An Incremental PoSW for General Weight Distributions

Hamza Abusalah[1] and Valerio Cini[2]

[1] IMDEA Software Institute, Madrid, Spain. hamza.abusalah@imdea.org
[2] Austrian Institute of Technology, Vienna, Austria. valerio.cini@ait.ac.at

Abstract. A proof of sequential work (PoSW) scheme allows the prover to convince a verifier that it computed a certain number of computational steps sequentially. Very recently, graph-labeling PoSW schemes, found applications in light-client blockchain protocols, most notably bootstrapping. A bootstrapping protocol allows a light client, with minimal information about the blockchain, to hold a commitment to its stable prefix. An incremental PoSW (iPoSW) scheme allows the prover to non-trivially increment proofs: given χ, π_1 and integers N_1, N_2 such that π_1 is a valid proof for N_1, it generates a valid proof π for $N_1 + N_2$.

In this work, we construct an iPoSW scheme based on the skiplist-based PoSW scheme of Abusalah et al. and prove its security in the random oracle model by employing the powerful on-the-fly sampling technique of Döttling et al. Moreover, unlike the iPoSW scheme of Döttling et al., ours is the first iPoSW scheme which is suitable for constructing incremental non-interactive arguments of chain knowledge (SNACK) schemes, which are at the heart of space and time efficient blockchain light-client protocols. In particular, our scheme works for general weight distributions, which we characterize as incrementally sampleable distributions. Our general treatment recovers the distribution underlying the scheme of Döttling et al. as well as the distribution underlying SNACK-enabled bootstrapping application as special cases. In realizing our general construction, we develop a new on-the-fly sampling technique.

1 Introduction

Proofs of Work (PoW) were introduced by Dwork and Naor [8], and in the past years have become very popular in the context of cryptocurrencies. A PoW scheme allows a prover to convince a verifier that a certain amount of computation was performed. However, this says nothing about whether the computation was done sequentially or in parallel.

A Proof of Sequential Work [12] (PoSW) is an (interactive) proof system in which the prover, on common inputs an integer parameter N and a statement χ, computes a proof that convinces the verifier that N *sequential* computational steps have been performed since χ was received.

A simple PoSW scheme based on a random oracle $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ is a hash chain: the prover computes $y_i := \tau(\chi, y_{i-1})$ for $i \in [N] := \{1, \dots, N\}$ and

$y_0 := \tau(\chi)$ and defines its proof $\pi := y_N$. The verifier verifies π by recomputation. From basic properties of random oracles, any accepting proof must have been computed, with overwhelming probability, in N sequential steps even by a massively parallel adversarial prover. From the prover's perspective, this scheme is optimal: the prover does exactly N sequential steps and keeps constant size (λ bits) memory during its computation. However, the verifier needs to recompute the proof, and hence, spend as much resources as the honest prover. Therefore, for the applicability of PoSW schemes, one also requires succinctness, beyond completeness and soundness. Succinctness requires that the proof size as well as the verifier's running time are poly-logarithmic in N.

Beyond classical applications of PoSW schemes for time stamping, where a prover wants prove to future verifiers that it stamped a certain message some time in the past, we have now more applications in the blockchain arena. The first such application uses a special form of a PoSW scheme which has *unique* proofs. Chia [6], in an effort to designing sustainable blockchains, combines in its mining process proofs of space [9, 1] and verifiable delay functions [4], which are a subclass of PoSW with unique proofs. While generating these proofs requires large space and sequential time resources, they must be efficiently and publicly verifiable.

Recently, Graph-Labeling PoSW (GL-PoSW) schemes found applications in light-client blockchain protocols, most notably *bootstrapping* [2]. A light-client, which has minimal information about the blockchain in question, say its genesis block, is said to have securely bootstrapped if it ends up, after potentially talking to multiple provers, holding a commitment to the honest stable-prefix of the blockchain. The light client must be efficient in the sense that its verification time is at most poly-logarithmic in the length of the blockchain.

In the above applications, different classes of PoSW schemes are assumed. When the sequential computation is used in mining as in Chia, the PoSW is required to have *unique* proofs. This subclass of PoSW schemes is called verifiable delay functions VDFs [4, 13, 14, 10]. The subclass used for bootstrapping application is called *graph-labeling* PoSW (GL-PoSW) schemes [12, 5, 7, 3, 2]. It is not known how to use VDFs to solve the bootstrapping problem, nor how to use graph-labeling PoSW in the mining application [6].

In this work, we focus on the class of GL-PoSW schemes, however, the same question that we address here in terms of incrementality is relevant for VDFs; in fact these are called *continuous* VDFs [10]. One main motivation of making this choice is that GL-PoSW schemes can be used to provide efficient solutions to blockchain light-client bootstrapping [2].

1.1 Graph-Labeling PoSW Schemes

A Graph-Labeling PoSW (GL-PoSW) scheme is a PoSW scheme for a weighted graph family $\varGamma = ((G_N, \varOmega_N))_{N\geq 0}$, where $G_N = ([N]_0, E_G)$ is a DAG on $[N]_0 := \{0, \ldots, N\}$ and $\varOmega_N : [N] \to [0,1]$ is a weigh function, i.e., $\sum_{i=1}^{N} \varOmega_N(i) = 1$. We refer to weight functions as probability distributions when convenient. All existing GL-PoSW schemes [12, 5, 7, 3, 2] are in the random oracle model. The prover, upon receiving a statement χ from the verifier, uses χ to refresh a random oracle $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ to compute a labeling $L : [N]_0 \to \{0,1\}^\lambda$ of G_N, where the label of $i \in [N]_0$ is simply defined as $L(i) := \tau(\chi, i, L(\mathsf{Parents}(i)))$, where $\mathsf{Parents}(i)$ denotes the (ordered) set of nodes with outgoing edges directed towards i. The prover then sends to the verifier a (vector) commitment ϕ to the labeling L. The verifier then engages with the prover in a challenge response protocol in which for each challenge $i \in [N]$ drawn by the verifier according to the distribution induced by Ω_N, the prover responds with protocol- and challenge-specific labels and openings from L, which the verifier then checks for consistency. (A part of the prover's response to challenge i is a ϕ-opening at position i.)

Completeness, (α, ϵ)-soundness, and succinctness are required. Completeness stipulates that the verifier always accepts honest proofs, and succinctness requires that for any honestly generated proof π, it holds that $|\pi| \in \mathsf{poly}(\lambda, \log N)$, and the running time of the verifier is upper-bonded by $\mathsf{poly}(\lambda, \log N)$, where λ is a security parameter. Now (α, ϵ)-soundness guarantees that except with probability ϵ, the verifier rejects any proof generated by a prover which made less that αN *sequential* queries to $\tau(\cdot)$.

Such an interactive protocol is then made non-interactive in the ROM by applying the Fiat-Shamir [11] transform. The GL-PoSW scheme of [7] is defined and constructed non-interactively, but otherwise it follows the above design template.

The first PoSW scheme [12] is a GL-PoSW whose underlying graph G_N is *depth robust*, has the disadvantage of large space requirement $\approx \lambda N$ bits during the prover's computation.

The high space requirement was addressed in [5], where they design a simple tree-based DAG G_N, which when used in the above design template, allows for a practically efficient GL-PoSW, where the prover requires much less space $\ll \lambda N$ bits. Moreover, they show general space-time tradeoffs of the prover strategy, that we will discuss shortly.

In [3], another practically efficient GL-PoSW scheme that uses a skiplist DAG as its underlying graph is proposed. The scheme uses a slightly different, but essentially the same labeling strategy: instead of defining $L(i) := \tau(\chi, i, L(\mathsf{Parents}(i)))$, it is computed using a permutation whose input is selected among $L(\mathsf{Parents}(i))$. The permutation-based labeling gives the PoSW the additional feature of *reversibility*: given $L(i)$ of any $i \in [N]$, one can efficiently compute $L(i-1)$. Reversibility comes at the cost of having labels of larger size: for parameter N, it holds that $|L(i)| = \lambda \cdot \log N$.

Motivated by applications of GL-PoSW schemes in designing light-client blockchain applications, a variant of the skiplist PoSW scheme is proposed in [2]. This scheme is identical to the original skiplist construction, except that labels are defined as $L(i) := \tau(\chi, i, L(\mathsf{Parents}(i)))$ for a random oracle τ. The resulting scheme is no longer reversible as in [3] . However, it has the advantage of smaller labels $|L(i)| = \lambda$. Having constant size labels is important in the design of blockchains which allow for light-client protocols, where in the ith block of the respective blockchain, a GL-PoSW label $L(i)$ is stored, and having a constant size label for an ever-growing blockchain saves a lot of otherwise wasted storage. (For more details on such augmented blockchains, see [2].)

For simplicity, we refer to the scheme from [5] as the tree construction TC, and both schemes from [3, 2] as the skiplist construction SC.

Space-Efficient Provers. In all these schemes, a prover sequentially computing the labeling L of the PoSW underlying graph G_N, which would take N sequential invocations to its random oracle τ, and storing all labels L, would be able to compile a convincing proof π out of L. However, if the prover does actually spend N sequential computations in computing L but stored nothing beyond $L(N)$, then it may need to spend another N sequential queries to be able to compile a convincing proof π. This issue can be seen as either a space-efficiency or soundness slack issue: while honest provers with essentially no storage need to spend $2N$ steps, soundness is quantified over malicious provers doing $< \alpha N$ sequential computation for $\alpha \in (0, 1]$.

This space issue was raised in both [12] and [5], where in the former it is left open, and in the latter, a general space-time tradeoff of the prover's strategy is given. Roughly speaking, for space $\approx \sqrt{N}$ an honest prover can spend $\approx N + \sqrt{N}$ and convince the verifier that it did N sequential steps.

1.2 Incremental PoSW

This gap between the number of sequential steps of the honest and malicious provers, was essentially closed by Döttling, Lai and Malavolta [7], who give a variant of the TC scheme, call it ITC, in which the prover uses space up to $\mathsf{poly}(\lambda, \log N)$ and spends N sequential steps to convince the verifier that it did N sequential steps. In doing so, they introduce the notion of an *incremental* PoSW. This is a *non-interactive* GL-PoSW scheme $(\mathsf{P}, \mathsf{V}, \mathsf{Inc})$, in which, in addition to the prover/verifier algorithms, there is an additional Inc algorithm that given χ, π_1 and integers N_1, N_2 such that π_1 is a valid proof for N_1, it generates a valid proof π for $N_1 + N_2$. To avoid trivialities, one requires that π is asymptotically succinct, even if the incrementation is applied arbitrarily many times, and that the running time of Inc is essentially independent[3] of N_1. This condition avoids the trivial solution, in which either Inc computes a proof π_2 from π_1 onwards and defines $\pi := (\pi_1, \pi_2)$, or simply ignores π_1 and computes π from scratch for $N_1 + N_2$.

The starting point for ITC is TC. Recall that TC is an interactive GL-PoSW whose underlying weighted DAG is $(G_N = ([N]_0, E_G), \Omega_M)$, where G_N is a tree-like DAG with a single sink N and Ω_M is such that $\Omega_M(i) := 1/M$ for every $i \in [M]$ leaf node among $[N]$. The prover P computes, using a random oracle $\tau(\cdot)$, the labeling L of G_N and sends $\phi_L := L(N)$ to V, which in return selects t challenges $i_1, \ldots, i_t$ according the distribution induced by Ω_M, which is in this case the uniform distribution over $[M]$, and sends them to P, which responds by

[3] By essentially independent we mean that the dependency is at most poly-logarithmic.

openings to these challenges. Finally V accepts if and only if the openings are consistent with ϕ_L, which serves as a commitment to L.

This construction is then made non-interactive using the Fiat-Shamir transform: P computes challenges by using enough random bits from applying the random oracle $\tau(\chi, \phi_L, \cdot)$ appropriately many times, say on inputs $1, 2, \ldots$, and then uses these random bits to deterministically generate t challenges. However, inherent to this approach, is that P only learns its challenges at the end of the computation, and by then, it either must have stored all L and can then open the challenges efficiently, or it recomputes the missing labels among L that are needed to answer the challenges. In between these two extremes, there are general space-time tradeoffs that the prover can employ as we discussed above.

ITC is an alternative approach towards making TC non-interactive and incremental at the same time. ITC employs the clever *on-the-fly sampling* technique: as P is labeling G_N, in topological order, it learns some *potential* challenges from its already computed labels, and it learns with certainty what labels it already computed will not be part of its final challenges. This allows P to discard the labels of such useless nodes, and hence keep only the essential labels in memory. By the time P computes $L(N)$, it learns its final t challenges, for which it already stored their respective openings.

1.3 Incremental PoSWs for Incremental SNACKs

On light-client blockchain protocols. Recently, [2] show how GL-PoSW schemes can be generically used to augment blockchains such that they would then allow for efficient light-client secure bootstrapping. In their terminology, a full miner holding the entirety of a PoSW-augmented blockchain provides the light client with a *succinct non-interactive argument of chain knowledge* (SNACK) proof. The SNACK proof, in addition to some blockchain suffix blocks, allows the light client to hold a commitment to a stable prefix of the blockchain. For this application, a standalone GL-PoSW suffices: in the SNACK construction of [2], any GL-PoSW is used generically to augment a blockchain such that the augmented blockchain becomes SNACK-friendly, i.e., one can generate SNACK proofs for the augmented blockchain efficiently.

