends

$$\phi := \widetilde{\ell}_\tau.$$

After receiving a challenge $\gamma$, the prover returns $z_v := \widetilde{\ell}_v$ for every $v \in \mathsf{ProverSet}(\gamma)$. Suppose that the cutter wins. Then for every vertex $v \in \mathsf{VerifierSet}(\gamma)$, no incoming edge was cut; hence $x_{u,v} = \widetilde{\ell}_u$ for every $u \in \mathsf{pred}(v)$, and the verifier computation of the labels matches the prover computation, producing $z_v = \widetilde{\ell}_v$ for every $v \in \mathsf{VerifierSet}(\gamma)$, including $z_\tau = \widetilde{\ell}_\tau = \phi$. Thus, the verifier accepts.

□

## 3.3 A universal cut strategy

It remains to show that every DAG admits cuts that are difficult to hit with a small number of tested edges. Our argument is a randomized variant of Valiant's classical depth-reduction argument [Val77].

Define the *rank* $\rho(v)$ of a node $v$ as the length (in edges) of the longest path that ends in that node. For every edge $(u, v)$, we have $\rho(u) < \rho(v)$. The maximum rank is $\mathsf{depth}(G) - 1$. Denote $\mathsf{depth}(G)$ by $n$ and assume it is a power of two, $n = 2^d$. We will write ranks in binary, as $d$-bit strings.

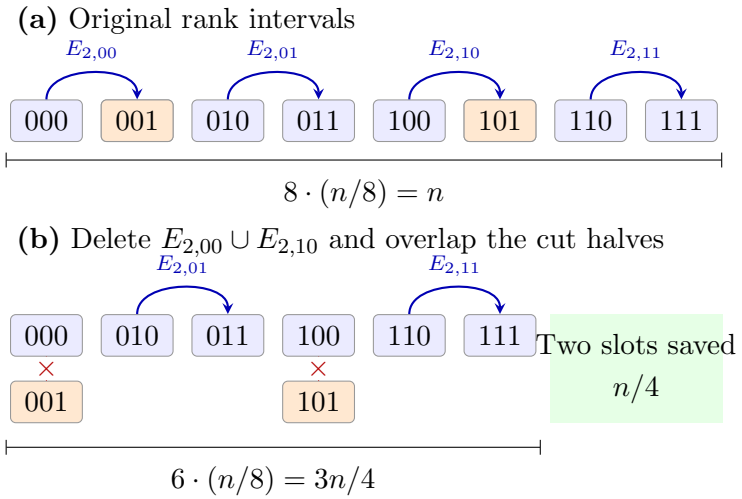



Figure 3: The depth-reduction step in the proof of Lemma 2, illustrated at layer $i = 2$. A block labelled $b \in \{0,1\}^3$ contains all vertices whose $d$-bit rank $\rho(v)$ starts with $b$; each block spans $n/8$ rank values, but need not contain $n/8$ vertices. In (a), each arrow represents the entire edge class $E_{2,p}$ from block $p0$ to block $p1$. Edges within blocks and between different prefix pairs are omitted. In (b), we choose $R = \{00, 10\}$ and delete $F = \bigcup_{p \in R} E_{2,p}$. For each selected prefix, no remaining directed path can visit both halves: ranks increase along every edge, and any crossing between the halves would use a deleted edge. We therefore place the right half below the left half, giving them the same horizontal rank interval. Each selected prefix saves $n/8$ rank positions, so $\mathsf{depth}(G - F) \le n - |R|n/8 = 3n/4$. This illustrates the general bound $\mathsf{depth}(G - F_{i,R}) \le n - |R|n/2^{i+1}$ used by the randomized cutter.

Partition all edges into layers, as follows. For an edge $(u, v)$, put it in *layer* $i$ if the ranks $\rho(u)$ and $\rho(v)$ agree on exactly $i$ most significant bits. Further, put it in *class* $E_{i,p}$ for $p \in \{0,1\}^i$ if this prefix is $p$; thus, there are $2^i$ classes in every layer, and in every class, every edge changes the $(i+1)$th most significant bit of rank from 0 to 1, keeping more significant bits untouched.

We note that our argument assumes $\delta \le 1/8$. We present it assuming $n$ is a power of 2 for ease of exposition; this restriction is not essential, and the argument can be easily adapted to other $n$.

