Probabilistically Checkable Arguments for all NP

Shany Ben-David [*]

April 17, 2024

Abstract

A probabilistically checkable argument (PCA) is a computational relaxation of PCPs, where soundness is guaranteed to hold only for false proofs generated by a computationally bounded adversary. The advantage of PCAs is that they are able to overcome the limitations of PCPs. A *succinct* PCA has a proof length that is polynomial in the witness length (and is independent of the non-deterministic verification time), which is impossible for PCPs, under standard complexity assumptions. Bronfman and Rothblum (ITCS 2022) constructed succinct PCAs for NC that are publicly-verifiable and have constant query complexity under the sub-exponential hardness of LWE.

We construct a publicly-verifiable succinct PCA with constant query complexity for all NP in the adaptive security setting. Our PCA scheme offers several improvements compared to the Bronfman and Rothblum construction: (1) it applies to all problems in NP, (2) it achieves adaptive security, and (3) it can be realized under any of the following assumptions: the *polynomial* hardness of LWE; $O(1)$-LIN on bilinear maps; or sub-exponential DDH.

Moreover, our PCA scheme has a *succinct prover*, which means that for any NP relation that can be verified in time T and space S, the proof can be generated in time $O_{\lambda,m}(T \cdot \text{polylog}(T))$ and space $O_{\lambda,m}(S \cdot \text{polylog}(T))$. Here, $O_{\lambda,m}$ accounts for polynomial factors in the security parameter and in the size of the witness. En route, we construct a new *complexity-preserving* RAM Delegation scheme that is used in our PCA construction and may be of independent interest.

Keywords: PCP; succinct arguments; instance compression; RAM delegation

[*]Department of Computer Science, Bar-Ilan University, Israel. Email: `shany.ben-david@biu.ac.il`. Sponsored in part by the Israel Science Foundation (Grant No. 2302/22), and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

Contents

1 Introduction

Probabilistically checkable proofs (PCPs) play a significant role in complexity theory and cryptography, leading to groundbreaking results in various fields. This is evidenced by the remarkable PCP theorem which is one of the most important results in theoretical computer science. The PCP theorem states that the satisfiability of a formula of size n can be proved in $\mathrm{poly}(n)$ time and can be verified by querying only a constant number of bits from the proof.

While PCPs are a very powerful tool, they have several limitations. For example, Fortnow and Santhanam [FS08] showed that PCPs cannot be *succinct* unless the polynomial hierarchy collapses. A PCP proof is succinct if, for a formula of size n with witness length m, the proof length is $\mathrm{poly}(m)$, rather than $\mathrm{poly}(n, m)$. Beyond showing a limit to the efficiency of PCPs, this fact has broader implications. Harnik and Naor [HN10] raised the *instance compression* (IC) question, in which we ask if it is possible to take any instance in any NP relation and compress it to a short instance while preserving the information of whether the instance is in the language or not. The main focus of this question is instances of size n with a witness of size m, where the instance is much larger than the witness, i.e., $m \ll n$. A line of work [HN10; FS08; Dru15; BDFH09] shows different use cases for instance compression, including constructing a variety of fundamental cryptographic primitives. Unfortunately, Fortnow and Santhanam showed that instance compression also implies succinct PCPs, which means that instance compression does not exist (unless the polynomial hierarchy collapses).

Probabilistically checkable arguments (PCA). The notion of PCA was introduced as a *computational* analog of PCPs [KR09; Zim02; BR22]. That is, soundness is required to hold only against *computationally bounded* adversaries. This allows PCAs to bypass information-theoretic barriers that limit PCP constructions.

In more detail, PCAs rely on a (honestly generated) common reference string (CRS) which is given to both the prover and the verifier. The key parameters of interest in a PCA scheme include the size of the proof, the verifier's query complexity and randomness complexity, and the prover's running time. Unlike typical cryptographic primitives, PCAs aim for *constant* soundness error against polynomially bounded adversaries. This is inherent since PCAs additionally aim for *constant* query complexity (independent of the security parameter). Note that the soundness error is defined over the randomness of generating the CRS, the adversary, and the verifier.

Succinct PCAs. A PCA for an NP relation $\mathcal{R}$ with verification time $t = t(n, m)$ is said to be *succinct* if the PCA proof is of length $\mathrm{poly}(\lambda, m, \log t)$, where n is the instance size, m is the witness size, and λ is the security parameter. Importantly, poly refers to a fixed universal polynomial (that does not depend on the relation $\mathcal{R}$).

Previous work on PCAs. Kalai and Raz [KR09] constructed a *privately-verifiable* PCA. In the privately-verifiable setting, the verifier must hold a trapdoor to the common reference string in order to verify the proof. Their PCA construction relies on exponential hardness assumptions for PIR schemes, providing *non-adaptive* soundness, ensuring security for instances that are chosen independently of the common reference string. Their construction gives $\mathrm{poly}(d, m)$ proof size and $\mathrm{polylog}(n)$ query complexity for languages that can be verified with circuits of size n and depth d, using witness of size m.

Later on, Bronfman and Rothblum [BR22] were the first to construct a *publicly-verifiable succinct* PCA with constant query complexity, also with *non-adaptive* soundness. Their results were a big step forward. However, they have two significant drawbacks. First, their PCA construction applies only to relations in NC rather than to the entire NP class. Second, their PCA construction is under the sub-exponential hardness assumption of the learning with error (LWE) problem.

Our PCA. We construct a publicly-verifiable succinct PCA with constant query complexity in the *adaptive* security setting for *all* NP. Compared to the Bronfman and Rothblum construction, our PCA scheme exhibits improvements in several aspects: (1) it applies to all problems in NP, (2) it achieves adaptive security, (3) it can be realized under any of the following assumptions: the *polynomial* hardness of LWE; $O(1)$-LIN on bilinear maps[1]; or sub-exponential Decisional Diffie-Hellman (DDH), and (4) an efficient prover, which we denote as a *succinct prover*.

PCA with a succinct prover. We say that a PCA scheme has a *succinct prover* if, for any NP relation $\mathcal{R}$ that can be verified in time $t = \text{poly}(n, m)$ and space $s = \text{poly}(n, m)$, the PCA prover (given the instance and the witness) generates the proof in time $t \cdot \text{poly}(\lambda, m, \log t)$ and space $s \cdot \text{poly}(\lambda, m, \log t)$. Here, poly refers to a fixed universal polynomial (that does not depend on the relation $\mathcal{R}$).[2]

Theorem 1.1 (Informal)**.** *Every* NP *relation has a publicly-verifiable succinct* PCA *with constant query complexity under any one of the following assumptions: (1)* polynomial *hardness of* LWE*; (2)* $O(1)$-LIN*; or (3) sub-exponential Decisional Diffie-Hellman (*DDH*).*

Moreover, the PCA *protocol is adaptively sound and has a succinct prover.*

In fact, our PCA construction (along with the argument presented in Theorem 1.2) relies on two general components, non-interactive batch argument for NP and a rate-1 OT.[3] Both components can be constructed under any of the assumptions mentioned in the theorem, which enables us to obtain our results.