However, if the PoSW is incremental in the construction of [2], then the full node miner, having produced or obtained a *valid* SNACK proof for an augmented blockchain of length N, will be able to increment its SNACK proof for the blockchain when it grows into any length $N' > N$, and hence allows the full node to bootstrap light clients without storing the first N blocks of the N'-long blockchain. (The genesis block is always assumed to be stored by all parties.) This allows full nodes to become space-efficient, which is a great advantage given the massive sizes of the ever-growing blockchains.

The ITC scheme and light-client blockchain protocols. In [2], it is shown how to build a SNACK system from any GL-PoSW scheme. Furthermore, if the underlying GL-PoSW is incremental, then so is the overlying SNACK. The standalone (non-incremental) SNACK can be used to provide efficient solutions to the blockchain light-client bootstrapping problem. Additionally, if the SNACK is incremental, it is suggested in [2] that such an incremental SNACK would make the prover of these light-client protocols even more efficient; the prover becomes space-efficient.

SNACKs are defined with respect to a family of weighted DAG $(G_N = ([N]_0, E_G), \Omega_N)_{N \geq 0}$. The underlying weighted DAG of the SNACK is inherited from the underlying GL-PoSW. For the SNACK application of light-client bootstrapping, $\Omega_N : [N] \to [0,1]$ doesn't induce the uniform distribution over $[N]$. Roughly speaking, $\Omega_N(i) \sim 1/(N-i)$, which is far from uniform over $[N]$. This motivates designing incremental PoSW schemes with general weight distributions.

On defining incremental PoSWs. For the light-client blockchain application, an honest prover that increments *any valid SNACK proof* should make the verifier accept, regardless of how the SNACK proof it incremented was generated. This means that we need the same guarantees from the underlying iPoSW: *any valid PoSW proof* that is honestly incremented will make the verifier accept. This is also a natural requirement on incrementality, when the iPoSW is used in a distributed fashion: there are multiple parties that compute and increment; one party may increment another's computation and if the incrementation was done honestly on a valid proof, then the resulting proof must also verify regardless of how the proof of the previous party was generated.

1.4 Our Contributions

Motivated by the recent connection between light-client blockchain applications and GL-PoSW schemes [2], we advance the current understanding by

- strengthening the definition of iPoSW of [7] such that it becomes useful for light-client blockchain applications. The iPoSW definition of [7] only requires that honestly incrementing an honestly generated proof, rather than *any valid proof*, makes the verifier accept. For the usability of iPoSW in distributed applications, like incremental SNACKs and blockchains, we strengthen their definition as highlighted above. We also observe that their construction, ITC, achieves our stronger definition.
- constructing an iPoSW scheme whose underlying weighted graph family $(G_N, \Omega_N)_{N \geq 0}$ is such that G_N is the simple skiplist $G_N = ([N]_0), E_N)$ and whose distribution Ω_N is any arbitrary *t-incrementally sampleable* distribution. In particular, both the uniform and the SNACK distributions are t-incrementally sampleable distribution. Therefore, our iPoSW is the first and only iPoSW that can be used to construct an incremental SNACK, which in turn, can be used to construct the first *space-efficient* prover in blockchain light-client bootstrapping [2]. Along the way, we give a simple characterization of t-incrementally sampleable distributions. Technically, we also devise a new on-the-fly-sampling technique that works for all such distributions.

In a bit more detail. We first give a standalone iPoSW scheme based on the skiplist graph (Sect. 5). The scheme uses the same distribution Ω_N as in [7] and can be thought of as a general-purpose iPoSW. To prove security, we use and adapt the same on-the-fly-sampling strategy of [7]. Informally speaking, and on a very high level, the on-the-fly sampling works by randomly and without replacement sampling a set S of t elements from two sets S_0 and S_1, each contains t elements and a certain fraction of which is inconsistent. One then uses a Hoeffding bound to reason about the fraction of inconsistent elements in S in relation to the corresponding fractions of the original sets S_0, S_1. Fortunately, the same Hoeffding bound can be used when sampling randomly *with or without* replacement.

When moving to general probability distributions, one can still apply a general Hoeffding bound when sampling *with replacement* from a general probability distribution, i.e., when the random variables of the samples are independent. However, when sampling is done without replacement, then we are not aware of appropriate Hoeffding-like bounds that one can apply generically.

Therefore, we devise a new sampling strategy. Our sampling follows the Poisson binomial distribution, and informally, given S_0 and S_1 whose elements are sampled from Ω_t, we sample S such that each $s_i \in S_0 \cup S_1$ is added to S with probability p_i that is proportional to $\Omega_{2t}(s_i)$. Instead of having $|S|$ exactly t, we have that $|S|$ is on expectation t. We show that this sampling strategy works for all t-incrementally sampleable distributions.

In Sect. 6, we use this new on-the-fly sampling technique to construct an iPoSW which works for any t-incrementally sampleable distribution. When applying the new technique, new challenges arise: the verifier no longer can verify the consistency of the on-the-fly sampled elements. We solve this problem by making the prover commit to, and give away, as part of its proof, extra sets that allow the verifier to check the consistency of the sampled challenges. This change increases the proof size slightly.

1.5 A High-Level Technical Overview.

In this section, we only give the high-level overview of our skiplist-based iPoSW (Sect. 5) when Ω_N is the uniform distribution and we sample without replacement. The general case when Ω_N is any t-incrementally sampleable distribution and our on-the-fly sampling follows the Poisson binomial distribution is given in Sect. 6.2.

Let's first review the interactive SC construction. Fig. 1 shows the skiplist graph $G_N = ([N], E_G)$ for $N = 16$. We only show how P, which has the labeling L of G_{16}, answers its challenges. For example, let $v \in [16]$ be a challenge, then P locates the unique shortest path from the source 0 to the sink 16 which passes through v. Then it answers with the labels on this path and the labels of the parents of the nodes on the path. V simply checks the consistency of the labels on the path. As the path is of length $O(\log N)$ and each node has $O(\log N)$ parents, the opening for a single challenge is of size $O(\lambda \log^2 N)$. In contrast, the opening for a path in TC is $O(\lambda \log N)$.

Our incremental (non-interactive) PoSW ISC is then obtained by employing the on-the-fly sampling technique of [7]. For simplicity of exposition, we show how the challenges are on-the-fly sampled, however, without showing the corresponding openings of these challenges. In Section 5, we give a more comprehensive overview of the construction, including how to incrementally re/combine *partial* openings to *full* openings of the final challenges.

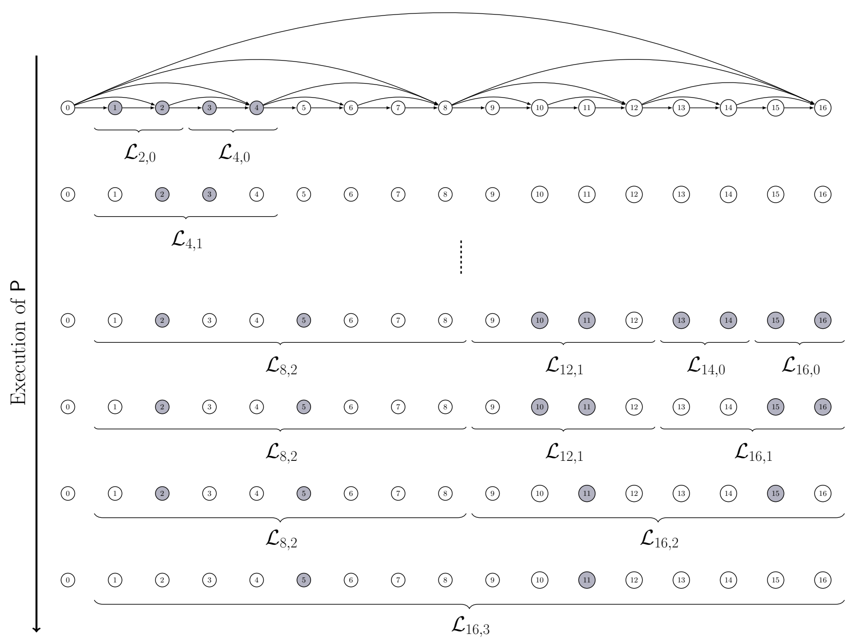

Fig. 1. Evolution of the stored challenges during the protocol. The elements of the list $\mathcal{L}_{v,i}$ are actually full openings of challenges. To ease readability, we have only drawn the corresponding challenge nodes. An example for G_{16}, $t = 2$.

The on-the-fly sampling is illustrated in Fig. 1 for an example graph G_{16} and $t = 2$. In general, the technique works as follow: let $t = 2^c$ for some integer c be the number of challenge nodes/openings to be produced by the end of the protocol. For every $v \in [N]$ that is a multiple of t, we will construct a list $\mathcal{L}_{v,0}$ which contains all the nodes $w \in [v - t + 1 : v]$. This list consists of all potential challenges defined by v at "level" 0. If $v \in [N]$ is also a multiple of $2t$, the sets $\mathcal{L}_{v-t,0}$ and $\mathcal{L}_{v,0}$ are merged into a unique set, denoted by $\mathcal{L}_{v,1}$, with t elements, in the following way: $L(v)$ is used as input to a random oracle to obtain random coins $r_{v,1}$. Using these random coins, we sample at random (and without replacement) a subset of size t from the set $\mathcal{L}_{v-t,0} \cup \mathcal{L}_{v,0}$. Such randomly sampled subset of size t will be stored as $\mathcal{L}_{v,1}$, and the sets $\mathcal{L}_{v-t,0}$ and $\mathcal{L}_{v,0}$ can be erased from memory. The set $\mathcal{L}_{v,1}$ will consist of t challenge nodes assigned to v at "level" 2. In a similar way, whenever v is a multiple of $2^i t$ for integer i, random coins $r_{v,i}$ are produced from $L(v)$ to obtain a random subset $\mathcal{L}_{v,i}$ of size t from the set $\mathcal{L}_{v-2^{i-1}t,i-1} \cup \mathcal{L}_{v,i-1}$. After obtaining $\mathcal{L}_{v,i}$, the sets $\mathcal{L}_{v-2^{i-1}t,i-1}$, and $\mathcal{L}_{v,i-1}$ are erased from memory. In the last step of the algorithm, the set $\mathcal{L}_{N,n-c}$ will be produced, where $N = 2^n$. This, together with $\phi_L := L(N)$, constitutes the proof to be verified.

Extra care is needed when proving soundness of the resulting ISC scheme, in which the prover learns partial information about its challenges, even before sending the commitment ϕ_L. The overall proof strategy is similar to proof of ITC from [7]. The intuition of why the on-the-fly sampling is sound comes from the observation that at each resampling node v, the samples are derived from randomness that depends on $L(v)$, and $L(v)$ serves as a commitment to all potential nodes from which the resampling takes place. The security proof then goes through a series of hybrid games, in which two consecutive games differ by a resampling-related bad event, which happens with a negligible probability. The analysis of the last hybrid game boils down to the original analysis of the SC scheme.

As mentioned before, when moving to general weight distributions, our on-the-fly sampling procedure follows the Poisson binomial distribution. This change introduces a novel issue: the number of final challenges (and thus their total weight) is not fixed in advanced as before, and more critically, the intermediate sets used in the sampling procedures are not implicitly defined. On the one hand, it can be shown using Hoeffding-like bounds, that the number of challenges (and their total weight) must be concentrated around the expected value with overwhelming probability. By modifying the verifier so that it additionally checks such constraints, it is possible to rule out such malicious behavior, and fix the first issue. On the other hand, by having the prover commit to the intermediate sets, and give, as part of its proof, the opening of the intermediate sets used in sampling the final challenge set, the verifier can check the correctness of the on-the-fly sampled challenges, and thus fix the second issue as well. A more elaborate overview is given in Sect. 6.

2 Preliminaries

2.1 Notations

For integers $m, n > 0$, define $[n] := \{1, \ldots, n\}$, $[n]_0 := [n] \cup \{0\}$, and $[m : n] := \{m, m+1, \ldots, n\}$. For a DAG $G = (V, E)$ and $v \in V$, we let $\mathsf{Parents}(v) := (v_1, \ldots, v_k)$ be the parents of v given in *reverse topological ordering.*[4]

For a distribution $\mathcal{D}$, we denote by $d \xleftarrow{\$} \mathcal{D}$ sampling d according to $\mathcal{D}$ (in case $\mathcal{D}$ is a set, the uniform distribution is implied).

We define the notion of (oracle-based) graph labeling which appeared in previous work on GL-PoSW schemes, say [12, 5].

[4] Although any fixed ordering suffices, it would be convenient for our construction to consider the reverse topological ordering.

2.2 Graph Labeling

Definition 1 (Oracle-based graph labeling). *Let $G_N = ([N]_0, E_N)$ be a DAG and $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ an oracle. We define the τ-labeling $L^\tau : [N]_0 \to \{0,1\}^\lambda$ of G_N recursively as*

$$L^\tau(i) := \begin{cases} \tau(i) & \text{if } \mathsf{Parents}(i) = \emptyset \\ \tau\big(i, L^\tau(i_1), \ldots, L^\tau(i_k)\big) & \text{else, i.e., } \mathsf{Parents}(i) = (i_1, \ldots, i_k) \end{cases} \tag{1}$$

When τ is clear from the context, we simply write L.