**Lemma 2.** *Let $G = (V, E)$ be a DAG of depth $n = 2^d$. Assume a fixed* VerifierSet *is such that* Inc(VerifierSet) *contains edges from at most $k$ distinct classes $E_{i,p}$ for $i < d - a$ for some $a$ (i.e., we ignore a lowest layers—intuitively, because the verifier set can easily contain too many edges from such layers, because they don't take long to compute). Fix $0 < \delta \le 1/8$, and let*

$$i_0 := \left\lceil \log_2\big(\max\{4k, 1/\delta\}\big) \right\rceil, \qquad L := d - a - i_0.$$

*Assume $L > 0$. Then there is a randomized cutter that always outputs a set $F$ satisfying*

$$\mathsf{depth}(G - F) \le (1 - \delta)n$$

*and*

$$\Pr[\mathsf{Inc}(\mathsf{VerifierSet}) \cap F = \emptyset] \geq \exp\left(-\frac{8\delta k}{L}\right). \tag{2}$$

*Consequently, the same bound holds for a randomly chosen verifier set, as long as for every choice,* $\mathsf{Inc}(\mathsf{VerifierSet})$ *contains edges from at most* $k$ *distinct classes* $E_{i,p}$*.*

*Proof.* (Figure 3 illustrates the proof idea on a concrete example.)

Fix a layer $i$. There are $B_i := 2^i$ prefixes $p$. Given a set of classes $R \subseteq \{0,1\}^i$ in the $i$th layer, define

$$F_{i,R} := \bigcup_{p \in R} E_{i,p}$$

as the set of edges that belong to these classes. For an $i$-bit-long prefix $p$, the ranks beginning with $p$ form an interval of $n/2^i$ consecutive ranks, split by the next bit into two equal halves. After deleting $E_{i,p}$, no path can contain vertices whose ranks lie in both halves: the first edge on such a path crossing from the lower half to the upper half would belong to $E_{i,p}$. Therefore, for every $R$,

$$\mathsf{depth}(G - F_{i,R}) \leq n - |R| \cdot \frac{n}{2^{i+1}}. \tag{3}$$

We consider *eligible* layers to be layers $i \in \{i_0, \ldots, d-1-a\}$. We avoid $i_0$ most significant layers, as there are not enough classes in such layers, and the $\mathsf{Inc}(\mathsf{VerifierSet})$ could contain edges from many of them. We avoid $a$ least significant layers, because we don't know anything about how much $\mathsf{Inc}(\mathsf{VerifierSet})$ contains in these layers. The cutter picks an eligible layer at random. The cutter then removes edges for a random $2\delta$ fraction of classes. To be precise, let

$$s_i := \lceil 2\delta B_i \rceil \ .$$

The cutter randomly picks an $s_i$-element subset $R \subseteq \{0,1\}^i$ and outputs $F := F_{i,R}$. By Equation (3),

$$\mathsf{depth}(G - F) \leq n - s_i \frac{n}{2B_i} \leq (1-\delta)n,$$

so every set output by the cutter cuts the depth as required.

It remains to bound the probability of hitting $\mathsf{Inc}(\mathsf{VerifierSet})$. Let $k_i$ be the number of classes in layer $i$ that intersect with $\mathsf{Inc}(\mathsf{VerifierSet})$. Because the classes $E_{i,p}$ partition $E$,

$$\sum_{i=0}^{d-1-a} k_i \leq k. \tag{4}$$

Conditioned on choosing layer $i$, the cut avoids $\mathsf{Inc}(\mathsf{VerifierSet})$ exactly when none of the $k_i$ prefixes touched by $\mathsf{Inc}(\mathsf{VerifierSet})$ is selected. Thus

$$\begin{aligned}\Pr[F \cap \mathsf{Inc}(\mathsf{VerifierSet}) = \emptyset \mid i] &= \frac{\binom{B_i - k_i}{s_i}}{\binom{B_i}{s_i}} \\ &= \prod_{j=0}^{k_i - 1} \left(1 - \frac{s_i}{B_i - j}\right). \end{aligned} \tag{5}$$

For an eligible layer, $B_i \geq 4k$ and $B_i \geq 1/\delta$. Hence $k_i \leq B_i/4$ and

$$\frac{s_i}{B_i} \leq 2\delta + \frac{1}{B_i} \leq 3\delta.$$

Consequently, for every $j < k_i$,

$$\frac{s_i}{B_i - j} \leq \frac{3\delta}{3/4} = 4\delta \leq \frac{1}{2}.$$

Using $\ln(1-x) \geq -2x$ for $0 \leq x \leq 1/2$ in Equation (5) gives

$$\Pr[F \cap \mathsf{Inc}(\mathsf{VerifierSet}) = \emptyset \mid i] \geq \exp(-8\delta k_i).$$

Averaging over the uniformly chosen layer and using convexity of $x \mapsto e^{-8\delta x}$,

$$\begin{aligned}\Pr[F \cap \mathsf{Inc}(\mathsf{VerifierSet}) = \emptyset] &\geq \frac{1}{L}\sum_{i=i_0}^{d-1-a} e^{-8\delta k_i} \\ &\geq \exp\left(-\frac{8\delta}{L}\sum_{i=i_0}^{d-1-a} k_i\right) \\ &\geq \exp\left(-\frac{8\delta k}{L}\right),\end{aligned}$$

where the last inequality follows from Equation (4). □

The preceding lemma is stated for a set of tested edges. At first glance this statement appears to be significantly suboptimal, because the verifier may test many more edges than nodes. We show that, in fact, the number of edges in different classes cannot exceed the number of nodes.