Computational instance compression (CIC). An instance compression (IC) is a very strong and useful tool. Unfortunately, it cannot be constructed under standard assumptions. Bronfman and Rothblum [BR22] solved this problem by introducing *computational instance compression* or CIC, which is the cryptographic equivalent for instance compression (IC). For a false statement, the new instance computed by the scheme might be in the language, but it is computationally infeasible to find a witness for the new instance. Bronfman and Rothblum show that PCA implies CIC. Their CIC is for the class NC and can be realized under the sub-exponential hardness assumption of the learning with error (LWE) problem. We combine their CIC construction with Theorem 1.1 and immediately get CIC for *all* NP, under the same hardness assumptions as for our PCA.

Theorem 1.2 (Informal)**.** *Every* NP *relation has a* CIC *scheme under any one of the following assumptions: (1)* polynomial *hardness of* LWE*; (2)* $O(1)$-LIN*; or (3) sub-exponential Decisional Diffie-Hellman (*DDH*).*

[1] Our PCA can be realized under the k-LIN assumption on bilinear maps for any arbitrary constant $k \geq 1$. We will refer to this assumption as $O(1)$-LIN assumption.

[2] Recall that a succinct PCA has proof size $\text{poly}(\lambda, m, \log t)$. In our construction, the prover's time and space has a multiplicative factor in that proof size. It remains open to achieve an additive factor, i.e, $O(t) + \text{poly}(\lambda, m, \log t)$ running time and $O(s) + \text{poly}(\lambda, m, \log t)$ space.

[3] Both components are formally defined in [KLVW22].

Our main building block in our PCA construction (which in turn, leads to our results in Theorem 1.1 and Theorem 1.2) is a RAM Delegation scheme. However, existing RAM Delegation schemes in the literature do not meet our specific requirements. This leads us to construct a new RAM Delegation scheme.[4]

RAM delegation. Efficient verification of computation is a fundamental notion in computer science both in theory and, recently, has been deployed in practice in cloud services and blockchains. This usage makes verification of computation schemes highly motivated in theory and practice.

In this paper, the computation we wish to delegate is described as a RAM machine. In a RAM Delegation scheme, a verifier wishes to evaluate the output of a RAM machine M on an input x without investing the computational resources required for the computation. Instead, the verifier delegates the computation to an untrusted prover, which generates the output of the computation $y = M(x)$ together with a proof Π that supports the correctness of the computation (given a suitable common reference string crs). We would like to minimize the computation time for both the prover and the verifier. The prover should run in time proportional to the original computation (polynomial or even linear in the actual computation). Ideally, the verifier running time should be sub-linear (or even poly-logarithmic) in the input size. In order to facilitate sub-linear computation time, the verifier is given a digest $\mathsf{d} = \mathsf{Digest}(\mathsf{crs}, x)$ of its input rather than the input itself.

Previous work on RAM delegation. Our focus is on publicly-verifiable RAM Delegation schemes that are constructed based on falsifiable and standard assumptions. There has been significant research trying to construct such delegation schemes: [KPY19; WW22] constructions achieve $\text{poly}(\lambda, t^{\epsilon})$ verification time for any constant $\epsilon > 0$ under either the standard decisional assumptions on groups with bilinear maps or the $O(1)$-LIN assumption in prime-order groups. In contrast, [CJJ21] achieves $\text{poly}(\lambda, \log t)$ verification time from LWE.

Kalai, Lombardi, Vaikuntanathan, and Wichs [KLVW22] construct the first RAM Delegation scheme under any of the following assumptions: (1) LWE; (2) $O(1)$-LIN; or (3) sub-exponential Decisional Diffie-Hellman (DDH). Their scheme has verification time $\text{poly}(\lambda, \log t)$. They prove their results using a novel transformation from any non-trivial batch argument to a highly efficient, strongly sound RAM Delegation scheme. Roughly, they begin with a non-trivial batch argument (BARG) and boost it into a highly efficient one. Then they transform the efficient BARG into an efficient RAM Delegation scheme. The [KLVW22] construction has another interesting property, which is *strong soundness*. Note that their construction focuses on *read-only* RAM machines (where the machine's memory cannot be modified).

Our RAM delegation scheme. We construct publicly-verifiable RAM Delegation scheme for *read-write* RAM machines and $\text{poly}(\lambda, \log t)$ verification time. Our RAM Delegation is also *complexity-preserving*, a property left unexplored in previous work. That is, for any RAM machine that runs in time T and space S, the honest RAM Delegation prover generates the proof in time that has quasi-linear dependence in T and space that has linear dependence in S and poly-logarithmic dependence in T. Our construction uses falsifiable assumptions (as mentioned in Theorem 1.1), while providing

[4]The discussion regarding the RAM Delegation scheme necessary for our PCA construction can be found in Section 2.2.

the *strong soundness* guarantee (as defined in [KLVW22]). [5]

Theorem 1.3 (Informal). *Under any of the assumptions mentioned in Theorem 1.1, there exists a publicly-verifiable* RAM Delegation *scheme with* strong soundness *for any* read-write RAM *machine. Moreover, the verifier running time is* $\mathrm{poly}(\lambda, \log t)$*, while the prover runs in time* $\tilde{O}_\lambda(t+n)$ *and uses* $\tilde{O}_\lambda(w \cdot \mathrm{polylog}(t) + n)$ *space.*

Here, $t = t(n)$ *is the running time of the* RAM *machine, and* $w = w(n)$ *is the number of distinct memory locations written by the machine.*

Here, the $\tilde{O}_\lambda$ notation accounts for polynomial factors in the security parameter and in the size of the local state of the machine. Note that similar to Theorem 1.1, our RAM Delegation construction actually relies on two general components, a non-interactive batch argument for NP and a rate-1 OT. Both of which can be realized under any of the assumptions mentioned in Theorem 1.1. For the formal theorem statement, please refer to Theorem 7.1.

Observe that w may be much smaller than both the running time and the total read-write memory of the RAM machine. On close inspection of the construction of [KLVW22], their prover runs in time $\tilde{\Theta}_\lambda(t^2 + n)$ and space $\tilde{\Theta}_\lambda(t + n)$. In comparison, the prover of Theorem 1.3, when applied to read-only RAM machines, runs in time $\tilde{O}_\lambda(t + n)$ and uses space $\tilde{O}_\lambda(\mathrm{polylog}(t) + n)$.