To formalize consistency of labels in this context, [2] define the notion of consistent strings, which is stronger than prior definitions in the literature [12, 5, 3]. Intuitively, to each vertex i a value y_i is associated, which represents the concatenation of the labels of the *parents* of i. In order to reason about the label of the last node as well, a dummy vertex is introduced for it, that is, we add vertex $N+1$ and an edge $(N, N+1)$.

Definition 2 (Consistent strings [2]). *Let $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ be an oracle. For a DAG $G_N = ([N]_0, E_N)$, let $G_N^+ = ([N+1]_0, E_N^+)$ with $E_N^+ = E_N \cup \{(N, N+1)\}$. Furthermore, $\forall i \in [N+1]_0$, let p_i be the number of parents of i in G_N^+ and $y_i := (i, y_i[1], \ldots, y_i[p_i]) \in ([N]_0 \times (\{0,1\}^\lambda)^{p_i})$. We say y_i is* consistent *with $y_{i'}$ w.r.t. G_N, and denote it by $y_i \prec y_{i'}$ if $(i, i') \in E_N^+$ and if i is the j-th parent of i' in G_N^+ (in reverse topological order), then the j-th block in the decomposition of $y_{i'}$ is equal to $\tau(y_i)$, i.e., $y_{i'}[j] = \tau(y_i)$.*

3 The Skiplist PoSW Scheme

3.1 Construction

In this section, we review the interactive GL-PoSW scheme of [2]. The scheme is reminiscent to the PoSW scheme from [3]. The scheme follows the same design template introduced earlier and we give a formal definition only for the non-interactive incremental GL-PoSW schemes later in Sect. 4.

Definition 3 (The Skiplist graph [3, 2]). *Let $G_N = ([N]_0, E_N)$ be a DAG with*

$$E_n = \left\{(i,j) \in ([N]_0)^2 : \exists k \geq 0 \text{ s.t. } (j-i) = 2^k \wedge 2^k | i \right\} \ .$$

Definition 4 (Labeled Paths). *Let $G_N = ([N]_0, E_N)$ be a skiplist graph as in Def. 3 and L^τ a labeling over its vertices for an oracle $\tau : \{0,1\} \to \{0,1\}^\lambda$. For integers $i_1, \ldots, i_j$ s.t. $0 \leq i_1 < \cdots < i_j \leq N$, define $\mathsf{Path}(i_1, \ldots, i_j)$ as the unique shortest path from i_1 to i_j passing through $i_1, \ldots, i_j$. Furthermore, define*

$$\mathsf{Path}^+(i_1, \ldots, i_j) := (\mathsf{Parents}(v))_{v \in \mathsf{Path}(i_1, \ldots, i_j)}$$
$$\mathsf{Path}^*(i_1, \ldots, i_j) := (v, L(\mathsf{Parents}(v)))_{v \in \mathsf{Path}(i_1, \ldots, i_j)}$$

and a predicate Consistent *over labeled paths as follow:*

$\mathsf{Consistent}\,(\mathsf{Path}^*(i_1,\ldots,i_j)) \in \{0,1\}$*: output* 1 *iff*
$\forall y_i := (v_i,\cdot), y_{i'} := (v_{i'},\cdot) \in \mathsf{Path}^*(i_1,\ldots,i_j)$ *s.t.* $(v_i,v_{i'}) \in E_N$*:*

$$y_i \prec y_{i'} \quad \textit{where } \prec \textit{ is as in Def. 2.}$$

To illustrate, consider G_8, then

$$\begin{aligned}
&\mathsf{Path}(0,3,8) = (0,2,3,4,8)\\
&\mathsf{Path}^+(0,3,8) = ((),(1,0),(2),(3,2,0),(7,6,4,0))\\
&\mathsf{Path}^*(0,3,8) = ((0),(2,L(1),L(0)),(3,L(2)),(4,L(3),L(2),L(0)),\\
&\qquad\qquad\qquad (8,L(7),L(6),L(4),L(0))).
\end{aligned}$$

Furthermore, $\mathsf{Consistent}(\mathsf{Path}^*(0,3,8)) = 1$ as

$$y_0 \prec y_2,\ y_0 \prec y_4,\ y_0 \prec y_8,\ y_2 \prec y_3,\ y_2 \prec y_4,\ y_3 \prec y_4,\ y_4 \prec y_8.$$

Note that Path^* contains redundant labels. This results in an increase in the proof size in the skiplist-based PoSW scheme. However, we keep it as is for the simplicity of exposition. In practical realizations, it is straightforward to remove the redundancy and modify the verification algorithm accordingly.

Now we describe the PoSW scheme from [2] in Fig. 2. Formally the PoSW is defined for a graph family $(\Gamma_N := (G_N, \Omega_N))_{N\in\mathbb{N}}$ where G_N is a skiplist as in Def. 3 and a weight function $\Omega_N : [N] \to [0,1]$ where $\sum_{i\in[N]} \Omega_N(i) = 1$. Besides, the PoSW scheme is parameterized by a security parameter $t \in \mathbb{N}$ that determines the number of challenges the verifier samples from Ω_N.

In [2], interactive GL-PoSW schemes were defined. Their definition generalizes existing definitions of PoSW in a few directions. First, the protocol's underlying graph is a weighted DAG $\Gamma_n := (G_N, \Omega_N)$ where G_N is a DAG on $[N]_0$ vertices, and the the verifier's challenges are drawn according to a weigh function $\Omega_N : [N]_0 \to [0,1]$ where $\sum_{i\in[N]_0} \Omega_N(i) = 1$. Second, they define *knowledge* soundness in addition to the classical notion of soundness. In Theorem 1, we restate their main theorem about the classical, rather than knowledge, soundness guarantees of the PoSW construction depicted in Fig. 2.

Towards generalizing PoSW to arbitrary weight functions, the weight of a sequence of parallel oracle queries to $\tau(\cdot)$ is defined. A parallel query is a sequence of simultaneous queries to τ, i.e. a sequence $((x_1,i_1),\ldots,(x_m,i_m))$ which is answered by $(\tau(i_1,x_1),\ldots,\tau(i_m,x_m))$. Intuitively, the weight of such a sequence is the sum of the respective "heaviest" nodes in each parallel query.

Definition 5 (Sequential weight [2]). *Let* $Q = (Q_1,\ldots,Q_\ell)$ *be a sequence of parallel queries to an oracle* τ*. We define the* sequential weight *of* Q *with respect to a weight function* $\Omega_N : [N] \to [0,1]$ *where* $\sum_{i\in[N]} \Omega_N(i) = 1$ *as*

$$\Omega_{seq}(Q) := \sum_{i=1}^{\ell} \max\{\Omega_N(j) : Q_i \text{ contains a query to } \tau(j,\cdot)\}\ .$$

```
Prover $\mathsf{P} = (\mathsf{P}_0, \mathsf{P}_1)$ :
Stage $\mathsf{P}_0$: On input $(1^\lambda, 1^N)$ and $\chi$:
  1. Compute the $\tau$-labeling $L$ of $G_N$ using oracle $\tau(\chi, \cdot)$
  2. $\phi_L := L(N)$
  3. send $\phi_L$ to $\mathsf{V}_1$
Stage $\mathsf{P}_1$:
  1. $\forall i \in [t]$, $y_i := \mathsf{Path}^*(0, \iota_i, N)$ where $\mathsf{Path}^*$ is as in Def. 4.
  2. send $y := (y_1, \ldots, y_t)$ to $\mathsf{V}_2$

Verifier $\mathsf{V} = (\mathsf{V}_0, \mathsf{V}_1, \mathsf{V}_2)$:
Stage $\mathsf{V}_0$: On input $(1^\lambda, N)$:
  1. $\chi \leftarrow \{0,1\}^\lambda$
  2. send $\chi$ to $\mathsf{P}_0$
Stage $\mathsf{V}_1$: On input $\phi_L$:
  1. $\forall i \in [t]$ do $\iota_i \xleftarrow{\$} \Omega_N$
  2. send $\iota = (\iota_i)_{i=1}^t$ to $\mathsf{P}_1$
Stage $\mathsf{V}_2$: On input $y = (y_i)_{i=1}^t$:
  1. $\forall i \in [t]$ do
     (a) parse $y_i = (y_{i_1}, \ldots, y_{i_k})$ where $(i_1, \ldots, i_k) := \mathsf{Path}(0, \iota_i, N)$
     (b) $b_i := 1$ iff
         i. $\tau(y_{i_k}) = \phi_L$ and
         ii. $\mathsf{Consistent}(y_i) = 1$
             where $\mathsf{Consistent}$ is as in Def. 4.
  2. output $\bigwedge_{i=1}^t b_i$
```

Fig. 2. $t \in \mathbb{N}$ is a parameter of the scheme, and (G_N, Ω_N) is s.t. G_N is a skiplist DAG as in Def. 3 and $\Omega_N : [N] \to [0,1]$ is a weight function, i.e., it satisfies $\sum_{i \in [N]} \Omega_N(i) = 1$.

The honest P in Fig. 2 defines a sequence $Q = (Q_1, \ldots, Q_N)$ of queries with $Q_i := \{(i, L(\mathsf{Parents}(i)))\}$ and it therefore holds that $\Omega_{seq}(Q) = 1$. To capture the sequential work of malicious provers, the notion of τ-sequence is defined.

Sequentiality of random oracles is formulated in terms of RO-sequences, which appeared in slightly different formulations and bounds in [12, 5, 3, 2]. Below we adopt the formulation from [2].

Definition 6 (τ-sequences [2]). *Let $G_N = ([N]_0, E_N)$ be a DAG and $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle. We call a sequence of strings $s := (y_{i_1}, \ldots, y_{i_{\ell+1}})$ with $y_{i_{\ell+1}} \in \{0,1\}^\lambda$ and $y_{i_j} \in ([N]_0 \times (\{0,1\}^\lambda)^{|\mathsf{Parents}(i_j)|})$ a* τ-sequence *of length ℓ if $\forall j \in [\ell], y_{i_j} \prec y_{i_{j+1}}$ w.r.t. G_N.($\prec$ is as in Def. 2.)*

For a weight function $\Omega_N : [N] \to [0,1]$, the weight of a τ-sequence $s = (y_{i_1}, \ldots, y_{i_{\ell+1}})$ is defined as $\Omega_N(s) := \sum_{j=1}^{\ell} \Omega(i_j)$.

Note the honest Q above can be made into a τ-sequence $s := (y_0, \ldots, y_N, y_{N+1})$ where $\forall i \in [N]_0, y_i := (i, L(\mathsf{Parents}(i)))$ and $y_{N+1} := \tau(y_N)$. Note the weight of s is defined to ignore the last index $N+1$ and hence is equal $\sum_{i=1}^N \Omega(i) = 1$. Furthermore, note if $\mathsf{Path}^*(0, i, N)$ is such that $\mathsf{Consistent}(\mathsf{Path}^*(0, i, N)) = 1$, then by definition $(\mathsf{Path}^*(0, i, N), \phi_L)$ constitutes a τ-sequence. Lemma 1 shows that, except with negligible probability, no malicious prover which makes a sequence of parallel queries Q with $\Omega_{seq}(Q) < \alpha$ can produce a τ-sequence of weight α.

Lemma 1 (Sequentiality of τ [2]). *Let $\Gamma_N = (G_N, \Omega_N)$ be a weighted DAG and $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ a random oracle. Let $\tilde{\mathsf{P}}^{\tau(\cdot)}$ be a malicious prover that* *makes (parallel) queries to $\tau(\cdot)$ of sequential weight $< \alpha$ and makes q oracle queries in total. Then the probability that $\tilde{\mathsf{P}}^{\tau(\cdot)}$ outputs a τ-sequence of weight α can be bounded by*

$$\Pr\left[s \leftarrow \tilde{\mathsf{P}}^{\tau(\cdot)} : \ s \textit{ is a } \tau\textit{-sequence} \wedge \Omega_N(s) = \alpha\right] \leq \frac{1}{2^\lambda} + \frac{q^2}{2^\lambda} = \frac{q^2+1}{2^\lambda}.$$

Now we state the theorem that shows the PoSW scheme in Fig. 2 is secure.

Theorem 1 ([2]). *Consider a malicious prover $\tilde{\mathsf{P}}$ against V from Fig. 2. If $\tilde{\mathsf{P}}$ makes a sequence Q of parallel queries to τ of sequential weight $\Omega_{seq}(Q) < \alpha \in (0,1]$ and a total number of queries q, then $\tilde{\mathsf{P}}$ can make V accept with probability at most $\epsilon := \alpha^t + 3 \cdot q^2/2^\lambda$.*

3.2 Prover Efficiency and Space-Time Tradeoffs

In this section, we give general space-time tradeoffs on the prover's strategy in the interactive PoSW from [2]. These tradeoffs pave the way to the incremental non-interactive PoSW scheme we give in Sect. 5. For space constraints, we give the proofs of the lemmas of this section in the full version of the paper.

The following lemma shows that the prover can label the skiplist graph in small space complexity.