To warm up, we consider a simple case.

**Lemma 3.** *Consider nonempty $W \subseteq V$. $E \cap (W \times W)$ contains edges from at most $|W| - 1$ distinct classes.*

*Proof.* Let $R(W) := \{\rho(w) : w \in W\}$ be the set of distinct ranks represented in $W$, written as $d$-bit strings, and let $\mathcal{T}(W)$ be their binary trie.

Each leaf in this trie corresponds to at least one element of $W$. Thus, the number of leaves is at most $|W|$, and the number of branching nodes (i.e., sibling pairs) is $|W|-1$. Each sibling pair in this trie corresponds to exactly one class that intersects with $W \times W$. □

Lemma 3 is useful when $\mathsf{VerifierSet}$ is small. But the verifier may spend time computing and may thus obtain a big set. The following lemma shows the computation for $s$ random oracle rounds can at most double the number of distinct classes in layers above $\log_2(s+1)$.

**Lemma 4.** *Suppose* VerifierSet *is obtained from* ProverSet *by $s$ or fewer sequential random oracle rounds, for $s \geq 1$. Then* Inc(VerifierSet) *has elements from at most* $2|\mathsf{ProverSet}|$ *distinct classes $E_{i,p}$ for $i < d - a$, where $a = \lceil \log_2 s \rceil$.*

*Proof.* Because the computation takes at most $s$ sequential rounds, for every $v \in \mathsf{VerifierSet}$, $\rho(v) \in [\alpha, \alpha + s]$, where $\alpha$ is the rank of some node in ProverSet, or $\rho(v) \in [0, s-1]$. Consider ranks truncated to length $d - a$. Because $s \leq 2^a$, each interval $[\alpha, \alpha + s]$ contains at most two distinct truncated ranks, and the interval $[0, s-1]$ contains just one rank $0^{d-a}$. Thus, the number of such distinct ranks in Inc(VerifierSet) is at most $2|\mathsf{ProverSet}| + 1$. We can build a trie on these truncated ranks the same way we build a trie of full ranks in Lemma 3. Each $E_{i,p}$ for $i < d - a$ that has edges in Inc(VerifierSet) corresponds to a node with two children (equivalently, a sibling pair) in this trie, and the number of such nodes is at most $2 \cdot |\mathsf{ProverSet}|$ because the number of leaves is at most $2 \cdot |\mathsf{ProverSet}| + 1$. □

Combining the above lemmas yields a lower bound on the total number of labels that are either supplied to or computed by the verifier. Unlike an edge count, this measure does not depend on the indegree of the graph.

**Theorem 1** (Lower bound for graph-based PoSWs)**.** *Let $\Pi$ be a graph-based PoSW whose underlying DAG has depth $n = 2^d$, and suppose that, for every challenge $\gamma$, $|\mathsf{ProverSet}(\gamma)| \leq m$ and the verifier performs no more than $s$ rounds of random oracle queries for some $s \geq 1$ to compute* VerifierSet$(\gamma)$.

*Let $0 < \delta \leq 1/8$, and define*

$$L := d - \left\lceil \log_2\big(\max\{8m, 1/\delta\}\big) \right\rceil - \lceil \log_2 s \rceil \,.$$

*If $\Pi$ has $2^{-\lambda}$ soundness against $\delta$-cheating, then*

$$m \geq \frac{\ln 2}{16} \frac{\lambda L}{\delta}. \tag{6}$$

*Proof.* Suppose (6) does not hold. Plug in the parameters of Lemma 4 into Lemma 2 to obtain a cut and apply Lemma 1 to obtain a contradiction to the soundness assumption. □

**Corollary 1.** *Let $m$ denote the size of* ProverSet *and $s$ denote the number of random oracle query rounds needed by the verifier. Let $n$ denote the depth of the graph $G$ in a graph-based PoSW. Suppose the PoSW is $2^{-\lambda}$ sound against $\delta$-cheating. Then Theorem 1 implies*

$$m = \Omega\left(\frac{\lambda}{\delta} \log \frac{n}{s \cdot \max\{m, 1/\delta\}}\right).$$

*In particular, fixing an arbitrary $\epsilon > 0$ and assuming $ms = O(n^{1-\epsilon})$ and $s/\delta = O(n^{1-\epsilon})$, we have*

$$m = \Omega\left(\frac{\lambda \log n}{\delta}\right).$$

# 4 The Diamond PoSW

In this section we construct a graph-based PoSW that matches the lower bound of Section 3 up to constant factors, while every vertex of the underlying DAG has indegree at most two. The graph is illustrated in Figure 2; the shape of the picture motivates the name *Diamond*.