Future work on efficient RAM delegation prover. Our current research on RAM delegation schemes has yielded an efficient prover, where the running time has a quasi-linear dependency on the running time of the original RAM computation, and the space has a quasi-linear linear dependency on the space of the original RAM computation. Achieving linear dependency on the original computation complexity remains an open question.

Future work on PCAs. In [BR22], Bronfman and Rothblum introduced the implications of PCAs on the hardness of approximation. They demonstrated that if P $\neq$ NP and there exists a publicly verifiable constant-query PCA for SAT, then there exists $\epsilon > 0$ for which there is no polynomial-time algorithm solving approximate MaxSATϵ[6]. The hardness of MaxSATϵ is expected, as it follows from the PCP theorem. The interesting aspect is that we can get this result using the notion of PCAs. We believe that our new results on *succinct* PCAs for NP can be useful in achieving new hardness of approximation results.

One potential direction suggested by Bronfman and Rothblum is the hardness of approximation in Fine-Grained complexity. Previously, PCPs were used to demonstrate the hardness of approximation in the Fine-Grained complexity of problems in P ([ARW17; CGLRR19]), where a significant barrier to achieving the result was the size of the PCP proofs. We believe that the use of *succinct* PCAs (instead of PCPs) may have new implications in this field.

2 Our Techniques

In this section, we begin by introducing our techniques for constructing a PCA scheme. Initially, the prover in the PCA construction is not succinct. Once we achieve our PCA results, we'll describe

[5]Both the standard soundness and strong soundness notions are sufficient for obtaining our results. However, the strong soundness guarantee may be useful for other applications.

[6]In the MaxSAT problem, the goal is to find an assignment that maximizes the number of satisfied clauses in a formula. In the approximate MaxSAT$_\epsilon$ problem, the goal is to find an assignment that is ϵ close to maximizing the number of satisfied clauses in a formula.

how to refine our implementation to ensure a succinct prover.

To elaborate further, the primary tool for our PCA construction is RAM Delegation scheme. We start by presenting a generic PCA construction from RAM Delegation scheme (Section 2.1). This construction leads us to obtain a PCA protocol for all NP that is (1) succinct and (2) has constant query complexity. The construction relies on the assumption that there exists a succinct RAM Delegation scheme for all deterministic *read-write* RAM machines, where here, succinct means that the proof size and verification time are poly-logarithmic in the original computation. It is important to note that, at this stage, the resulting PCA protocol does not have a succinct prover.

In Section 2.2, we describe how to construct the required succinct RAM Delegation scheme for any deterministic *read-write* RAM machines. The RAM Delegation scheme is also publicly verifiable and adaptively sound, which implies the same properties for our PCA scheme. The RAM Delegation scheme, and accordingly, the PCA scheme can be realized under any of the following assumptions: (1) *polynomial* hardness of LWE, (2) $O(1)$-LIN, or (3) sub-exponential Decisional Diffie-Hellman (DDH).

Subsequently, in Section 2.3, we delve into the implementation details of the same RAM Delegation protocol introduced in Section 2.2, with a specific emphasis on achieving a complexity-preserving prover. Integrating this RAM Delegation protocol into our PCA construction naturally leads us to obtain a PCA with a succinct prover.

2.1 Adaptive PCA for all NP

The primary objective of this subsection is to provide an overview of our PCA construction. Here, our focus is on constructing a PCA protocol for all NP that is (1) succinct and (2) has constant query complexity. Recall that a succinct PCA has a proof size that is polynomial in the witness size, and constant query complexity indicates that the verifier makes a constant number of queries to the proof. Note that here, the resulting PCA prover is not succinct.

While achieving either succinctness or constant query complexity separately might be relatively straightforward, the challenge arises when we aim to achieve both properties simultaneously. In what follows, we begin with an overview of this challenge. Then, we describe the Bronfman and Rothblum [BR22] construction, which successfully addresses this challenge and results in a PCA protocol for any problem in NC. Finally, we describe our PCA construction that achieves both properties simultaneously and provides a PCA protocol for any relation in NP.

An attempt to balance succinctness and constant query complexity. Succinctness alone can be achieved by simply sending the entire witness to the verifier. The difficulty arises when we aim to combine this with constant query complexity. A standard approach would be to encode the witness using an error-correcting code and add a probabilistically checkable proof of proximity (PCPP) certifying that the NP verifier would have accepted had it read the entire witness. In such construction, for any instance $x \in \{0,1\}^n$ and witness $\mathbb{w} \in \{0,1\}^m$, the proof is: $\pi = (\mathsf{Enc}(\mathbb{w}), \Pi_{\mathsf{PCPP}})$.

This PCA construction has constant query complexity, but the proof size is $\mathrm{poly}(n, m)$, while our goal is to have the proof size sub-linear in n. This large proof size follows from the fact that known PCPPs have a proof length that depends polynomially on the running time of the computation that the PCPP certifies (as described in [BGHSV05]). In our case, the computation being certified is the NP verifier, whose running time is at least linear in n.

This issue can be resolved by replacing the NP verifier with an alternative verifier that has a significantly smaller running time. In our case, as well as in the previous construction by Bronfman and Rothblum, the new verifier will run in time $\mathrm{poly}(\log n, m)$.

The Bronfman and Rothblum PCA. Bronfman and Rothblum [BR22] constructed the first publicly verifiable succinct PCA. Their construction uses the idea described above of replacing the NP verifier with a more efficient one. In order to replace the NP verifier, [BR22] used SNARGs for P (PSNARGs). PSNARG schemes are powerful tools that allow us to verify a long deterministic computation in time that is sub-linear in the computation. Essentially, they apply a PSNARG to the NP verifier's computation, thus reducing the verification time. The resulting PCPP proof is $\pi = (\mathsf{Enc}(\mathbb{w} \mid\mid \Pi_{\mathrm{PSNARG}}), \Pi_{\mathsf{PCPP}})$.

Unfortunately, this is not enough. The PSNARG verifier needs to at least read its input, which in our case is of size $n + m$ (the NP verifier, when thought of as a P computation, takes as input $\mathbb{x}' = (\mathbb{x}, \mathbb{w})$). The PCPP proof size in this construction is again $\mathrm{poly}(n, m)$. To solve this problem, [BR22] used a specific type of PSNARG called a *holographic* PSNARG. In a holographic PSNARG the verifier is given oracle access to an (honestly generated) encoding of the input (rather than input explicitly). This enables the verifier to run in sub-linear time in the input size.