Lemma 2. *Let G_N be the skiplist graph from Def. 3 and $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ an oracle. Then G_N for $N = 2^n$ can be τ-labeled in topological order using at most $(n+1)\lambda$ bits of memory*

Let $N = 2^n$ and the level of a node denote its in-degree. In G_N, we have only the source node 0 with level 0, only the sink node N with level $n+1$, and for every $i \in [n]$, we have 2^{n-i} nodes with level i. For simplicity of exposition, we treat the source node differently and assume that its label is always known. When storing labels of nodes of level m and greater, the prover needs to store the labels of $\sum_{i=m}^{n} 2^{n-i} + 1 = 2^{n-m+1}$ nodes. Recomputing the labels of the remaining nodes can be done by making $2^{m-1} - 1$ queries sequentially; this can be done assuming 2^{n-m+1} parallel processors by partitioning 2^n into intervals of length 2^{m-1}. To compute how many (sequential) queries are required to compute all the labels necessary to correctly reply to some challenge $i \in [N]$, we need to first analyze how many nodes' labels will be needed in answering that challenge.

Lemma 3. *Le G_N be the graph from Def. 3 with $N = 2^n$, and let $k \in [n+1]$. If a challenge hits a k-level node, then the prover can convince the verifier by providing the labels of $\phi_n(k) := 2 + \frac{1}{2}(n-k+2)(n+k-1)$ vertices.*

The following lemma says that, when storing labels of all nodes of level greater than or equal to m, given $\rho(n,m,k)$ parallel processors, it is possible to reply to the challenge query making only $2^{m-1} - 1$ sequential hash queries.

Lemma 4. *If a challenge hits a k-level node, $k \in [n+1]$, of a graph with $N = 2^n + 1$ nodes, where $n \geq 2$, then a prover storing the labels of all nodes of level greater than or equal to m, for $m \in [n+1]$, will have to make $\rho(n,m,k) \cdot (2^{m-1} - 1)$ many queries, where*

$$\rho(n,m,k) := \begin{cases} 0 & \text{if } m = 0, \\ n-k+2 & \text{if } m < k, \\ n-m+2 & \text{otherwise.} \end{cases}$$

4 Incremental Proofs of Sequential Work

We provide a definition of incremental proofs of sequential work that is stronger than the definition given by [7]. In fact, their construction as well as ours achieve the stronger definition. Our definition is stronger in the sense that it guarantees that honestly incrementing *any valid proof*, regardless of how it is generated, makes the verifier accept.

This issue is particularly important in the context of graph-labeling proofs of sequential work schemes as these are not proofs of *correctness* of the sequential computation, but rather are proofs of sequential computation. Consider any graph-labeling PoSW with underlying graph G_N and a prover that follows the honest prover strategy except for a randomly chosen $i \leftarrow [N]$, it sets $L(i) = 0^\lambda$ and continues the computation correctly. Such a prover has an overwhelming probability of convincing the verifier, although the proof was not honestly generated, and with overwhelming probability the proof will be different from the honestly generated proof. However, this is not a problem of the concept of a PoSW, as still the malicious prover did $N-1$ computational steps sequentially.

Definition 7 (Incremental PoSW). *Let $\Gamma = (\Gamma_N = (G_N, \Omega_N))_{N \in \mathbb{N}}$ be a family of weighted DAGs such that for all N, G_N has a unique sink N. A tuple of oracle aided PPT algorithms $(\mathsf{P}^{\tau(\cdot)}, \mathsf{Inc}^{\tau(\cdot)}, \mathsf{V}^{\tau(\cdot)} := (\mathsf{V}_0^{\tau(\cdot)}, \mathsf{V}_1^{\tau(\cdot)}))$ for an oracle $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ is an* incremental (non-interactive) proof of sequential work *w.r.t. τ if the following properties hold:*

Completeness: *For every $\lambda, N \in \mathbb{N}$, every $(\chi, N, \mathsf{state}) \leftarrow \mathsf{V}_0^{\tau(\cdot)}(1^\lambda, N)$ and*

- *honestly generated $\pi \leftarrow \mathsf{P}^{\tau(\cdot)}(1^\lambda, 1^N, \chi)$ or*
- *honestly incremented $\pi \leftarrow \mathsf{Inc}^{\tau(\cdot)}(1^\lambda, 1^{N''}, \chi, N', \pi')$ for integers N', N'' s.t. $N = N' + N''$ and π' is* accepting[5] *proof for parameter N', i.e., $\mathsf{V}_1^{\tau(\cdot)}(\mathsf{state}, \chi, N', \pi') = 1$,*

it holds that $\Pr\left[\mathsf{V}_1^{\tau(\cdot)}(\mathsf{state}, \chi, N, \pi) = 1\right] \geq 1 - \mathsf{negl}(\lambda)$.

(α, ϵ)**-Soundness:** *For every $\lambda, N \in \mathbb{N}$ and every PPT adversary $\tilde{\mathsf{P}}^{\tau(\cdot)}$ which makes a sequence Q of parallel queries to τ of sequential weight $\Omega_{seq}(Q) < \alpha$:*

[5] We consider $\mathsf{Inc}^{\tau(\cdot)}$ to be honest regardless of how π' was generated. In contrast, [7] defines honest $\mathsf{Inc}^{\tau(\cdot)}$ only over honestly generated π'.

$$\Pr\left[\begin{array}{c}(\chi, N, \mathsf{state}) \leftarrow \mathsf{V}_0^{\tau(\cdot)}(1^\lambda, N);\\ \pi \leftarrow \tilde{\mathsf{P}}^{\tau(\cdot)}(1^\lambda, 1^N, \chi)\end{array} : \ \mathsf{V}_1^{\tau(\cdot)}(\mathsf{state}, \chi, N, \pi) = 1\right] \leq \epsilon(\lambda).$$

Succinctness: *For every $\lambda, N \in \mathbb{N}$ and every honestly generated proof π for parameter N, we have $|\pi| \leq \mathsf{poly}(\lambda, \log N)$ and $\mathsf{V}^{\tau(\cdot)}$ runs in time $\mathsf{poly}(\lambda, \log N)$.*

Remark 1. For $\mathsf{Inc}^{\tau(\cdot)}$ to be non-trivial, note that $\mathsf{Inc}^{\tau(\cdot)}$ runs in time that is essentially independent of N'; it gets N' in binary.

Remark 2. For some applications, standard (α, ϵ)-soundness might not be enough, and *knowledge* soundness might be required. It is straightforward to define knowledge soundness for Def. 7 exactly the same way knowledge soundness of GL-PoSW from [2] is defined.

5 A Skiplist-Based Incremental PoSW Scheme

In this section, we construct an incremental GL-PoSW scheme based on the skiplist graph G_N, Ω_N where Ω_N is defined as $\forall i \in [N] : \Omega_N(i) = 1/N$. A construction of an iPoSW for more general families of weight distributions Ω_N is given in Sect. 6.

In Section 5.2, we will give a high-level overview of our construction, using some concrete example to better explain our design. A formal description of the algorithms can be found in Section 5.3. The security proof is in Section 5.5.

5.1 Parameters

Our iPoSW scheme depends on the following parameters and objects.

- A time parameter N of the form $N = 2^n$, for some integer $n \in \mathbb{N}$.
- A computational security parameter λ.
- A statistical security parameter t.
- Let $\tau : \{0,1\}^* \to \{0,1\}^\lambda$ be a random oracle, we define two oracles, τ_ℓ, τ_r as
$$\tau_\ell(\cdot) := \tau(0, \chi, \cdot) \quad \tau_r(\cdot) := \tau(1, \chi, \cdot) \qquad (2)$$
- $(i_1, \ldots, i_t) := \mathsf{Sample}(r)$: on input random coins r, uniformly sample $(i_1, \ldots, i_t)$ from all possible such sequences $\binom{2t}{t}$. Since $\binom{2t}{t} < (\frac{2t \cdot e}{t})^t = (2e)^t$, where $\log_2(2e) \approx 2.44$, random coins of size $3t$ are sufficient to sample statistically close to a uniform subset.

Remark 3. Suppose that we have two sets S_1, S_2 with t elements each, where in S_1 the elements are indexed from 1 to t, and in S_2 from $t+1$ to $2t$. Moreover, suppose that there are M_1 elements having a *specific feature* in S_1 and M_2 having the same feature in S_2. Then, using Sample, one can sample a subset from $S_1 \cup S_2$ of size t, where the distribution of the cardinality of elements with this same specific feature is described by a random variable X distributed as $\mathsf{Hypergeometric}(2t, M_1 + M_2, t)$.

For simplicity of exposition, we assume that $t = 2^c$ for some $c \in \mathbb{N}$. Our construction can be easily adapted to the more general case where t is arbitrary.

5.2 A High-Level Overview

Before formally describing the algorithms defining the iPoSW scheme, we give an intuition of how they work.

We start by describing the PoSW scheme upon which we build. The scheme is described in Fig. 2. In such interactive protocol, the first step is performed by the verifier V, which samples a random statement χ with enough min entropy and sends it to the prover P, which in turn uses χ to refresh the common random oracle $\tau(\cdot)$ as $\tau_\ell(\cdot) := \tau(0, \chi, \cdot)$ and computes a τ_ℓ-labeling L of G_N. Then P sends $\phi_L := L(N)$ to V. As shown in [2], ϕ_L constitutes a (position-binding) commitment to L.[6] After receiving ϕ_L, V samples t challenge nodes and sends them to P, which in turn sends to V valid openings for all the received challenges. In the last step of the interaction, V checks the consistency of the openings and accepts or rejects accordingly.

This protocol can be made non-interactive using the Fiat-Shamir heuristic: the challenges chosen by V are now produced by P itself by querying a random oracle $\tau_r(\cdot) := \tau(\chi, 1, \cdot)$ on $(\phi_L, 1), \ldots, (\phi_L, t)$. Now V, besides verifying the consistency of the openings, must also verify the correctness of the received challenges, by recomputing them based on ϕ_L and $\tau_r(\cdot)$.

Using this protocol, in order to compute a proof, the prover has to either remember the N labels for the entire G_N graph, or recompute the labels required in the proof, once the challenge nodes are fixed, which may require up to N sequential invocations to τ_ℓ. Alternatively, it is possible to have general space-time trade-offs, as shown in Lemma 4: the prover upon spending N sequential invocation to τ_r, could store $\approx 2^{n-m}$ labels from L, and perform an additional $\approx 2^{m-1}$ sequential computation using $\rho(n, m, k) \leq n - m + 2$ parallel processors, for any $m \in [n+1]$. For example, taking $m = n/2$, yields a protocol where the prover uses $\approx \sqrt{N}$ memory, and $\approx N + \sqrt{N}/2$ sequential work, while using $\approx (\log N)/2$ processors.

Still, there is a concrete substantial gap between, on the one hand, the sequential work the honest prover has to actually perform and, on the other hand, the sequential work it is able to convince the verifier of. For example, a memory-efficient prover, i.e., one where $m = n$, has to perform $2N$ sequential queries to τ_r in order to show it did N sequential queries. For large values of N, this gap becomes considerable; say N steps would take approximately a year to compute, then to generate a proof of that, one would need another year.

As first observed by Döttling, Lai and Malavolta in [7], at the core of this slack is the fact that the challenge nodes are determined solely by ϕ_L. Therefore, the prover has to label the entire graph before knowing which labels it has to recompute and include in the proof.

The problem was first tackled by [7], who introduced the idea of letting the prover choose the challenge nodes *on the fly*: the prover chooses the random challenges as it labels the graph, and eventually discards some of them as the graph

[6] This is reminiscent to the fact that a Merkle tree's root commits to its leaves (and internal nodes).

gets labeled. This allows the prover to compute a valid proof using *a single pass* over the graph. However, since now the prover knows partial information about the possible challenges while labeling the graph, soundness of the so obtained protocol has to be carefully studied.

We adapt the on-the-fly sampling technique of [7] to the skiplist-based PoSW scheme. Let t be the number of challenge nodes/openings to be produced by the end of the protocol (for ease of notation we will consider $t = 2^c$ to be a power of 2). For each node v in $[N]$ which is a multiple of t, we will construct a list $\mathcal{L}_{v,0}$ which contains all the nodes $w \in [v - t + 1, v]$. This list will represent the challenge node assigned to v at "level" 0.

When v is also a multiple of $2t$, the sets $\mathcal{L}_{v-t,0}$ and $\mathcal{L}_{v,0}$ are merged into a unique set, denoted by $\mathcal{L}_{v,1}$, with t elements, in the following way: $L(v)$ is used as input to the random oracle τ_r to obtain random coins $r_{v,1}$. Using these random coins, we sample a subset $\mathcal{L}_{v,1}$ of size t uniformly at random from $\mathcal{L}_{v-t,0} \cup \mathcal{L}_{v,0}$. Once $\mathcal{L}_{v,1}$ is stored, $\mathcal{L}_{v-t,0}$ and $\mathcal{L}_{v,0}$ are erased from memory. The set $\mathcal{L}_{v,1}$ consists of t challenge nodes assigned to v at "level" 2. In a similar way, whenever v is a multiple of $2^i t$, random coins $r_{v,i}$ are produced from the labeling of v, to obtain a random subset $\mathcal{L}_{v,i}$ of size t from the set $\mathcal{L}_{v-2^{i-1}t,i-1} \cup \mathcal{L}_{v,i-1}$. After obtaining $\mathcal{L}_{v,i}$, the sets $\mathcal{L}_{v-2^{i-1}t,i-1}$, and $\mathcal{L}_{v,i-1}$ are erased from memory.