## 4.1 The Diamond graph

The Diamond graph can be viewed as two complete binary trees built on the same set of leaves, one above and one below the leaves. The edges in both trees point upwards, and thus the root of the lower tree is the source, while the root of the upper tree is the sink. In addition to the regular tree edges, every left child in the upper tree has an edge to the corresponding right child in the lower tree.

Recursively, the Diamond graph is defined as follows:

**Definition 2** (The Diamond graph)**.** $\mathsf{Diamond}_0$ consists of a single vertex. To build $\mathsf{Diamond}_d$ for $d > 0$, take two copies of $\mathsf{Diamond}_{d-1}$ with sources $s_0, s_1$ and sinks $t_0, t_1$, respectively, and add a new source $s$ and a new sink $t$ and five new edges

$$(s, s_0), \qquad (s, s_1), \qquad (t_0, t), \qquad (t_1, t), \qquad (t_0, s_1)\,.$$

It is convenient to name the vertices and edges of $\mathsf{Diamond}_d$ explicitly (the equivalence of this explicit definition to Definition 2 follows easily by induction). In both trees we use the usual binary tree numbering, with vertices indexed by strings in $\{0,1\}^{\le d}$ (where $\varepsilon$, which denotes the empty string, is the index of the root). To distinguish the upper tree from the lower tree, given a binary string $x$, we will denote by $\overline{x}$ a node in the upper tree and by $\underline{x}$ a node in the lower tree; a leaf (when $|x| = d$) can be denoted by either $\underline{x}$ or $\overline{x}$ or just by $x$. Formally,

$$V_d = \{\underline{x} \mid x \in \{0,1\}^{<d}\} \cup \{x \mid x \in \{0,1\}^{d}\} \cup \{\overline{x} \mid x \in \{0,1\}^{<d}\}\,,$$

and $\underline{x}$ and $\overline{x}$ when $x \in \{0,1\}^d$ are just aliases for $x$. Then the edge set $E_d$ consists of the following for every $x \in \{0,1\}^{<d}$:

$$\begin{aligned} (\overline{x0}, \overline{x}) \text{ and } (\overline{x1}, \overline{x}) &\qquad \text{(upper-tree edges)},\\ (\underline{x}, \underline{x0}) \text{ and } (\underline{x}, \underline{x1}) &\qquad \text{(lower-tree edges)},\\ (\overline{x0}, \underline{x1}) &\qquad \text{(cross edges)}. \end{aligned}$$

**Lemma 5.** *Let $N = 2^d$. Then $|V_d| = 3N - 2$ and $|E_d| = 5N - 5$; the maximum indegree and outdegree are at most 2; the graph is acyclic and there is a unique topological ordering that gives a Hamiltonian path from the source $\underline{\varepsilon}$ to the sink $\overline{\varepsilon}$.*

*Proof.* The sizes, degrees, and acyclicity are immediate from the above description of $V_d$ and $E_d$ (or by induction from the recursive definition). The topological sort (whose correctness and uniqueness are immediate because it is a path and the graph is acyclic) is as follows: for every $x \in \{0,1\}^{<d}$

- the successor of $\underline{x}$ is $\underline{x0}$;
- the successor of $\overline{x0}$ is $\underline{x1}$;
- the successor of $\overline{x1}$ is $\overline{x}$.

□

## 4.2 The PoSW

We now instantiate the graph-based PoSW with $G = \mathsf{Diamond}_d$. Given a seed $\chi$, the prover computes the graph labeling and sends to the verifier the label $\phi$ of the sink. The prover and the verifier then perform $t$ instances of the following verification protocol.