This still does not suffice, since PCPPs are not designed for computations involving an oracle. To cope with this issue, they added a pre-processing phase that hashes the encoding of the input. For this final phase to work, they need a PSNARG with even more specific requirements. For example, the prover needs to know which are the input locations that the verifier is going to query. These requirements limit [BR22] to using a very specific PSNARG construction due to [JKKZ21]. The use of this PSNARG limits their PCA to work only for NC computations, and under sub-exponential LWE.

We take a different approach to lowering the verifier running time. Rather than working with PSNARGs, we use a $\mathsf{RAM\ Delegation}$ scheme. This enables us to get around the limitations of Bronfman and Rothblum's PCA.

RAM delegation. Before delving into the details of our PCA construction, we introduce the concept of a $\mathsf{RAM\ Delegation}$ scheme. It is worth noting that $\mathsf{RAM\ Delegation}$ schemes can have different definitions in the literature. For our construction, we adopt the notion introduced in [KLVW22] and extend it to apply to *read-write* RAM machines.

In more detail, we consider read-write RAM computations where the machine is given its input in *read-only* memory and has access to a large *read-write* memory initially filled with zeros. It can access both memory modules at an arbitrary location with unit cost. To allow for later flexibility, we think of the input to the RAM machine as a pair $x = (x^{\mathsf{imp}}, x^{\mathsf{exp}})$, where we call x^{imp} the implicit input (which we think of as large), and x^{exp} the explicit input (which we think of as small).

In a $\mathsf{RAM\ Delegation}$ scheme, the prover wants to convince the verifier that $M(x) = y$ for some RAM machine M, input x, and output y. In the completeness experiment, the prover is given as input a common reference string crs and an input $x = (x^{\mathsf{imp}}, x^{\mathsf{exp}})$. The prover generates a proof Π and sends it to the verifier. The verifier is given as input the crs, a digest of the implicit input $\mathsf{d} \leftarrow \mathsf{Digest}(\mathsf{crs}, x^{\mathsf{imp}})$, the explicit input x^{exp}, and an output y. The verifier outputs accept or reject. The soundness guarantee implies that it is computationally hard to generate $(M, x = (x^{\mathsf{imp}}, x^{\mathsf{exp}}), \pi, y)$ such that $M(x) \neq y$, and yet the verifier accepts the proof with respect to $\mathsf{d} \leftarrow \mathsf{Digest}(\mathsf{crs}, x^{\mathsf{imp}})$.

For the efficiency parameters, we want the proof size and the verification time to be sub-linear in the size of the (large) implicit input x^{imp} and sub-linear in the running time t of the original RAM computation. To capture this notation, we define a $\mathsf{RAM\ Delegation}$ scheme as *succinct* if both the proof size and verification time are at most $\mathrm{poly}(\lambda, |x^{\mathsf{exp}}|, \log |x^{\mathsf{imp}}|, \log t)$.

PCA from RAM delegation. We turn back to constructing PCAs. Recall that our starting point is a proof of the following structure: $\pi = (\mathsf{Enc}(\mathbb{w}), \Pi_{\mathsf{PCPP}})$, where $\mathsf{Enc}(\mathbb{w})$ is an encoding of the witness, and Π_{PCPP} is a PCPP proof certifying that the NP verifier would have accepted had it read the entire witness $\mathbb{w}$. Our goal is to replace the NP verifier with a more efficient computation. For this purpose, we will use a *succinct* $\mathsf{RAM\ Delegation}$ scheme, in which the verifier runs in poly-logarithmic time in the input size (given a digest of the input).

Fix a relation $\mathcal{R}$ in NP, and a RAM machine M that verifies the relation. Given an instance $\mathbb{x}$ and a witness $\mathbb{w}$, consider a $\mathsf{RAM\ Delegation}$ scheme certifying that $M(x^{\mathsf{imp}}, x^{\mathsf{exp}}) = 1$, where $x^{\mathsf{imp}} = \mathbb{x}$ is the NP instance, and $x^{\mathsf{exp}} = \mathbb{w}$ is the NP witness. Observe that the running time of the succinct $\mathsf{RAM\ Delegation}$ verifier is $\mathrm{poly}(\lambda, m, \log n)$ given the digest of $\mathbb{x}$, where m refers to the witness size and n refers to the instance size.

We can now describe our PCA construction:

- <u>Prover:</u>
  1. Compute the following
     1.1. Π_{RAM}, the $\mathsf{RAM\ Delegation}$ proof certifying $M(\mathbb{x}, \mathbb{w}) = 1$.
     1.2. d, the digest of the input $\mathbb{x}$.
     1.3. Π_{PCPP}, the PCPP proof certifying that after decoding $\mathsf{Enc}(\mathbb{w} \mid\mid \Pi_{\mathsf{RAM}})$ the $\mathsf{RAM\ Delegation}$ verifier accepts given $(\mathsf{d}, \mathbb{w}, \Pi_{\mathsf{RAM}})$.
  2. Output $\Pi = (\mathsf{Enc}(\mathbb{w} \mid \Pi_{\mathsf{RAM}}), \Pi_{\mathsf{PCPP}})$.
- <u>Verifier:</u>
  1. Compute d, the digest of the input $\mathbb{x}$.
  2. Check that the PCPP verifier accepts.

This construction achieves constant query complexity directly by the PCPP scheme. To establish succinctness, we need to bound the size of the proof, where $\Pi = (\mathsf{Enc}(\mathbb{w} \mid\mid \Pi_{\mathsf{RAM}}), \Pi_{\mathsf{PCPP}})$. By the running time $\mathsf{vt} = \mathrm{poly}(\lambda, m, \log n)$ of the $\mathsf{RAM\ Delegation}$ verifier, we get that $|\Pi_{\mathsf{RAM}}| \leq \mathrm{poly}(\lambda, m, \log n)$. Recall that a PCPP proof is of size polynomial in the original computation. In our case, this is the computation of the $\mathsf{RAM\ Delegation}$ verifier. Therefore, $|\Pi| \leq \mathrm{poly}(\mathsf{vt}) + \mathrm{poly}(\lambda, m, \log n) \leq \mathrm{poly}(\lambda, m, \log n)$. Overall, we get that the construction achieves both succinctness and constant query complexity.

Regarding the publicly verifiable and adaptively sound properties of the PCA scheme, these properties naturally follow when the $\mathsf{RAM\ Delegation}$ scheme is publicly verifiable and adaptively sound.

We now turn to construct our $\mathsf{RAM\ Delegation}$ scheme.