In the last step of the algorithm, the set $\mathcal{L}_{N,n-c}$ will be produced. This, together with $\phi_L = L(N)$, constitute the proof to be verified.

We depict the sampling process pictorially for graph G_{16} and parameter $t = 2$, i.e., $n = 4$, and $c = 1$, in Fig. 1. The prover algorithm P labels all nodes of the graph in topological order. Once it has labeled the first 2 nodes, it creates the set $\mathcal{L}_{2,0}$, which contains the nodes 1 and 2 as possible challenges. It then continues to label the graph. When it reaches node 4, it first creates the set $\mathcal{L}_{4,0}$, which contains the nodes 3 and 4 as possible challenges. From the union of these two sets, a random subset of 2 elements is chosen to obtain the set $\mathcal{L}_{4,1}$. In the last step of the protocol, lists $\mathcal{L}_{16,0}$, $\mathcal{L}_{14,0}$, $\mathcal{L}_{12,1}$, and $\mathcal{L}_{8,2}$ are merged in succession to finally obtain $\mathcal{L}_{16,3}$.

As already anticipated, in the actual protocol the lists $\mathcal{L}_{w,j}$, for nodes $w \in [N]$ and $j \in [n - c]$ contain more than simple challenge nodes. Each element in $\mathcal{L}_{w,j}$ is an opening (a labeled path) for a challenge node in $[w - 2^j t + 1 : w]$ with respect to $G_N|_{[w-2^j t:w]}$, where $G|_{V'}$, with $V' \subset V$, denotes the subgraph of G induced by the vertex set V', i.e., the graph G' with vertex set V' and edge set consisting of those edges both of whose endpoints are in V'.

Given $\mathcal{L}_{v-2^{i-1}t,i-1}$ and $\mathcal{L}_{v,i-1}$ with openings (labeled paths), rather than challenges, we show how to construct $\mathcal{L}_{v,i}$. As before, using $L(v)$ we extract randomness to sample a random subset of size t from $\mathcal{L}_{v-2^{i-1}t,i-1} \cup \mathcal{L}_{v,i-1}$. Moreover, suppose that the first element sampled in this way is $\mathcal{L}_{v,i-1}[b]$, for some $b \in [t]$. Now $\mathcal{L}_{v,i-1}[b]$ is an opening of some challenge node in $[v - 2^{i-1}t + 1 : v]$ with respect to $G_N|_{[v-2^{i-1}t:v]}$. Since elements in $\mathcal{L}_{v,i}$ have to be valid openings of some challenge node in $[v - 2^i t + 1 : v]$ with respect to $G_N|_{[v-2^i t:v]}$, simply adding $\mathcal{L}_{v,i-1}[b]$ to $\mathcal{L}_{v,i}$ won't work. Instead, we extend $\mathcal{L}_{v,i-1}[b]$ to a valid opening over the whole subgraph $G_N|_{[v-2^i t:v]}$. This can be done, using the structure of the skiplist graph and the definition of shortest path, by simply pre-appending the missing labeled edge $(v - 2^{i-1}t, L(\mathsf{Parents}(v - 2^{i-1}t)))$ to $\mathcal{L}_{v,i-1}[b]$. Similarly, any element in $\mathcal{L}_{v-2^{i-1}t,i-1}$ can be extended to a valid opening in $G_N|_{[v-2^i t:v]}$ by appending $(v, L(\mathsf{Parents}(v))$ to it. This step of the algorithm is formally described by (4) in the scheme description.

The prover's final proof is $\pi = (\phi_L, \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c})$. For each $b \in [t]$, the verifier checks that (1) $\mathcal{L}_{N,n-c}[b]$ is consistent with ϕ_L (2) that $\mathcal{L}_{N,n-c}[b]$ is internally consistent, and (3) that the challenge in $\mathcal{L}_{N,n-c}[b]$ is consistent with the on-the-fly-sampling. The first two checks are similar to the base PoSW scheme. To check (3), algorithm Check of Fig. 3 is run: on input $\mathcal{L}_{v,i}[b]$ and $\mathcal{I}_{v,i}[b]$, it recomputes the randomness $r_{v,i}$ used to sample elements from $\mathcal{L}_{v-2^{i-1}t,i-1}[b]$ and $\mathcal{L}_{v,i-1}[b]$, checks if this is consistent with what is stored in $\mathcal{I}_{v,i}[b]$, and determines whether $\mathcal{L}_{v,i}[b]$ was selected from $\mathcal{L}_{v-2^{i-1}t,i-1}[b]$ or from $\mathcal{L}_{v,i-1}[b]$. If $\mathcal{L}_{v,i}[b]$ was obtained by extending some element of $\mathcal{L}_{v-2^{i-1}t,i-1}[b]$ to $G_N|_{[v-2^i t,v]}$, then it must be the case that the last two nodes are exactly $v - 2^{i-1}t$ and v. If this is not the case, the algorithm returns immediately 0. Otherwise, $(v, L(\mathsf{Parents}(v))$, gets removed from it, and Check is run recursively on the so obtained opening.

5.3 Scheme Description

$\underline{\mathsf{P}^{\tau(\cdot)}(1^\lambda, 1^N, N)}$:

1. Traverse the graph $G_n = (V = [N]_0, E)$ in topological order, starting from 0. At every node $v \in [N]_0$ which is traversed, do the following:
   (a) Compute $L(v)$ according to (1).
   (b) If $t \mid v$ and $v \in [N]$, write $v = 2^k \cdot h \cdot t$, with h odd and $k \in \mathbb{N}_0$. For $j \in [t]$:
$$\mathcal{L}_{v,0}[j] := \mathsf{Path}^*(v - t, v - t + j, v), \qquad \mathcal{I}_{v,0}[j] := j. \tag{3}$$
   If $k \geq 1$, do the following for $i \in [k]$:
      i. Compute $r_{v,i} := \tau_r(v, i, L(v))$.
      ii. Choose a random t-subset $S_{v,i}$ of $[2t]$ via $S_{v,i} := \mathsf{Sample}(r_{v,i})$.
      iii. Set $u := v - 2^{i-1} \cdot t$.
      iv. For $j \in [t]$, write $S_{v,i}[j] = at + b$ with $a \in \{0, 1\}$ and $b \in [t]$, and set
$$\mathcal{L}_{v,i}[j] := \begin{cases} (\mathcal{L}_{u,i-1}[b], (v, L(\mathsf{Parents}(v)))) & \text{if } a = 0 \\ ((u, L(\mathsf{Parents}(u))), \mathcal{L}_{v,i-1}[b]) & \text{if } a = 1 \end{cases} \tag{4}$$
$$\mathcal{I}_{v,i}[j] := \begin{cases} \mathcal{I}_{u,i-1}[b], j & \text{if } a = 0 \\ \mathcal{I}_{v,i-1}[b], j & \text{if } a = 1 \end{cases}$$
      v. Store $\mathcal{L}_{v,i}$ and $\mathcal{I}_{v,i}$ in memory. Note that by design, $\mathcal{L}_{v,i}[j]$ satisfies
$$\mathsf{Consistent}(\mathcal{L}_{v,i}[j]) = 1. \tag{5}$$
      vi. Erase $\mathcal{L}_{u,i-1}$, $\mathcal{L}_{v,i-1}$, $\mathcal{I}_{u,i-1}$, and $\mathcal{I}_{v,i-1}$ from memory.
2. Terminate and output $\pi := (\phi_L := L(N), \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c})$.

$\underline{\mathsf{Inc}^{\tau(\cdot)}(1^\lambda, 1^{N'}, N, \pi, \chi)}$:

```
1. Parse $\pi$ as $(\phi_L, \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c})$.
2. Compute $N + N' = N''$ and check that $N'' = 2^{n''}$ for some $n'' \in \mathbb{N}$,
3. Execute the algorithm $\mathsf{P}^{\tau(\cdot)}(1^\lambda, 1^{N''}\chi)$ starting from step 1.(a) with a slight
   change: traverse the graph $G_{2^{n''}}$ starting from $N + 1$.
```

$\underline{\mathsf{V}^{\tau(\cdot)}(1^\lambda, 1^N, \pi, \chi)}$:

```
1. Parse $\pi$ as $(\phi_L, \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c})$.
2. For all $i \in [t]$ :
   (a) parse $\mathcal{L}_{N,n-c}[i]$ as $y_i = (y_{i_1}, \ldots, y_{i_k})$, for some $k \in \mathbb{N}$.
   (b) $b_i := 1$ iff the following hold
       i. $\tau_\ell(y_{i_k}) = \phi_L$ and
       ii. $\mathsf{Consistent}(y_i) = 1$ where $\mathsf{Consistent}$ is as in Def. 4.
       iii. $\mathsf{Check}(\mathcal{L}_{N,n-c}[i], \mathcal{I}_{N,n-c}[i], 0, N, n-c) = 1$, where the algorithm $\mathsf{GCheck}$
            is described in Fig. 3.
3. Return $\bigwedge_{i=1}^{t} b_i$
```

Algorithm Check

On input a path Path, a list of indices ind, a starting node s, an ending node e, and a recursion index d:

```
1. Parse $\mathsf{Path} := (y_1, \ldots, y_k)$ and $\mathsf{ind} := (i_0, \ldots, i_{\mathsf{d}}) \in [t]^{\mathsf{d}+1}$
2. Compute $r_{\mathsf{e},\mathsf{d}} := \tau_r(\mathsf{e}, \mathsf{d}, \tau_\ell(y_k))$ and $S_{\mathsf{e},\mathsf{d}} := \mathsf{Sample}(r_{\mathsf{e},\mathsf{d}})$
3. Write $S_{\mathsf{e},\mathsf{d}}[i_{\mathsf{d}}] = at + b$ with $a \in \{0, 1\}$ and $b \in [t]$
4. Define $\mathsf{m} := (\mathsf{e} - \mathsf{s})/2$
5. If $\mathsf{d} = 0$, do the following:
– Compute the challenge $c$ corresponding to $\mathsf{Path}$
– If $a = 0$, return 1 iff $\mathsf{s} + b = c$
– If $a = 1$, return 1 iff $\mathsf{s} + \mathsf{m} + b = c$
6. If $\mathsf{d} \geq 1$, do the following
– If $a = 0$ and $i_{\mathsf{d}-1} = b$, $\mathsf{Check}((y_1, \ldots, y_{k-1}), (i_1, \ldots, i_{\mathsf{d}-1}), \mathsf{s}, \mathsf{s} + \mathsf{m}, \mathsf{d} - 1)$
– If $a = 1$ and $i_{\mathsf{d}-1} = b$, $\mathsf{Check}((y_2, \ldots, y_k), (i_1, \ldots, i_{\mathsf{d}-1}), \mathsf{s} + \mathsf{m}, \mathsf{e}, \mathsf{d} - 1)$
– Return 0
```

Fig. 3. Description of the Check algorithm.

5.4 Efficiency Analysis

We now discuss the efficiency of our scheme in terms of proof size, computation, and communication.

Proof Size. The proof consists of the sink-label ϕ_L and two lists, $\mathcal{L}_{N,n-c}$ and $\mathcal{I}_{N,n-c}$, with t elements each. Each entry in $\mathcal{L}_{N,n-c}$ is a path of the form

$\mathsf{Path}^*(0, i, N)$ for some $i \in [N]$. By Lemma 3, it therefore consists of at most $2 + \frac{n(n+1)}{2} = O(n^2)$ labels and $n + 1 = O(n)$ indices. Each entry in $\mathcal{I}_{N,n-c}$ is a tuple of $n - c$ indices in $[t]$, which can therefore be represented using $\log t$ bits each. Since each label can be stored using λ bits, we get that the entire proof has size at most $O(t \cdot (\lambda \cdot n^2 + n)) = O(t \cdot \lambda \cdot n^2)$.

Prover Efficiency. The prover traverses the N nodes of the skiplist graph G_N in topological order. By the same argument used in [7], the challenges $\tau_r(\cdot)$ can be evaluated in parallel by computing $r_{v,i} := \tau_r(v, i, L(\mathsf{Parents}(v)))$, instead of $r_{v,i} := \tau_r(v, i, L(v))$, This is possible as both τ_ℓ and τ_r are random oracles. With such modification, τ_ℓ and τ_r can be evaluated in parallel, thus the parallel complexity is not increased by the evaluation of τ_r. Therefore, the parallel complexity of the prover is bounded by the time needed for $O(N)$ sequential calls to the random oracle.

As far as the memory complexity of the prover is concerned, by Lemma 2 we know that the labeling of the skiplist graph G_N can be computed using $(n+1)\lambda$ bits of memory. In our construction, the prover algorithm is moreover storing at most $n + 1$ lists $\mathcal{L}_{v,i}$, where at most 2 of them at any point share the same "level" i. Using Lemma 3 to compute the memory required to store all such lists, one obtains that the memory complexity of the prover is bounded by $O(t \cdot \lambda \cdot n^3)$.