- The verifier selects a uniformly random leaf $x = x_1 \dots x_d \in \{0,1\}^d$.
- The prover sends the usual Merkle authenticating path for $x$ in the upper tree: namely, labels of siblings
$$\overline{x_1 \dots x_{k-1}(1-x_k)}$$
of nodes on the path from $x$ to $\overline{\varepsilon}$ for $k < d$ (the labels of $x$ and its sibling are not needed). Call this set of siblings $\mathsf{ProverSet}$ and call the labels sent by the verifier *purpoted*.
- The verifier computes, in topological order, *purported* labels the following set $\mathsf{VerifierSet}$ of nodes: the nodes on the path from $\underline{\varepsilon}$ through $x$ and the sibling of $x$ to $\overline{\varepsilon}$ . That is,
$$\underline{x_1 \dots x_k} \text{ and } \overline{x_1 \dots x_k} \text{ for every } 0 \le k < d, \text{ and } x \text{ and } x_1 \dots x_{d-1}(1-x_d)\,.$$
(Observe, by inspection of the binary expansion of the predecessors of all three edge types defined in Section 4.1) the verifier is able to perform this computation, because for every node in $\mathsf{VerifierSet}$, its predecessors are either in $\mathsf{VerifierSet}$ or in the prover's response $\mathsf{ProverSet}$.) The verifier accepts if and only if the purported label of $\overline{\varepsilon}$ is equal to $\phi$.

Note that for every leaf $x$, the protocol is exactly the same whether the verifier chooses $x$ or its sibling.

## 4.3 Security

The security proof has three parts. First, we have to handle the problem of contradictory labels potentially used by the prover on different outgoing edges of the same node. Fortunately, the commitment $\phi$ fixes all the labels for the upper tree, by the usual Merkle tree commitment argument. We call such labels *canonical* and use them to also define canonical lower-tree values that are implied by the lower-tree computation starting from $\underline{\varepsilon}$, with canonical inputs from the upper tree. Second, we call nodes consistent if they are computed

via these canonical values, show that leaves that are inconsistent or have inconsistent tree ancestors are rejected, and argue that long consistent paths must be computed sequentially. Third, we use a simple inductive argument on the graph to show that if enough leaves pass verification, there must be a long consistent path. Before we do any of this, we make sure the random oracle behaves nicely, via a well-known argument.

**Ruling Out RO Accidents** First, we consider the following bad event $\mathsf{Bad}_{H_\chi}$: among all the well-formed queries made by the prover and the verifier in the protocol,

1. there is a collision (i.e., there are two distinct inputs to $H_\chi$ that produce the same output), or
2. there is a lucky guess of $H_\chi$ (i.e., there is an edge $(u, v)$ such that the part of the input to $H_\chi(v, \cdot)$ corresponding to $u$ only later appears as an output of $H_\chi(u, \cdot)$), or
3. the claimed PoSW output $\phi$ produced by the prover appears as an output of $H_\chi$ only after the prover is done computing the PoSW.

**Lemma 6.** $\Pr[\mathsf{Bad}_{H_\chi}] \leq \frac{3q^2}{2^{w+1}}$.

*Proof.* There are $q$ total queries, and well-formed ones have at most two predecessor inputs each. Consider only well-formed ones and remove duplicates. Number the queries consecutively in the order they are asked; for queries made in parallel, resolve ties in a way that is the opposite of the topological order of their corresponding nodes. Then the bad event happens whenever an output of some hash query matches the output of a lower-numbered hash query, or an input to a lower-numbered hash query, or value $\phi$ output by the prover. Thus query number $i$ has at most $3(i-1)+1$ ways out of $2^w$ total to cause such an event to happen ($i-1$ collision possibilities, $2(i-1)$ possibilities to match an already existing input for a successor node, and 1 possibility to match $\phi$). Now apply the union bound over all $q$ queries. □

**Canonical Labels** The challenge in formalizing the proof is that for nodes whose outdegree is greater than 1, a malicious prover may use different labels along different edges. To help us reason about these different values, it helps to define a "canonical" label, denoted $\kappa_i$, for every node $i$. We define it so that every node has exactly one canonical label, which is well-defined by the end of the compute phase.

**Definition 3** (Upper-Tree and Leaf Canonical Labels)**.** Call $\phi$ the *canonical* label of the root, $\kappa_\epsilon$. For every upper-tree node $\overline{x} \in \{0,1\}^{<d}$ with a canonical label $\kappa_{\overline{x}}$, consider the first well-formed RO query for the node $\overline{x}$ that produced $\kappa_{\overline{x}}$ as output (if any such well-formed query for $\overline{x}$ was made by the prover during the compute phase). That query had, as inputs, some claimed labels for the predecessors $\overline{x0}$ and $\overline{x1}$ of this node. Define the *canonical* labels $\kappa_{\overline{x0}}$ and $\kappa_{\overline{x1}}$ of nodes $\overline{x0}$ and $\overline{x1}$ to be these inputs. This process recursively defines canonical labels for some nodes; once this process completes, define the canonical label of any node $\overline{x} \in \{0,1\}^{\leq d}$ that does not receive a canonical label through this process to be $\perp$ (we assume $\perp$ is not a possible RO output or input, and we consider only well-formed RO queries).

For lower-tree nodes, we define canonical labels differently: based on hypothetical, rather than actual, computation.