2.2 RAM delegation

In this subsection, we outline the construction of the succinct RAM Delegation scheme essential for our PCA construction discussed in Section 2.1. By incorporating the resulting RAM Delegation scheme into our PCA construction, we achieve publicly verifiable succinct PCA with constant query complexity in the adaptive security setting under various standard assumptions. However, it is important to note that this PCA *prover* is not yet succinct. We will address the succinctness of the prover in Section 2.3 by constructing complexity-preserving RAM Delegation .

The RAM Delegation notion we use in our PCA construction is the notion presented in [KLVW22]. This notion differentiates itself from standard RAM Delegation by splitting the input into explicit and implicit components, where the explicit input is considered small and the implicit input is considered large. The verifier receives the explicit input in the clear and obtains only the digest of the implicit input. This distinction is reflected in the soundness definition, which we inherently use for proving security for our PCA construction. However, the RAM Delegation construction presented in [KLVW22] is not sufficient for our PCA construction as it specifically applies to read-only RAM machines.

There are two immediate potential approaches for achieving construction for read-write RAM machines under the required notion. We could modify existing constructions that apply to read-write RAM machines ([KPY19], [WW22], and [CJJ21]) to support explicit and implicit input. This can be achieved relatively easily. However, as we will see in Section 2.3, updating the RAM Delegation construction from [KLVW22] to apply to any read-write RAM machines allows us to leverage the specific structure of the prover in that construction to achieve a complexity-preserving prover. Ultimately, this leads to the desired succinct prover for our PCA scheme.

Moreover, extending the construction in [KPY19] maintains a stronger soundness security notion than achieved in previous RAM Delegation constructions.[7] Additionally, it provides a RAM Delegation scheme for any read-write RAM machine assuming two generic components. That is, a batch argument scheme and rate-1 OT (as formally defined in [KLVW22]). These components can, in turn, be constructed from either one of the following assumptions: (1) polynomial hardness of LWE; (2) $O(1)$-LIN; or (3) sub-exponential Decisional Diffie-Hellman (DDH), and achieve our PCA construction under those assumptions.

In what follows we describe the techniques we used for constructing our RAM Delegation scheme. The main tool that we use is a somewhere extractable batch argument (BARG) scheme.

Somewhere extractable batch arguments (seBARGs). In a batch argument scheme (BARG), the goal is to efficiently verify a batch of k NP statements. The prover is given a batch of k NP statements $\mathbb{x}_1, \ldots, \mathbb{x}_k$ along with their corresponding witnesses $\mathbb{w}_1, \ldots, \mathbb{w}_k$, and sends a short proof to the verifier. We want the proof size and verification time to be significantly smaller than the total size of the witnesses. Specifically, the main interest is to have a sub-linear (or ideally, poly-logarithmic) dependence in the number of statements.

Our construction actually uses a stronger primitive called somewhere extractable batch argument (seBARG). A BARG is said to be somewhere extractable if, for some (hidden) pre-choice of i, given a trapdoor to the crs and an accepting proof, it is possible to extract a witness $\mathbb{w}_i$ for the i-th NP statement.

[7]Our construction maintains the same *strong soundness* guarantee as defined in [KLVW22]. The weaker notion is sufficient for our construction. See Definition 3.9 for the formal definition of both security notions.

We upgrade any BARG to be seBARG using *somewhere extractable hash with local opening* (SEH). An SEH family is a family of hash functions with local openings where, given a hashed value and a suitable trapdoor for the i-th bit (generated during the SEH setup), one can efficiently extract the i-th bit of the hashed input. Upgrading BARG to be seBARG using SEH can be done by first committing to all the witnesses using the SEH, and then modifying the NP statements to include consistency with the hashed value. In our construction, we replace BARG with seBARG without loss of generality since we also use SEH explicitly (as previously done implicitly in [CJJ21; KVZ21], and explicitly in [KLVW22]).

Our RAM delegation scheme. Our $\mathsf{RAM\ Delegation}$ construction extends the Kalai, Lombardi, Vaikuntanathan, and Wichs [KLVW22] construction which, in turn, is inspired by the transformations of [CJJ21; KVZ21]. To prove that $M(x) = y$ using a BARG scheme, [KLVW22] employs a step-by-step approach, dividing the computation into smaller steps. The *read-only* RAM machine M starts with an initial state st_0. At each step $i \in [t]$, it reads one bit from the input x at some location j_i and transitions from state st_i to state st_{i+1}. The BARG proof should certify that each computation step is performed correctly.

In more detail, for computing the proof, the prover in [KLVW22] first computes the digest of the input $\mathsf{d} \leftarrow \mathsf{Digest}(x)$ (using a hash family with local openings) and a somewhere extractable hash $\mathsf{com}_{\mathsf{st}} \leftarrow \mathsf{SEH.Hash}(\mathsf{st}_0, \dots, \mathsf{st}_t)$. Then, the prover constructs a seBARG proof for the following batch NP statement: For each $i \in [t]$, (1) the values $(\mathsf{st}_{i-1}, \mathsf{st}_i)$ are consistent with $\mathsf{com}_{\mathsf{st}}$, (2) the bit $x[j_i]$ is consistent with d, (3) the machine transformation $\mathsf{st}_{i-1} \to \mathsf{st}_i$ consist with the bit $x[j_i]$, (4) if $i = t$ then check that st_t is an accepting state. The witness $\mathbb{w}_i$ includes the values $(\mathsf{st}_{i-1}, \mathsf{st}_i, x[j_i])$ along with local opening certifying the consistency of the values with their digest.

Moving forward to constructing *read-write* $\mathsf{RAM\ Delegation}$, in addition to the input and the intermediate states, to describe one computation step of a read-write RAM machine, we need more information. The read-write machine M starts with an initial state st_0 and an all-zero memory D_0. Then, at each step, the machine reads one bit from the input, reads one bit from the memory, and writes one bit to the memory. The state of the machine is then transformed from $(\mathsf{st}_{i-1}, D_{i-1})$ to (st_i, D_i).

One possible strategy is to reconsider the definition of the machine's state to include both the local state and the memory: $\mathsf{st}'_i = (\mathsf{st}_i, D_i)$. Then we can use the same algorithm in [KLVW22]. The witness, $\mathbb{w}_i$, is now the values $(\mathsf{st}'_{i-1}, \mathsf{st}'_i, x[j_i])$ along with local opening certifying the consistency of the values with their digest. However, incorporating the memory into the state definition introduces a challenge in terms of proof size. The size of the state includes the description of the entire memory, which can be arbitrarily large (even exceeding the computation time t). Consequently, the size of one witness, and accordingly the resulting seBARG proof, may become too large (larger than the computation itself).