Verifier Efficiency. The verifier need to check, for each opening of a challenge node that i) the opening is consistent with ϕ_L, ii) that the labels of the opening are consistent, and that iii) the opening is consistent with the randomness used while generating the proof. Each opening can be checked in parallel. Checking consistency of each label in a given opening can also be done in parallel. The runtime of the verifier is dominated by the runtime of the Check algorithm, which uses $O(n)$ time to verify that the opening is consistent with the randomness used in the proof. Therefore, the parallel time needed overall is $O(n \log t)$.

5.5 The Security Proof

Theorem 2. *Let $\Pi := (\mathsf{P}^{\tau(\cdot)}, \mathsf{V}^{\tau(\cdot)}, \mathsf{Inc}^{\tau(\cdot)})$ be as in Sect. 5.3 and q be an upper bound on the total number of queries to $\tau(\cdot)$, then Π is an (α, ϵ)-soundness incremental PoSW scheme, as per Def. 7, with any $\alpha \in (0, 1]$ and*

$$\epsilon = \frac{1 + q^2}{2^\lambda} + \frac{q(q-1)}{2^{\lambda+1}} + q \cdot e^{-2t \cdot \left(\frac{1-\alpha}{n}\right)^2} \quad .$$

We remark that the weighted skiplist DAG (G_N, Ω_N) over which Π works is such that Ω_N is defined as $\forall i \in [N] : \Omega_N(i) = 1/N$. This induces the uniform distribution over the $[N]$ challenges.

Proof. (of Theorem 2) Completeness and succinctness are clear from the discussion in Sect. 5.4. We analyze soundness. The soundness guarantees of our construction rely on the soundness guarantees of the original interactive PoSW construction from Sect 5.3 modulo a number of hybrids that reflect the more power the adversary $\tilde{\mathsf{P}}$ has in the security experiment of the incremental PoSW scheme. This power comes from the fact that $\tilde{\mathsf{P}}$ gets to know some of its possible challenges early on during its computation. More precisely, before computing the label of the sink node which defines the set of challenges, $\tilde{\mathsf{P}}$ gets to know that some nodes would not belong to its challenges. We will show that this extra power gives $\tilde{\mathsf{P}}$ only a negligible advantage over the interactive counterpart PoSW. We start by describing the hybrid games, the last of which, almost corresponds to the security experiment in the interactive PoSW scheme.

$\underline{\mathsf{Exp}_{\tilde{\mathsf{P}},0}^{\tau(\cdot)}(1^\lambda, 1^N) \in \{0,1\}}$*:* Sample $\chi \leftarrow \{0,1\}^\lambda$, run $\pi \leftarrow \tilde{\mathsf{P}}^{\tau(\cdot)}(1^\lambda, 1^N, \chi)$, and observe its queries $Q_{\tau_\ell} \cup Q_{\tau_r}$ where $Q_{\tau_\ell} := \{(x,y) : y = \tau_\ell(x)\}$ and $Q_{\tau_r} := \{(x,y) : y = \tau_r(x)\}$ and τ_ℓ, τ_r are as in (2). Use Q_{τ_ℓ} to build the query graph $\mathsf{QG} := (V, E)$.

The query graph has as vertices the queries in Q_{τ_ℓ} and an edge is added between two vertices if and only if these two vertices have a corresponding edge in the skiplist graph $G_N := ([N]_0, E_N)$ and that the queries are consistent on that edge. Formally, let $V, E := \emptyset$, then we populate them as follows:

$$\text{For every } v_i := (x_i, y_i), v_j := (x_j, y_j) \in Q_{\tau_\ell},$$

add v_i, v_j to V and the edge (v_i, v_j) to E iff $x_i \prec x_j$ w.r.t. the skiplist graph G_N and the operator $\prec$ as in Def. 2. If $(x_j = (N, x'_j), y_j) \in V$ for some x'_j, then check whether $y_j = \phi_L$ and if not, remove v_j from V and all its incoming edges; note that $y_j = \phi_L$ implies that $x_j \prec \phi_L$.

If π is invalid output 0, otherwise let $\pi := (\phi_L, \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c})$ be a valid proof: $\forall i \in [t]$, parse $\mathcal{L}_{N,n-c}[i]$ as $z_i = (z_{i_1}, \ldots, z_{i_k})$ for some $k \in \mathbb{N}$. As π is a valid proof, it holds that $\mathsf{Consistent}(z_i) = 1$ and that $\tau_\ell(z_{i_k}) = \phi_L$. By definition of $\mathsf{Consistent}$, this means that

$$s_i := (z_{i_1}, \ldots, z_{i_k}, \phi_L) \ , \tag{6}$$

forms a τ_ℓ-sequence according to Def. 6.

Finally define the output of the experiment to be 1 if and only if π is valid and $\forall i \in [t]$, s_i is *extractable* from QG. Formally, we say that $s_i := (z_{i_1}, \ldots, z_{i_k}, \phi_L)$ is extractable from QG if

$$\forall j \in [k-1], (z_{i_j}, y_{i_j}) \in V \text{ and that } (z_{i_k}, \phi_L) \in V \ . \tag{7}$$

$\underline{\mathsf{Exp}_{\tilde{\mathsf{P}},1}^{\tau(\cdot)}(1^\lambda, 1^N) \in \{0,1\}}$*:* This experiment is identical to $\mathsf{Exp}_{\tilde{\mathsf{P}},0}^{\tau(\cdot)}(1^\lambda, 1^N)$ except that $\mathsf{Exp}_{\tilde{\mathsf{P}},1}^{\tau(\cdot)}(1^\lambda, 1^N)$ doesn't check the winning condition above that requires that $\forall i \in [t]$, s_i is extractable from QG, i.e., $\mathsf{Exp}_{\tilde{\mathsf{P}},1}^{\tau(\cdot)}(1^\lambda, 1^N) = 1$ iff π is valid.

Proposition 1. *Let q upper-bounds the total number of queries $\tilde{\mathsf{P}}^{\tau(\cdot)}$ makes to $\tau : \{0,1\} \to \{0,1\}^\lambda$, then*

$$\left| \Pr\left[\mathsf{Exp}_{\tilde{\mathsf{P}},0}^{\tau(\cdot)}(1^\lambda, 1^N) = 1\right] - \Pr\left[\mathsf{Exp}_{\tilde{\mathsf{P}},1}^{\tau(\cdot)}(1^\lambda, 1^N) = 1\right] \right| \le \frac{1}{2^\lambda} + \frac{q(q-1)}{2^{\lambda+1}} \ . \tag{8}$$

The proof of Proposition 1 boils down to either finding a collision under τ_ℓ from at most q queries, or that $\tilde{\mathsf{P}}$ can guess the output of τ_ℓ. The latter happens with probability $1/2^\lambda$ and the former with probability $q(q-1)/2^{\lambda+1}$. The formal proof is given in the full version of the paper.

Recall that $t = 2^c$ for some integer c and $N = 2^n$ and that $t \mid N$. We define the challenge sampling set $\mathcal{D}$ and the challenge re-sampling set $\mathcal{C}$ as follow. These two sets will define a series of games that the security proof will go through.

$$\mathcal{D} := \{(v,0) : \forall v \in [N] \text{ s.t. } t \mid v\}$$
$$\mathcal{C} := \{(v,1),\dots,(v,k) : \forall v \in [N] \text{ s.t. } v = 2^k \cdot h \cdot t \text{ for odd } h \text{ and } k \geq 1\}$$

Examples of such (ordered) sets would be

$$\mathcal{D} = \{(t,0),(2t,0),(3t,0),(4t,0),(5t,0),\dots\}$$
$$\mathcal{C} = \{(2t,1),(4t,1),(4t,2),(6t,1),\dots,(N,1),\dots,(N,n-c-1),(N,n-c)\}$$

For $\beta := (v,i) \in \mathcal{D} \cup \mathcal{C}$, the prover algorithm from Sect. 5.3 implicitly defines a set of associated t challenges, call it $\mathsf{chal}(\beta)$, and computes for each such set, the corresponding set of labeled paths, denoted as $\mathcal{L}_\beta$. This is formally described in (3) and (4), however, for readability's sake, we elaborate upon it below. For $\beta := (v,0) \in \mathcal{C}$, the prover algorithm defines $\mathsf{chal}(\beta) := \{v, v-1, \dots, v-t+1\}$ and no resampling is needed. However, for $\beta := (v,i) \in \mathcal{C}$, i.e., $i \geq 1$, $\mathsf{chal}(\beta)$ is resampled from $\mathsf{chal}(\beta_0)$ and $\mathsf{chal}(\beta_1)$ where by construction [7] $\beta_0 := (v, i-1)$ and $\beta_1 := (u, i-1)$. The resampling is according to the hypergeometric distribution as done by Sample.

For $\beta \in \mathcal{D} \cup \mathcal{C}$, we define functions $\delta, \gamma, \eta : \mathcal{D} \cup \mathcal{C} \to [0,1]$, event bad_β, security experiment $\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},\beta}$, and $\mathsf{neighbor} : \mathcal{D} \cup \mathcal{C} \to \mathcal{D} \cup \mathcal{C}$.

- $\gamma(\beta)$: the fraction of inconsistent nodes among all possible nodes that could have been included into $\mathsf{chal}(\beta)$. (Inconsistent in the sense that if $v_i \in \mathsf{chal}(\beta)$ and its corresponding path in $\mathcal{L}_\beta$ is $\mathsf{Path}^*_{v_i}$, then $\mathsf{Consistent}(\mathsf{Path}^*_{v_i}) = 0$.) For example, for $\beta = (N, n-c)$, it holds that $\gamma(\beta)$ equals the number of all inconsistent nodes among $[N]$ divided by N.
- $\delta(\beta)$: the faction of inconsistent nodes in $\mathsf{chal}(\beta)$. We will be interested in analyzing how close $\delta(\beta)$ is to $\gamma(\beta)$.
- For α from Theorem 2, define
$$\eta(\beta) := \frac{i}{n-c} \cdot (1-\alpha) \ . \tag{9}$$
- Event bad_β is defined whenever a re/sampling takes place, i.e., when $\tau_r(\beta,\cdot)$ is called
$$\mathsf{bad}_\beta := 1 \ \Leftrightarrow \ \delta(\beta) < \gamma(\beta) - \eta(\beta) \ . \tag{10}$$
- $\mathsf{neighbor}(\beta) := \beta'$: Let $\mathcal{C}^* \subseteq \mathcal{C}$ be the (ordered) set on whose elements $\tilde{\mathsf{P}}$ issued resampling queries, i.e., $\tau_r(\beta,\cdot)$ is a resampling query on $\beta \in \mathcal{C}^*$. If β is the first such element in $\mathcal{C}^*$, then set $\beta' = 1$, and otherwise β' is defined to be the previous element in $\mathcal{C}^*$.

[7] See Sect. 5.3 and/or Fig. 1.

– $\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},\beta}(1^\lambda, 1^N) \in \{0,1\}$: this is identical to its neighboring $\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},\beta'}(1^\lambda, 1^N)$ except it outputs 0 if $\mathsf{bad}_\beta = 1$.

Proposition 2. *For every $\beta \in \mathcal{D} \cup \mathcal{C}$ and $\beta' := \mathsf{neighbor}(\beta)$, the following holds*

$$\left|\Pr\left[\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},\beta}(1^\lambda, 1^N) = 1\right] - \Pr\left[\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},\beta'}(1^\lambda, 1^N) = 1\right]\right| \leq e^{-2t\cdot\left(\frac{1-\alpha}{n-c}\right)^2} \ . \qquad (11)$$

Before proving Proposition 2, we make a few observations. For the first $\beta \in \mathcal{C}^*$, it holds that $\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},\beta'} = \mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},1}$, and the last experiment corresponds to $\beta = (N, n-c) \in \mathcal{C}$. As the total number of such experiments is at most q, and the distance between each neighboring experiments is bounded by Proposition 2, it then holds by a simple union bound that

$$\left|\Pr\left[\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},1}(1^\lambda, 1^N) = 1\right] - \Pr\left[\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},(N,n-c)}(1^\lambda, 1^N) = 1\right]\right| \leq q \cdot e^{-2t\cdot\left(\frac{1-\alpha}{n-c}\right)^2} \ . \qquad (12)$$

Observe that for the final game with $\beta = (N, n-c)$ corresponds to the sink vertex G_N. If $\mathsf{bad}_\beta = 0$, then the soundness analysis is similar to the analysis of the interactive PoSW given in [2], and recalled in Sect. 5.3. More concretely, by definition it follows that $\eta(\beta) = (1-\alpha)$. Now if $\tilde{\mathsf{P}}$ made Q_{τ_ℓ} queries of sequential weight $\Omega_{seq}(Q_{\tau_\ell}) < \alpha$, then by Lemma 1, except with probability $\epsilon_{\tau_\ell} := 1/2^\lambda + q^2/2^\lambda$, this must mean that $\gamma(\beta) > (1-\alpha)$, which then implies that $\delta(\beta) > 0$, which implies, that V rejects the proof, as the proof will contain at least one inconsistent path. Therefore,

$$\Pr\left[\mathsf{Exp}^{\tau(\cdot)}_{\tilde{\mathsf{P}},(N,n-c)}(1^\lambda, 1^N) = 1\right] \leq \epsilon_{\tau_\ell} := \frac{1}{2^\lambda} + \frac{q^2}{2^\lambda} \ . \qquad (13)$$

Before bounding the final probability, we observe that in the probability ϵ_{τ_ℓ} above, $1/2^\lambda$ accounts for guessing the output of τ_ℓ. And as we accounted for such event in analyzing Proposition 1, we conclude that the probability ϵ in Theorem 2 can be upper bounded by summing the probability in (8), (11), (12), (13) and subtracting $1/2^\lambda$. This concludes the proof. □

Proof of Proposition 2. The proof makes use of the following lemma, which is simply a restatement from [7].