**Definition 4** (Lower-Tree Canonical Labels)**.** The canonical label $\kappa_{\underline{\varepsilon}}$ of $\underline{\varepsilon}$ is defined to be its correct label $\ell_{\underline{\varepsilon}}$ based on the verifier salt $\chi$. The canonical label $\kappa_{\underline{x}}$ for $x \in \{0,1\}^{<d}$ is defined as $\perp$ if any predecessor of $\underline{x}$ has $\perp$ as its canonical label, and otherwise as the label that would be computed if the canonical labels of its predecessors were to be used as inputs to $H_\chi$ during labeling. That is, for any $x \in \{0,1\}^{<d}$,

$$\kappa_{\underline{x}} = H_\chi(\underline{x}, (\kappa_u)_{u \in \mathsf{pred}(\underline{x})}) \,.$$

Note that this definition does not depend on whether such queries were actually made.

**Lemma 7.** *If verification passes and $\neg\mathsf{Bad}_{H_\chi}$, then the purported labels sent by the prover and computed by the verifier are all canonical.*

*Proof.* Suppose not. First we handle upper-tree nodes and leaves. Let $\overline{x}$ be a highest (closest to the root $\overline{\varepsilon}$) node whose purported value is not equal to its canonical value. We know $\overline{x}$ is not the root, because verification passed, and the verifier checks that the purported and canonical labels of the root are equal. Therefore, $\overline{x}$ has a parent $\overline{y}$ in the upper tree, and the verifier made the following query to $H_\chi$: it gave, as one of the inputs to $H_\chi$, the purported label of $\overline{x}$, and received the canonical label $\kappa_{\overline{y}}$. Because $\neg\mathsf{Bad}_{H_\chi}$, there are no sequentiality violations, and since $\kappa_{\overline{y}}$ is canonical and thus was used as an input to $H_\chi$ by the prover during the compute phase (or returned as $\phi$), it must have been obtained by the prover as an output of $H_\chi$ during the compute phase, as well. When it was obtained as an output for the first time, the purported label of $\overline{x}$ must have been used as input (because $\neg\mathsf{Bad}_{H_\chi}$ implies there are no collisions), and thus this purported label is canonical by Definition 3. We have reached a contradiction.

Given this result for the upper-tree nodes, the statement for lower-tree nodes follows trivially by induction starting with the label of $\underline{\varepsilon}$ (which the verifier computes correctly, and thus canonically by definition) and going up (because all the inputs used by the verifier on the way up are canonical). □

**Consistent Nodes** Note that there is no guarantee that the prover uses the canonical label of a node for each of its outgoing edges. This motivates the definition of a *consistent* node as one whose inputs were properly supplied.

**Definition 5.** A node $v$ is *consistent* if the following two conditions hold:

- $H_\chi$, when queried on the canonical labels of the predecessors of $v$, produces the canonical label $\kappa_v$ of $v$, i.e.,
$$\kappa_v = H_\chi(v, (\kappa_u)_{u \in \mathsf{pred}(v)}) \,.$$
(this condition is already true simply by definition of "canonical" for all nodes but the leaves, as long as their and their predecessors' canonical labels are not $\perp$), and

- such a query was made by the prover during the compute phase (this condition is not part of the definition of "canonical" for any node).

Note that Definition 3 and Definition 4 find canonical labels via *actual* random oracle *inputs* for the leaves and upper tree, and via *hypothetical* random oracle *outputs* for the lower tree. In particular, the canonical value of a leaf might never appear as a random oracle output—but if verification passes for this leaf, then it must, because the purported leaf label is produced as a random oracle output, and must be equal to the canonical leaf label by the commitment property of the Merkle tree. This fact is shown in the following lemma, which relates inputs and hypothetical outputs to actual outputs.

**Lemma 8.** *If verification passes and* $\neg\mathsf{Bad}_{H_\chi}$*, then every node whose label is computed by the verifier is consistent.*

*Proof.* By Lemma 7, the verifier uses only canonical inputs and produces only canonical outputs, so the first condition of Definition 5 is satisfied. It remains to show that queries to $H_\chi$ that are made by the verifier were first made by the prover during the compute phase. We do so by a simple induction argument, starting $\overline{\varepsilon}$ and going down. Indeed, suppose not—that is, there is some node $u$ such that the query $H_\chi(u, (\kappa_w)_{w\in\mathsf{pred}(u)})$ is made by the prover after the compute phase or by the verifier. Among all such nodes, pick $u$ that is closest to $\overline{\varepsilon}$. We know $\kappa_u$ is not an output of any other query to $H_\chi$, by the first condition of $\mathsf{Bad}_{H_\chi}$. The node $u$ cannot be $\overline{\varepsilon}$, because that would contradict the third condition of $\neg\mathsf{Bad}_{H_\chi}$ (since $\kappa_u = \phi$ would be seen for the first time as an $H_\chi$ output). So $\overline{\varepsilon}$ is consistent. Then, by induction, because the verifier computes a path from $u$ to $\overline{\varepsilon}$, there must be an edge $(u, v)$, and $v$ is consistent because it is closer than $u$ to $\overline{\varepsilon}$. Because $v$ is consistent, there is an $H_\chi$ query during the compute phase that has $\kappa_u$ as one of its inputs and $\kappa_v$ as its output, which contradicts the second condition of $\neg\mathsf{Bad}_{H_\chi}$. □