We combine techniques from [KPY19; CJJ21] to overcome this obstacle and gain succinctness again. Instead of directly computing $\mathsf{com}_{\mathsf{st}} \leftarrow \mathsf{SEH.Hash}(\mathsf{st}'_0, \dots, \mathsf{st}'_t)$, we introduce an intermediate step. First, we compute the Merkle roots $\mathsf{rt}_0, \dots, \mathsf{rt}_t$ of the memory states $D_0, \dots, D_t$. These Merkle roots serve as compact representations of the entire memory throughout the computation. Next, we compute $\mathsf{com} \leftarrow \mathsf{SEH.Hash}((\mathsf{st}_0, \mathsf{rt}_0), \dots, (\mathsf{st}_t, \mathsf{rt}_t))$. Our batch NP statement is now: For each $i \in [t]$, (1) the values $((\mathsf{st}_{i-1}, \mathsf{rt}_{i-1}), (\mathsf{st}_i, \mathsf{rt}_i))$ are consistent with com, (2) the bit $x[j_i]$ is consistent with d, (3) the bit $D_{i-1}[k]$ is consistent with rt_{i-1}, (4) assuming the bit $x[j_i], D_{i-1}[k]$ has been read, the machine transformation is $(\mathsf{st}_{i-1}, \mathsf{rt}_{i-1}) \to (\mathsf{st}_i, \mathsf{rt}_i)$, and (5) if $i = t$ then check that st_t

is an accepting state. The witness $\mathbb{w}_i$ includes the values $((\mathsf{st}_{i-1}, \mathsf{rt}_{i-1}), (\mathsf{st}_i, \mathsf{rt}_i), x[j_i])$ along with local opening certifying the consistency of the values with their digest, and the proof that the transformation $\mathsf{rt}_{i-1} \to \mathsf{rt}_i$ is correct.

Our RAM Delegation construction is then:

- Prover:
  1. Compute the following:
     1.1. d, the digest of the input.
     1.2. $(\mathsf{st}_0, D_0), \ldots, (\mathsf{st}_t, D_t)$, the states of the machine through the computation.
     1.3. $\mathsf{rt}_0, \ldots, \mathsf{rt}_t$, the Merkle roots of the values $D_0, \ldots, D_t$.
     1.4. com, the SEH of $((\mathsf{st}_0, \mathsf{rt}_0), \ldots, (\mathsf{st}_t, \mathsf{rt}_t))$, along with correlated openings to each one of the values.
     1.5. $\mathbb{w}_1, \ldots, \mathbb{w}_t$, the witnesses for the NP statements.
     1.6. Π, the seBARG proof for the NP statements (using $\mathbb{w}_1, \ldots, \mathbb{w}_t$).
  2. Output (Π, com).

- Verifier:
  1. Given the digest d, check that the BARG proof accepts (relative to the NP statement defined by com).

Our analysis is based on the analysis in [KLVW22]. However, in our case, we need to pay special attention to the proof of each computation step, verifying efficiently that the transformation $((\mathsf{st}_{i-1}, \mathsf{rt}_{i-1}) \to (\mathsf{st}_i, \mathsf{rt}_i))$ is done correctly. See full analysis in Section 6.

Note that the space complexity of the prover is $\tilde{O}_\lambda(t + S + n)$ since the prover simulates the RAM machine that uses space of size S, and it holds a list of t elements in the memory. Moreover, the running time of the prover is $\tilde{O}_\lambda(t \cdot S + n)$ since it computes the Merkle root of the memory at each step (which takes time $t \cdot S$), and in step 1.4. it computes the t openings to com (which takes time t for each opening). Here, $\tilde{O}_\lambda$ notation accounts for polynomial factors in the security parameter and in the size of the machine's local state. In the following subsection, we improve the time and space complexity of the prover.

2.3 Complexity-preserving RAM delegation.

Up to this point, we constructed a publicly verifiable succinct PCA with constant query complexity for all NP in the adaptive security setting. However, the PCA prover is not yet succinct. The running time of our PCA prover (described in Section 2.2) is dominated by the running time of the RAM Delegation prover. Ultimately, improving the running time of the RAM Delegation prover to be complexity-preserving results in a PCA scheme with a succinct prover.[8]

This subsection presents an efficient implementation of the RAM Delegation prover introduced in Section 2.2, that achieves complexity-preserving RAM Delegation scheme.

The implementation of the prover presented in Section 2.2 demonstrates a time complexity of $\tilde{O}_\lambda(t^2 + t \cdot S + n)$, and space complexity of at least $\tilde{O}_\lambda(t + S + n)$, where S denotes the size of the *large* read-write memory used by the machine. For simplicity, in what follows, we will focus on a

[8]See Section 4 for the running time analysis.

specific inefficient part of the prover's computation and see how to improve both the running time and space complexity. By applying the same approach to other parts of the computation, we can achieve time complexity $\tilde{O}_\lambda(t+n)$ and a space complexity $\tilde{O}_\lambda(w \cdot \text{polylog}(t)+n)$, where w is the number of *distinct* memory locations written by the RAM machine.

For this high-level overview, we are specifically targeting a simplified version of the prover, with a focus on enhancing selected steps. These steps include the computation of:
1. $(\mathsf{st}_1, \ldots, \mathsf{st}_t)$, the machine's states list through the process.
2. $(\mathsf{com}, \rho_1, \ldots, \rho_t)$, a commitment to the machine's states list and the corresponding openings.
3. $(\mathbb{w}_1, \ldots, \mathbb{w}_t)$, the witnesses to the NP statement.
4. Π, the BARG proof for the NP statements which is generated using $(\mathbb{w}_1, \ldots, \mathbb{w}_t)$.

Note that the underlying structure of the commitment scheme in our construction is essentially a Merkle tree. Hence, in the subsequent discussion, we will refer to the commitment mentioned in step 2 as a Merkle root, and the openings as the authentication paths within this tree.

We divided the description of our improved implementation into three phases. The starting point is the naive implementation as described above that achieves $\tilde{O}_\lambda(t^2+t\cdot S+n)$ running time and $\tilde{O}_\lambda(t+S+n)$ space. Then, in each phase, we incrementally improve these parameters as follows:
- In phase one, we achieve $\tilde{O}_\lambda(t^2+t\cdot w+n)$ running time and $\tilde{O}_\lambda(t+w+n)$ space.
- In phase two, we achieve $\tilde{O}_\lambda(t\cdot w+n)$ running time and $\tilde{O}_\lambda(w\cdot \text{polylog}(t)+n)$ space.
- In phase three, we achieve $\tilde{O}_\lambda(t+n)$ running time and $\tilde{O}_\lambda(w\cdot \text{polylog}(t)+n)$ space.