Lemma 5. *Let t be an integer, $b \in \{0,1\}$, $\delta_b \in [0,1]$, and sets U_b s.t. $|U_b| = t$, $U_0 \cap U_1 = \emptyset$ and U_b contains elements that are either* consistent *or* inconsistent *with δ_b being the fraction of inconsistent elements in U_b, and*

$$\delta_b \geq \gamma_b - \eta_b \ , \qquad (14)$$

for some global values $\gamma_b, \eta_b \in [0,1]$.

We sample from $U_0 \cup U_1$ without replacement in t draws a set U, with $|U| = t$, and let X be the random variable indicating the number *of inconsistent elements in U. Then an arbitrary $\eta \in [0,1]$, we have*

$$\Pr\left[X \leq t \cdot \left(\frac{\gamma_0 + \gamma_1}{2} - \eta\right)\right] \leq e^{-2t\cdot\left(\eta - \frac{\eta_0+\eta_1}{2}\right)^2} \ . \qquad (15)$$

The proof of Lemma 5 uses a Hoeffding bound and is given in the full version of the paper.

Proof (of Proposition 2). The proof amounts to bounding the probability of bad_β. We have two cases: If $\beta = (v, 0) \in \mathcal{D}$, then by definition, $\mathsf{bad}_\beta = 0$ as $\eta(\beta) = 0$ and $\delta(\beta) = \gamma(\beta)$. If $\beta = (v, i) \in \mathcal{C}$, then we resample t elements according to Sample from $\mathsf{chal}(\beta_0) \cup \mathsf{chal}(\beta_1)$ for some $\beta_0 = (v, i-1)$ and $\beta_1 = (u, i-1)$ for which $\mathsf{bad}_{\beta_0} = \mathsf{bad}_{\beta_1} = 0$, for otherwise we would not be in this game. Then we have that $\gamma_b := \gamma(\beta_b), \delta_b := \delta(\beta_b), \eta_b := \eta(\beta_b)$ satisfy

$$\delta_b \geq \gamma_b - \eta_b = \gamma_b - \frac{i-1}{n-c} \cdot (1-\alpha) \ . \tag{16}$$

Now applying Lemma 5 on $U_b := \mathsf{chal}(\beta_b)$, $\eta := \eta(\beta)$ and noting that $\gamma := \gamma(\beta) := (\gamma_0 + \gamma_1)/2$, we get

$$\Pr[\mathsf{bad}_\beta = 1] = \Pr[t \cdot \delta(\beta) < t \cdot (\gamma - \eta)] \leq e^{-2t \cdot \left(\frac{i \cdot (1-\alpha)}{n-c} - \frac{(i-1)\cdot(1-\alpha)}{n-c}\right)^2} = e^{-2t \cdot \left(\frac{1-\alpha}{n-c}\right)^2} .$$

□

6 Incremental PoSW for General Distributions

In this section, we give our skiplist-based iPoSW for general weight distributions. We characterize these distributions as t-incrementally sampleable distributions. The construction is similar to the construction of Sect. 5, except that, we devise and use a new on-the-fly sampling technique, which samples according to the Poisson Binomial distribution. The need for the new sampling technique is motivated by applying Hoeffding-like tail bounds when sampling from general distributions, rather than sampling without replacement uniformly at random as is the case for the construction from Sect. 5 and that of [7]. The new sampling technique introduces a small modification to the main construction: the prover now needs to additionally give out, as part of its proof, subsets that are used for the resampling algorithm. These are needed for the verifier to validate the consistency of the resampling. Moreover, the verifier will have to check that number of nodes (and their weight) of such subsets is within appropriate intervals centered around their expectation.

Our on-the-fly sampling technique. Let $\Omega := \{\Omega_{2^i \cdot t} : [2^i \cdot t] \to [0,1]\}_{i \geq 0}$ be a family of (weight) distributions, $U_0 \subseteq [2^i \cdot t]$ and $U_1 \subseteq [2^i \cdot t + 1 : 2^{i+1} \cdot t]$ sets sampled according to $t \cdot \Omega_{2^i \cdot t}$.(Technically, as the domains of $\Omega_{2^i \cdot t}$ and U_1 mismatch, we think of the $2^i \cdot t$-shifted U_1 as being sampled from $t \cdot \Omega_{2^i \cdot t}$.)

The goal of the on-the-fly sampling is to sample W from $U := U_0 \cup U_1$ such that W is distributed according to $t \cdot \Omega_{2^{i+1} \cdot t}$. That is, instead of directly sampling W from $[2^{i+1} \cdot t]$ according to $t \cdot \Omega_{2^{i+1} \cdot t}$, the on-the-fly sampling allows one to first sample each U_0 and U_1 individually according to $t \cdot \Omega_{2^i \cdot t}$, and then sample W from U. For the sampling of W from U to be possible, it necessary that every $u \in U$ is at least as probable in $\Omega_{2^i \cdot t}$ as in $\Omega_{2^{i+1} \cdot t}$. This is reflected in conditions

2 and 3 in Def. 8. Furthermore, technically, it must be that $t \cdot \Omega_{2^i \cdot t} \leq 1$ for every i. This is implied by conditions 1, 2 and 3 in Def. 8.

The t factor in the sampling process above is stipulated by the security of iPoSW schemes, which requires the prover to open t challenges. This corresponds to requiring that W has t samples. Our technique ensures this on expectation.

GSample (Fig. 4) formalizes how W is sampled from U. Note in Fig. 4, W consists of indices of elements from U, rather than the actual elements.

6.1 Incrementally Sampleable Distributions

We characterize weight distributions that can be on-the-fly sampled. These are t-incrementally sampleable (weight) distributions.

Definition 8. *For $t \in \mathbb{N}^+$ and a family of weight functions $\Omega := \{\Omega_{2^i \cdot t} : [2^i \cdot t] \to [0,1]\}_{i \geq 0}$, we say Ω is t-incrementally sampleable if the following hold:*

1. $\forall j \in [t] : t \cdot \Omega_t(j) \leq 1$
2. $\forall i \geq 0, \forall j \in [2^i \cdot t] : \Omega_{2^{i+1} \cdot t}(j) \leq \Omega_{2^i \cdot t}(j)$
3. $\forall i \geq 0, \forall j \in [2^i \cdot t + 1 : 2^{i+1} \cdot t] : \Omega_{2^{i+1} \cdot t}(j) \leq \Omega_{2^i \cdot t}(j - 2^i \cdot t)$

It is immediate to see that the uniform distribution on $[N]$ is t-incrementally sampleable for any $t \leq N$. Another distribution that is of particular interest to the application of our incremental PoSW to incremental SNACKs is the SNACK distribution of [2]. For a fixed positive integer ℓ and every positive integer m, the SNACK distribution $\Omega_m : [m] \to [0,1]$ is defined as

$$\Omega_m(j) = S_m \cdot \frac{1}{m + \ell - j} \quad \text{where} \quad S_m := \left(\sum_{j=1}^{m} \frac{1}{m + \ell - j} \right)^{-1} . \tag{17}$$

Lemma 6. *The SNACK distribution $\Omega := \{\Omega_{2^i \cdot t}\}_{i \geq 0}$ where $\Omega_{2^i \cdot t}$ is as in* (17) *is t-incrementally sampleable for $\ell \geq t \cdot S_t$.*

The proof is elementary and is given in the full version of the paper.

6.2 Scheme Description

Our iPoSW scheme for t-sampleable distributions is in spirit similar to the iPoSW scheme from Sect.5. The main difference is that we employ our new on-the-fly sampling technique. This has correctness and security consequences that we address.

Recall that the on-the-fly sampling technique from Sect. 5 always produces W of size exactly t from intermediate sets U_0 and U_1 which are implicitly defined. This allows the verifier V to check the correctness of W. In contrast, our on-the-fly sampling produces W whose size is t on expectation, and more critically, the intermediate sets U_0 and U_1 are not implicitly defined, and hence, the prover P has to give U_0 and U_1 for V to verify the correctness of W. This results in an increase in the proof size: for each challenge v, P gives $\log N$ pairs of sets with total expected size $2 \cdot t \cdot \log^2 N$. The increased proof size is still succinct though.

This could potentially allow a malicious prover $\tilde{\mathsf{P}}$ to manipulate the sets U_0 and U_1 such that the challenges in W don't include challenges in $U_0 \cup U_1$ that the adversary can't correctly answer. To get around this issue, P commits to the sampling sets across the execution of the protocol. This ensures that the sets in different challenges are fixed and consistent with each other. However, this doesn't rule out that $\tilde{\mathsf{P}}$ would not gain any advantage by committing to manipulated sets. We address this issue in the security proof by showing that except with a negligible probability, $\tilde{\mathsf{P}}$ gains no advantage in cheating on the sampling sets.

We give the formal construction with similar parameters as in Sect. 5.1 and the following additional parameters: $t_{\mathsf{min}} := (1-\zeta) \cdot t$ and $t_{\mathsf{max}} := (1+\zeta) \cdot t$ for $\zeta \in (0,1)$, $\omega_{\mathsf{min},d,\Omega} \in (0,1)$ and $\omega_{\mathsf{max},d,\Omega} \in (0,1)$ for $d \in \mathbb{N}$.

<u>$\mathsf{P}^{\tau(\cdot)}(1^\lambda, 1^N, N, \Omega)$:</u>

1. Traverse the graph $G_n = (V = [N]_0, E)$ in topological order, starting from 0. At every node $v \in [N]_0$ which is traversed, do the following:
   (a) Compute $L(v)$ according to (1).
   (b) If $t \mid v$ and $v \in [N]$, write $v = 2^k \cdot h \cdot t$, with h odd and $k \in \mathbb{N}_0$. For $j \in [t]$:

$$\mathcal{L}_{v,0}[j] := \mathsf{Path}^*(v-t, v-t+j, v), \qquad \mathcal{I}_{v,0}[j] := j \tag{18}$$
$$U_{v,0}[j] := v-t+j, \qquad \mathcal{U}_{v,0}[j] := \bot$$

   If $k \geq 1$, do the following for $i \in [k]$:
   i. Compute $u := v - 2^{i-1} \cdot t$
   ii. Compute $h_0 := |U_{u,i-1}|$, $h_1 := |U_{v,i-1}|$
   iii. Compute $r_{v,i} := \tau_r(v, i, L(v), U_{u,i-1}, U_{v,i-1})$.
   iv. Choose a subset $S_{v,i}$ from $[h_0 + h_1]$ as $S_{v,i} := \mathsf{GSample}(i, \Omega, t, U_{u,i-1}, U_{v,i-1}; r_{v,i})$ with $\mathsf{GSample}$ in Fig. 4.
   v. For $j \in [|S_{v,i}|]$, do the following

$$a := \begin{cases} 0 & \text{if } S_{v,i}[j] \leq h_0 \\ 1 & \text{if } S_{v,i}[j] > h_0 \end{cases} \qquad \text{and} \qquad b := S_{v,i}[j] - h_0 \cdot a$$

$$\mathcal{L}_{v,i}[j] := \begin{cases} (\mathcal{L}_{u,i-1}[b], (v, L(\mathsf{Parents}(v)))) & \text{if } a = 0 \\ ((u, L(\mathsf{Parents}(u))), \mathcal{L}_{v,i-1}[b]) & \text{if } a = 1 \end{cases} \tag{19}$$

$$\mathcal{I}_{v,i}[j] := \begin{cases} \mathcal{I}_{u,i-1}[b], j & \text{if } a = 0 \\ \mathcal{I}_{v,i-1}[b], j & \text{if } a = 1 \end{cases}$$

$$\mathcal{U}_{v,i}[j] := \begin{cases} \mathcal{U}_{u,i-1}[b], U_{u,i-1}, U_{v,i-1} & \text{if } a = 0 \\ \mathcal{U}_{v,i-1}[b], U_{u,i-1}, U_{v,i-1} & \text{if } a = 1 \end{cases}$$

$$U_{v,i}[j] := \begin{cases} U_{u,i-1}[b] & \text{if } a = 0 \\ U_{v,i-1}[b] & \text{if } a = 1 \end{cases}$$

```
        vi. Store $\mathcal{L}_{v,i}, \mathcal{I}_{v,i}, \mathcal{U}_{v,i}$, and $U_{v,i}$ in memory. Note that by design, $\mathcal{L}_{v,i}[j]$
            satisfies $\mathsf{Consistent}(\mathcal{L}_{v,i}[j]) = 1$.
       vii. For $x \in \{u, v\}$, erase from memory $\mathcal{L}_{x,i-1}, \mathcal{I}_{x,i-1}, \mathcal{U}_{x,i-1}$, and $U_{x,i-1}$.
2. Terminate and output $\pi := (\phi_L := L(N), \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c}, \mathcal{U}_{N,n-c}, U_{N,n-c})$.
```