**Lemma 9.** *Let* $p$ *be the length of the longest path of consistent nodes that ends in* $\overline{\varepsilon}$*. If* $\neg\mathsf{Bad}_{H_\chi}$*, then the prover must have taken sequential time at least* $p$ *to compute its PoSW output* $\phi$*.*

*Proof.* Indeed, consider an edge $(u, v)$ such that $u$ and $v$ are both consistent. Then each of the canonical labels $\kappa_u$ and $\kappa_v$ was produced as the output of an $H_\chi$ query made during the compute phase (by Definition 5). Because $\kappa_u$ is an input to the query that produced $\kappa_v$, and $\neg\mathsf{Bad}_{H_\chi}$ states that there are no lucky guesses, the query that produced $\kappa_u$ came before the first query that produced $\kappa_v$. Since there are $p - 1$ such edges on the path, and the end of the path is output by the end of the compute phase, the compute phase must take time at least $p$. □

**The Existence of a Long Path** It remains to show that there is a long path of consistent nodes. First, we make a purely combinatorial observation about our graph $\mathsf{Diamond}_d$.

**Lemma 10.** *Choose any* $B < 2^d$ *leaves in* $\mathsf{Diamond}_d$ *and call them* bad*. Call the rest* good*. Call a non-leaf node* good *if it has at least one good leaf in its subtree. Then there is a path*

*that contains all the good nodes and no other nodes, starts at $\underline{\varepsilon}$, ends at $\overline{\varepsilon}$, and has length at least*

$$\begin{cases} 3 \cdot 2^d - 2 \text{ if } B = 0 \\ 3 \cdot (2^d - B) \text{ otherwise} \end{cases},$$

*i.e.,* $\min(3 \cdot (2^d - B), |V_d|)$.

*Proof.* This lemma is a simple induction on $d$, constructing the obvious paths with careful counting. Let $T_d = |V_d| = 3 \cdot 2^d - 2$. The base case of a single node (which is either good or bad) for $d = 0$ is trivial. For the inductive case, consider the left and the right halves of the graph $\mathsf{Diamond}_d$; let $G_0$ and $G_1$ be the number of good leaves in each, with $2^d - B = G_0 + G_1$.

If $G_0 < 2^{d-1}$ and $G_1 < 2^{d-1}$ (each half has some good leaves), create a path starting at $\underline{\varepsilon}$, going to $\underline{0}$, to the path given by the inductive hypothesis for the left half that ends in $\overline{0}$, to $\underline{1}$, to the path given by the inductive hypothesis for the right half that ends in $\overline{1}$, to $\overline{\varepsilon}$. If $G_0 = 2^{d-1}$, skip the left half, and go from $\underline{\varepsilon}$ directly to $\underline{1}$; if $G_1 = 2^{d-1}$, skip the right half, and from $\overline{0}$ directly to $\varepsilon$. The total number of nodes on the path is $2 + \min(3 \cdot G_0, T_{d-1}) + \min(3 \cdot G_1, T_{d-1}) \geq \min(3 \cdot (2^d - B), T_d)$. □

We now put it all together.

**Theorem 2** (Soundness of the Diamond PoSW)**.** *Let $T = 3 \cdot 2^d - 2$ be the number of nodes in $\mathsf{Diamond}_d$. Suppose a malicious prover sends its commitment after at most $(1 - \delta)T$ sequential random-oracle rounds. Then acceptance probability in the $\mathsf{Diamond}_d$ PoSW with verification repeated $t$ times is at most*

$$(1 - \delta)^t + \frac{3q^2}{2^{w+1}} \leq \exp(-\delta t) + \frac{3q^2}{2^{w+1}}. \tag{7}$$