Phase one: Our first step towards complexity-preserving is to reduce the large read-write space factor and achieve $\tilde{O}_\lambda(t^2+t\cdot w+n)$ running time and $\tilde{O}_\lambda(t+w+n)$ space.

For generating the RAM proof, the prover is required to at least simulate the RAM computation. It is important to note that even though the machine might have access to a large read-write memory, it might only write to a small number of locations. In other words, the memory might be sparse. In our construction, we take advantage of this property by emulating the machine's read-write memory using a *sparse hash tree* scheme.

Sparse hash tree. A *sparse hash tree* scheme is a data structure that represents a Merkle tree, particularly suited for sparse Merkle trees, meaning that most of the leaf nodes are empty. The sparsity of the tree enables efficient simulation of the tree, with time complexity proportional to the tree's depth and the number of non-empty elements, rather than being proportional to the entire tree's size. Specifically, we construct a sparse hash tree that requires space that is linear in the number of non-zero elements, and each read or write operation takes time that is proportional to the depth of the tree.

The RAM machine starts with all zero memory and writes to at most w distinct locations in the memory, which allows us to emulate the read-write memory of size S with $\tilde{O}_\lambda(w)$ space. This naturally results in a time complexity of $\tilde{O}_\lambda(t^2+t\cdot w+n)$ and a space complexity of $\tilde{O}_\lambda(t+w+n)$.

Phase two: In what follows, we reduce the running time to $\tilde{O}_\lambda(t\cdot w+n)$ and space to $\tilde{O}_\lambda(w\cdot \text{polylog}(t)+n)$. For that purpose, the focus of this phase is to maintain a running time that is quasi-linear in t, and simultaneously, avoid storing t elements in the memory.

Initially, our $\mathsf{RAM\ Delegation}$ prover stores the entire computation history $(\mathsf{st}_1, \ldots, \mathsf{st}_t)$ in memory, leading to a memory overhead of $\tilde{O}_\lambda(t)$. Instead, we construct a stream of the following data: $(\mathsf{st}_1, \ldots, \mathsf{st}_t)$. However, simply creating a stream of states is not sufficient, as we need to access this information multiple times for various computations. [9] Same challenges also apply for the list of openings $(\rho_1, \ldots, \rho_t)$ and for the list of witnesses $(\mathbb{w}_1, \ldots, \mathbb{w}_t)$. To address this issue, we introduce the new notion of *rewindable stream.*

Rewindable stream. A *stream* is a sequence of elements that are accessed incrementally over time. In our context, we introduce the concept of a *rewindable stream*, which refers to the ability to return to a specific point in the stream and continue generating the stream from that point onward. In more detail, a rewindable stream allows a client to efficiently create a backup of the stream's state at any time and later restore the stream's state to that specific time point.

In what follows, we describe how to (1) construct rewindable stream of states $(\mathsf{st}_1, \ldots, \mathsf{st}_t)$, (2) compute the commitment com to the state's list based on the stream of states, (3) compute a rewindable stream of openings $(\rho_1, \ldots, \rho_t)$ using the stream of states, and (4) generate the BARG proof in step 4 based on a stream of witnesses (which is essentially the stream of openings).

Rewindable stream of states. The computation of a RAM machine progresses in a step-by-step manner. Given a configuration of the machine, which includes its state and the contents of its read-write memory, the next configuration can be computed efficiently. Leveraging this structure, we can design a rewindable stream where the states of the stream correspond to an intermediate configuration of the RAM machine. For the backup operation, we simply copy the entire intermediate configuration of the RAM machine. This process can be implemented using $\tilde{O}_\lambda(w)$ time and space, as we use a sparse hash tree to emulate the machine's memory.

In what follows we explain how to compute the commitment com to the state's list based on the stream of states. Recall that we refer to the commitment in step 2 as a Merkle root, and the openings are the authentication paths in the Merkle tree. This brings us to the broader challenge of efficiently computing a Merkle root when provided with rewindable stream access to the data.

Computing the commitment com. Our task is to efficiently compute Merkle root given a rewindable stream access to the data. It is well-known that computing a Merkle root can be done in quasi-linear time while storing only a logarithmic number of elements in memory (as proved in [Szy04]). A careful look at the algorithm will show that this can be done using a single pass over the data, allowing us to use the stream of states to implement step 2 while storing only $\log t$ elements in the memory, and in time $\tilde{O}_\lambda(t \cdot \log t)$.

We have explained how to efficiently compute the commitment com, and reduce the space complexity required for this operation. Now, we will describe how to efficiently construct a rewindable stream of openings. Both steps together ensure the time and space efficiency of step 2 .

[9] Note that naively traversing the entire stream from the beginning for each access to st_i would result in an overhead of at least $\tilde{O}_\lambda(t^2)$ running time.

Rewindable stream of openings. Recall that each opening to com is essentially an authentication path in a Merkle tree. An authentication path can be computed in a quasi-linear time while storing only a logarithmic number of elements in memory (as proved in [Szy04]). We can use the algorithm to generate a stream of t authentication paths while storing only $\log t$ elements in the memory. However, each authentication path is computed in time $\tilde{O}_\lambda(t \cdot \log t \cdot w)$, which concludes with an overall running time of $\tilde{O}_\lambda(t^2 \cdot \log t \cdot w)$ for computing all the authentication paths.

Our next challenge is to reduce the running time of the stream. Instead of generating each authentication path auth_{i+1} from scratch, our algorithm leverages the information in the previous authentication path auth_i. An authentication path represents the path from a leaf to the root and consists of the sibling's values encountered along the way. By exploiting the concept of the *lowest common ancestor*, we observe that the leaves $(i, i+1)$ share the same path from the lowest common ancestor to the root and, therefore, also share the same siblings from the lowest common ancestor to the root. Consequently, for the lowest common ancestor of height h, these two authentication paths have $(\log t - h)$ identical values. With this insight, we can split the computation of Next into two steps. The first step will take the relevant data from the previous authentication path auth_i, and the second step will compute the authentication path within the sub-tree of height h. By utilizing the *rewind* property of the stream, which allows us to start from a specific point in the stream, the second step will take $\tilde{O}_\lambda(2^h \cdot h \cdot w)$ time rather than $\tilde{O}_\lambda(t \cdot \log^2 t \cdot w)$.

This approach achieves a running time of $\tilde{O}_\lambda(t \cdot \log^2 t \cdot w)$ for one pass over the entire stream. In other words, in the amortized case, the running time is $\tilde{O}_\lambda(\log^2 t \cdot w)$.[10]

To efficiently compute the list of witnesses (step 3), we construct a rewindable stream of witnesses that contain the machine's states and the corresponding openings to com by simply wrapping the rewindable streams we constructed. The following step computes the BARG proof efficiently using only rewindable stream access to the witnesses, thereby avoiding the need for explicit access to the witness list.