$\underline{\mathsf{Inc}^{\tau(\cdot)}(1^\lambda, 1^{N'}, N, \pi, \chi)}$:

```
1. Parse $\pi$ as $(\phi_L, \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c}, \mathcal{U}_{N,n-c}, U_{N,n-c})$.
2. Compute $N + N' = N''$ and check that $N'' = 2^{n''}$ for some $n'' \in \mathbb{N}$,
3. Execute the algorithm $\mathsf{P}^{\tau(\cdot)}(1^\lambda, 1^{N''}\chi)$ starting from step 2 with a slight
   change: traverse the graph $G_{2^{n''}}$ starting from $N + 1$.
```

$\underline{\mathsf{V}^{\tau(\cdot)}(1^\lambda, 1^N, \pi, \chi)}$:

```
1. Parse $\pi$ as $(\phi_L, \mathcal{L}_{N,n-c}, \mathcal{I}_{N,n-c}, \mathcal{U}_{N,n-c}, U_{N,n-c})$.
2. If $|\mathcal{L}_{N,n-c}| \neq |U_{N,n-c}|$, return 0.
3. Let $s := |\mathcal{L}_{N,n-c}|$. For all $i \in [s]$ :
   (a) Parse $\mathcal{L}_{N,n-c}[i]$ as $y_i = (y_{i_1}, \ldots, y_{i_k})$, for some $k \in \mathbb{N}$.
   (b) $b_i := 1$ iff the following hold
         i. $\tau_\ell(y_{i_k}) = \phi_L$
        ii. $\mathsf{Consistent}(y_i) = 1$ where $\mathsf{Consistent}$ is as in Def. 4,
       iii. $\mathsf{GCheck}(\mathcal{L}_{N,n-c}[i], \mathcal{I}_{N,n-c}[i], \mathcal{U}_{N,n-c}[i], U_{N,n-c}, 0, N, n-c) = 1$, where
            the algorithm $\mathsf{GCheck}$ is described in Fig. 5, and
4. Return $\bigwedge_{i=1}^{s} b_i$
```

```
Algorithm GSample: On input $(i, \Omega, t, U_0, U_1; r)$
 1. Define a normalization factor:
```

$$\psi(i,j) := \begin{cases} 1/t & \text{if } i = 1 \\ \Omega_{2^{i-1}\cdot t}(j) & \text{if } i > 1 \text{ and } j \in [2^{i-1} \cdot t] \\ \Omega_{2^{i-1}\cdot t}(j - 2^{i-1} \cdot t) & \text{if } i > 1 \text{ and } j \in [2^{i-1} \cdot t + 1 : 2^i \cdot t] \end{cases} \quad (20)$$

```
 2. Initialize $S := \emptyset$ and $\forall j \in [|U_0| + |U_1|]$, do:
    (a) If $j \leq |U_0|$, let $v := U_0[j]$, else $v := U_1[j]$
    (b) Use $r$ to toss a biased coin that returns 1 w.p. $p := \Omega_{2^i \cdot t}(v)/\psi(i, v)$ and 0 w.p.
        $1 - p$.
    (c) $S := S \cup \{j\}$ iff the coin outcome is 1
 3. Return $S$
```

Fig. 4. Algorithm GSample

6.3 Theorem Statement and Proof Outline

The main difference in the constructions of Sect. 5 and 6 is that we apply the new on-the-fly sampling technique as employed in GSample in Fig. 4 to two sets

```
Algorithm GCheck

On input a path Path, a list of indices ind, a list of lists of nodes $\mathcal{U}$, a list of nodes
$U$, a starting node s, an ending node e, and a recursion index d:
 1. Parse Path $:= (y_1, \dots, y_k)$ and ind $:= (i_0, i_1, \dots, i_\mathsf{d}) \in \mathbb{N}^{\mathsf{d}+1}$
 2. Parse $\mathcal{U} := (U_{0,0}, U_{1,0}, U_{0,1}, U_{1,1}, \dots, U_{0,d-1}, U_{1,d-1})$
 3. Compute $r_{\mathsf{e,d}} := \tau_r(\mathsf{e}, \mathsf{d}, \tau_\ell(y_k), U_{0,d-1}, U_{1,d-1})$ and
    $S_{\mathsf{e,d}} := \mathsf{GSample}(d, \Omega, t, U_{0,d-1}, U_{1,d-1}; r_{\mathsf{e,d}})$
 4. Let $h_0 = |U_{0,d-1}|$ and $h_1 = |U_{1,d-1}|$
 5. Let $w_0 = \Omega_{2^{d-1} \cdot t}(U_{0,d-1})$ and $w_1 = \Omega_{2^{d-1} \cdot t}(U_{1,d-1})$
 6. Let $U_{0,d-1}||U_{1,d-1}$ be the list of nodes obtained by concatenating lists $U_{0,d-1}$
    and $U_{1,d-1}$. Let $U^* := \{(U_{0,d-1}||U_{1,d-1})[s] : s \in S_{\mathsf{e,d}}\}$. If $U \neq U^*$, return 0
 7. Write
    $a := \begin{cases} 0 & \text{if } S_{\mathsf{e,d}}[i_\mathsf{d}] \leq h_0 \\ 1 & \text{if } S_{\mathsf{e,d}}[i_\mathsf{d}] > h_0 \end{cases}$   and   $b := S_{\mathsf{e,d}}[i_\mathsf{d}] - h_0 \cdot a.$
 8. Define $\mathsf{m} := (\mathsf{e} - \mathsf{s})/2$
 9. If $\mathsf{d} = 1$, do the following:
  – Compute the challenge $c$ corresponding to Path
  – Compute $j$ such that $c \in [j \cdot t + 1 : (j+1) \cdot t]$
  – If $a = 0$, return 1 iff $\mathsf{s} + b = c$ and
    $U_{0,0} = [j \cdot t + 1 : (j+1) \cdot t]$   and   $U_{1,0} = [(j+1) \cdot t + 1 : (j+2) \cdot t]$
  – If $a = 1$, return 1 iff $\mathsf{s} + \mathsf{m} + b = c$ and
    $U_{0,0} = [(j-1) \cdot t + 1 : j \cdot t]$   and   $U_{1,0} = [j \cdot t + 1 : (j+1) \cdot t]$
10. If $\mathsf{d} > 1$, do the following
  – If $a = 0$, $S_{\mathsf{e,d}}[i_\mathsf{d}] = i_{\mathsf{d}-1}$,
    $(1-\zeta) \cdot t \leq h_0, h_1 \leq (1+\zeta) \cdot t$   and   $\omega_{\mathsf{min},d-1,\Omega} \leq w_0, w_1 \leq \omega_{\mathsf{max},d-1,\Omega}$
    $\mathsf{GCheck}((y_1, \dots, y_{k-1}), (i_1, \dots, i_{\mathsf{d}-1}), (U_{0,0}, U_{1,0}, \dots, U_{0,d-2}, U_{1,d-2}), U_{0,d-1}, \mathsf{s}, \mathsf{s}+$
    $\mathsf{m}, \mathsf{d}-1)$
  – If $a = 1$, $S_{\mathsf{e,d}}[i_\mathsf{d}] = h_0 + i_{\mathsf{d}-1}$,
    $(1-\zeta) \cdot t \leq h_0, h_1 \leq (1+\zeta) \cdot t$   and   $\omega_{\mathsf{min},d-1,\Omega} \leq w_0, w_1 \leq \omega_{\mathsf{max},d-1,\Omega}$
    $\mathsf{GCheck}((y_2, \dots, y_k), (i_1, \dots, i_{\mathsf{d}-1}), (U_{0,0}, U_{1,0}, \dots, U_{0,d-2}, U_{1,d-2}), U_{1,d-1}, \mathsf{s}, \mathsf{s}\ +$
    $\mathsf{m}, \mathsf{d}-1)$
  – Return 0
```

Fig. 5. Description of the GCheck algorithm.

U_0 and U_1 to obtain W. A second difference is that, while in Sect. 5, the sampling sets U_0, U_1 are implicitly defined, and hence are assumed to be given to the verifier V, they are explicitly provided by the prover P in the main construction. Modulo these differences, the two constructions are essentially identical. In Theorem 3 below, the bound $q \cdot \epsilon_0$ comes from analyzing our on-the-fly sampling, and the bound $q \cdot \epsilon_1$ comes form the analysis of a new scenario where a malicious prover commits to sampling sets that are maliciously chosen.

Theorem 3. *Let $\Pi := (\mathsf{P}^{\tau(\cdot)}, \mathsf{V}^{\tau(\cdot)}, \mathsf{Inc}^{\tau(\cdot)})$ be as in Sect. 6 and q be an upper bound on the total number of queries to $\tau(\cdot)$, then Π is an (α, ϵ)-soundness incremental PoSW scheme, as per Def. 7, with any $\alpha \in (0,1]$ and*

$$\epsilon = \frac{1+q^2}{2^\lambda} + \frac{q(q-1)}{2^{\lambda+1}} + q \cdot (\epsilon_0 + \epsilon_1) \qquad (21)$$

where $\epsilon_0, \epsilon_1 \in (0,1)$ depend on the underlying t-incrementally sampleable weight distribution Ω. For uniform Ω, we have

$$\epsilon_0 \leq e^{-\frac{t\cdot(1-\alpha)^2\cdot\zeta^{2(n-c)+4}}{(n-c)^2\cdot(2-\zeta)^{2(n-c)+3}}} \quad and \quad \epsilon_1 \leq 2^{-\zeta\cdot t} \ .$$

In the full version, we give concrete ϵ_0, ϵ_1 for any t-incrementally sampleable Ω, as well as concrete upper bounds for the SNACK distribution.

Acknowledgements. Parts of this work were done while the first author was at TU Wien and was supported by the Vienna Science and Technology Fund (WWTF)[10.47379/VRG18002]. The first author was also partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program under project PICOCRYPT (grant agreement No. 101001283), by the Spanish Government under project PRODIGY (TED2021-132464B-I00), and by the Madrid Regional Government under project BLOQUES (S2018/TCS-4339). The last two projects are co-funded by European Union EIE, and NextGenerationEU/PRTR funds.

The research of the second author was in part funded by the Austrian Science Fund (FWF) and netidee SCIENCE grant P31621-N38 (PROFET).

References

1. Abusalah, H., Alwen, J., Cohen, B., Khilko, D., Pietrzak, K., Reyzin, L.: Beyond hellman's time-memory trade-offs with applications to proofs of space. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part II. LNCS, vol. 10625, pp. 357–379. Springer, Heidelberg (Dec 2017). https://doi.org/10.1007/978-3-319-70697-9_13
2. Abusalah, H., Fuchsbauer, G., Gazi, P., Klein, K.: Snacks: Leveraging proofs of sequential work for blockchain light clients. In: Agrawal, S., Lin, D. (eds.) Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13791, pp. 806–836. Springer (2022). https://doi.org/10.1007/978-3-031-22963-3_27, https://doi.org/10.1007/978-3-031-22963-3_27
3. Abusalah, H., Kamath, C., Klein, K., Pietrzak, K., Walter, M.: Reversible proofs of sequential work. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 277–291. Springer, Heidelberg (May 2019). https://doi.org/10.1007/978-3-030-17656-3_10
4. 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, Heidelberg (Aug 2018). https://doi.org/10.1007/978-3-319-96884-1_25
5. 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, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78375-8_15
6. Cohen, B., Pietrzak, K.: The Chia Network blockchain (July, 2019), https://www.chia.net/assets/ChiaGreenPaper.pdf

7. Döttling, N., Lai, R.W.F., Malavolta, G.: Incremental proofs of sequential work. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 292–323. Springer, Heidelberg (May 2019). `https://doi.org/10.1007/978-3-030-17656-3_11`
8. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) CRYPTO'92. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (Aug 1993). `https://doi.org/10.1007/3-540-48071-4_10`
9. Dziembowski, S., Faust, S., Kolmogorov, V., Pietrzak, K.: Proofs of space. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part II. LNCS, vol. 9216, pp. 585–605. Springer, Heidelberg (Aug 2015). `https://doi.org/10.1007/978-3-662-48000-7_29`
10. Ephraim, N., Freitag, C., Komargodski, I., Pass, R.: Continuous verifiable delay functions. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020, Part III. LNCS, vol. 12107, pp. 125–154. Springer, Heidelberg (May 2020). `https://doi.org/10.1007/978-3-030-45727-3_5`
11. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO'86. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (Aug 1987). `https://doi.org/10.1007/3-540-47721-7_12`
12. Mahmoody, M., Moran, T., Vadhan, S.P.: Publicly verifiable proofs of sequential work. In: Kleinberg, R.D. (ed.) ITCS 2013. pp. 373–388. ACM (Jan 2013). `https://doi.org/10.1145/2422436.2422479`
13. Pietrzak, K.: Simple verifiable delay functions. In: Blum, A. (ed.) ITCS 2019. vol. 124, pp. 60:1–60:15. LIPIcs (Jan 2019). `https://doi.org/10.4230/LIPIcs.ITCS.2019.60`
14. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part III. LNCS, vol. 11478, pp. 379–407. Springer, Heidelberg (May 2019). `https://doi.org/10.1007/978-3-030-17659-4_13`