*Proof.* Lemma 6 bounds the second term in (7). It remains to bound the probability that $\neg\mathsf{Bad}_{H_\chi}$ and the verifier accepts. Define a leaf to be *bad* if it, its sibling, or any of its binary-tree ancestors in either of the trees, is inconsistent; let $B$ be the number of bad leaves. Define *good* as in Lemma 10. Note that whether nodes are good or consistent is determined by the end of the compute phase. Note also that every good node is consistent. Thus, by Lemma 10, there is a path of length $3 \cdot (2^d - B)$ (if $B > 0$) or $T$ (if $B = 0$), which contains only consistent nodes and ends at $\overline{\varepsilon}$. By Lemma 9, such a path takes as many sequential steps to compute as its number of nodes. Therefore $3 \cdot (2^d - B) \leq (1 - \delta)T$ (if $B > 0$) or $T \leq (1 - \delta)T$ (if $B = 0$), and thus $B/2^d \geq \delta$. If a bad leaf is queried and $\neg\mathsf{Bad}_{H_\chi}$, verification will fail by Lemma 8, so the probability verification passes and $\neg\mathsf{Bad}_{H_\chi}$ is at most $(1 - B/2^d)^t \leq (1 - \delta)^t$. □

**Corollary 2.** *To ensure the soundness error does not exceed $2^{-\lambda} + \frac{3q^2}{2^{w+1}}$ against a prover who tries to save $\delta$ fraction of the work, it suffices to take $t = \lceil \lambda \ln 2/\delta \rceil$; thus, the length of the proof $\pi$ is most $(d - 1)\lceil \lambda(\ln 2) \rceil/\delta$ hash outputs (of $w$ bits each), and less if we take care to merge overlapping authenticating paths (not sending labels for the same node twice, and not sending labels for ancestors of queried leaves, which are anyway computed by the verifier).*

## Acknowledgments

Much of this work was done while the authors were visiting the Simons Institute for the Theory of Computing during the Summer 2025 program "Cryptography 10 Years Later: Obfuscation, Proof Systems, and Secure Computation".

## References

[AAK+25] Hamza Abusalah, Karen Azari, Chethan Kamath, Erkan Tairi, and Maximilian von Consbruch. Impossibility of VDFs in the ROM: The complete picture. Cryptology ePrint Archive, Report 2025/1773, 2025.

[AKK+19] Hamza Abusalah, Chethan Kamath, Karen Klein, Krzysztof Pietrzak, and Michael Walter. Reversible proofs of sequential work. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 277–291. Springer, Cham, May 2019.

[Bac02] Adam Back. Hashcash: A denial of service counter-measure. Technical report, Hashcash.org, August 2002. Technical report.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Cham, August 2018.

[CP18] Bram Cohen and Krzysztof Pietrzak. Simple proofs of sequential work. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 451–467. Springer, Cham, April / May 2018.

[CP19] Bram Cohen and Krzysztof Pietrzak. The chia network blockchain: Proof of space and time. `https://chia-network.github.io/greenpaper/`, 2019. Chia Green Paper.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 139–147. Springer, Berlin, Heidelberg, August 1993.

[GRY25] Ziyi Guan, Artur Riazanov, and Weiqiang Yuan. Breaking verifiable delay functions in the random oracle model. In Yael Tauman Kalai and Seny F. Kamara, editors, *CRYPTO 2025, Part VII*, volume 16006 of *LNCS*, pages 161–191. Springer, Cham, August 2025.

[GS98] David M. Goldschlag and Stuart G. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In Rafael Hirschfeld, editor, *Financial Cryptography, Second International Conference, FC'98, Anguilla, British West Indies, February 23-25, 1998, Proceedings*, volume 1465 of *Lecture Notes in Computer Science*, pages 214–226. Springer, 1998.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Berlin, Heidelberg, December 2010.

[MMV13] Mohammad Mahmoody, Tal Moran, and Salil P. Vadhan. Publicly verifiable proofs of sequential work. In Robert D. Kleinberg, editor, *ITCS 2013*, pages 373–388. ACM, January 2013.

[MSW20] Mohammad Mahmoody, Caleb Smith, and David J. Wu. Can verifiable delay functions be based on random oracles? In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *ICALP 2020*, volume 168 of *LIPIcs*, pages 83:1–83:17. Schloss Dagstuhl, July 2020.

[Rab83] Michael O. Rabin. Transaction protection by beacons. *J. Comput. Syst. Sci.*, 27(2):256–267, 1983.

[RSW96] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, USA, 1996.

[Val77] Leslie G. Valiant. Graph-theoretic arguments in low-level complexity. In *Mathematical foundations of computer science (Proc. Sixth Sympos., Tatranska Lomnica, 1977)*, volume 53 of *Lecture Notes in Computer Science*, pages 162–176, Berlin, 1977. Springer.

# A Statement on AI usage

Generative AI (ChatGPT) was used to convert proof outlines written by us into fullerfledged detailed proofs. These proofs were then fully edited and largely rewritten, though some calculations and presentation ideas produced by ChatGPT remain. Generative AI was also used to help generate figures and to proofread the paper.