Computing the BARG proof. For this step, we need to construct an efficient BARG with stream rewind access to the witnesses. For t instances, the BARG prover will run in time $\tilde{O}(t \cdot |\mathbb{w}|) \cdot \mathrm{poly}(\lambda)$, and use space of size $\tilde{O}(|\mathbb{w}| \cdot \mathrm{polylog}(t)) \cdot \mathrm{poly}(\lambda)$. Moreover, the prover will access each element in the stream at most $\mathrm{polylog}(t)$ times. Note that the bounded access to the stream will allow us to start with a stream that is efficient only in the amortized case, and yet conclude with $\tilde{O}(t \cdot |\mathbb{w}|) \cdot \mathrm{poly}(\lambda)$ time complexity.

We use the BARG scheme constructed in [KLVW22], but we suggest an alternative implementation for the prover. The implementation will use the same techniques as described above for the $\mathsf{RAM\ Delegation}$ efficient prover (see Section 7.3 for more details).

Overall, we reduced the t^2 overhead in the running time and t space overhead. This leads us to $\tilde{O}_\lambda(t \cdot w + n)$ running time and $\tilde{O}_\lambda(w \cdot \mathrm{polylog}(t) + n)$ space complexity.

<u>Phase three:</u> As a final step toward complexity-preserving, in this phase, we achieve $\tilde{O}_\lambda(t + n)$ running time and $\tilde{O}_\lambda(w \cdot \mathrm{polylog}(t) + n)$ space.

To achieve this, we implement the rewindable stream of states in a way that each operation in the stream, including backup, restore, and advancing to the next element, only takes $\tilde{O}_\lambda(\mathrm{polylog}(t))$

[10]For a more detailed analysis of the algorithm's efficiency, please refer to Section 7.1.

time, rather than $\tilde{O}_\lambda(w \cdot \mathrm{polylog}(t))$. By following a similar analysis as in Phase 2, but with the improved stream running time, we attain the necessary complexity parameters for the prover.

The challenge arises in the stream backup operation that requires copying the entire memory of the RAM machine. Here, instead of copying the entire memory, we use a data structure that represents the RAM memory and allows for efficient backup operation.

Memory scheme. *Memory scheme* is a data structure that represents a memory of a deterministic program, allowing for reading, writing, backing up, and restoration of the memory.

We construct a memory scheme, where each operation, including backups, takes time that is poly-logarithmic in the size of the original memory. This construction allows us to implement the rewindable stream of states efficiently, and consecutively, to achieve our desirable complexity parameters.

3 Preliminaries

Notations. We denote the set of all positive integers up to n as $[n] := \{1, ..., n\}$. For any two string $x, y \in \Sigma^*$ over alphabet Σ, we denote $(a \mid\mid b)$ to be the concatenation of the two strings. The relative distance between strings $x, y \in \Sigma^\ell$ over alphabet Σ is $\Delta(x, y) := \frac{|\{i \mid x_i \neq y_i\}|}{\ell}$. The relative distance between $x \in \Sigma^\ell$ and a non-empty set $S \subseteq \Sigma^\ell$ is $\Delta(x, S) := \min_{y \in S}(\Delta(x, y))$.

3.1 Probabilistically checkable proofs

A probabilistically checkable proof (PCP) is a special format for writing a proof that can be verified by reading only a few bits. The following definition is taken from [BR22].

Definition 3.1 (PCP). *A probabilistically checkable proof (PCP) for language $L \in \mathrm{NTIME}(t)$ consists of a $\mathrm{poly}(t)$ prover PCP.P, that gets as input the instance $\mathbb{x}$ as well as a witness $\mathbb{w}$, and a $\mathrm{poly}(|\mathbb{x}|, \log t)$ time oracle machine PCP.V, that recieves $\mathbb{x}$ as input as well as an oracle to a proof string π.*

Completness. *For every $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ it holds that:*

$$\Pr\left[\ \mathsf{PCP.V}^{\pi}(\mathbb{x}) = 1 \ \middle|\ \pi \leftarrow \mathsf{PCP.P}(\mathbb{x}, \mathbb{w})\ \right] = 1\ .$$

Soundness. *For every $\mathbb{x} \notin L$, and for every oracle $\tilde{\pi}$ it holds that*

$$\Pr\left[\mathsf{PCP.V}^{\tilde{\pi}}(\mathbb{x}) = 1\right] < \frac{1}{2}$$

The length of π as a function of $|\mathbb{x}|$ and $|\mathbb{w}|$ is called the proof length. In order to verify its oracle, the verifier PCP.V tosses $\mathsf{r} = \mathsf{r}(|\mathbb{x}|)$ random coins, and makes $\mathsf{q} = \mathsf{q}(|\mathbb{x}|)$ queries to π. The functions r and q are called the randomness complexity and query complexity, respectively.

Theorem 3.2 ([ALMSS98]). *Every $L \in \mathrm{NP}$ has a PCP with constant query complexity and logarithmic randomness complexity*

Succinct PCPs. A PCP for $L \in \mathrm{NP}$ is said to be succinct [FS08; KR09] if there exists a polynomial p (which may depend on L), such that for every $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$ it holds that $|\pi| = \mathsf{p}(|\mathbb{w}|, \log(|\mathbb{x}|))$, for $\pi = \mathsf{PCP.P}(\mathbb{x}, \mathbb{w})$.

3.2 Probabilistically checkable proofs of proximity

In what follows, we define probabilistically checkable proofs of proximity (PCPP). The definition is taken from [BR22].

PCPPs, much like PCPs, allows the verifier to read only a small number of bits from the proof. However, the key distinction between PCPs and PCPPs is that a PCPP verifier also reads only a small number of bits from its *input*, and is therefore, only required to reject inputs that are far from its language. In what follows, we define PCPP for pair languages.

Pair languages. A language L is said to be a pair language if $L \subseteq \{0,1\}^* \times \{0,1\}^*$. Given instance $\mathbb{x}$, the projection of L on $\mathbb{x}$ is the set $L(\mathbb{x}) = \{y \mid (\mathbb{x}, y) \in L\}$.

Definition 3.3 (PCPP). *A probabilistically checkable proof of proximity (*PCPP*) for a pair language $L \in \mathrm{DTIME}(t)$ consists of a $\mathrm{poly}(t)$ prover* $\mathsf{PCPP.P}$*, that gets as input the pair $(\mathbb{x}, y)$ and a $\mathrm{poly}(|\mathbb{x}|, \log t)$ time oracle machine* $\mathsf{PCPP.V}$*, that recieves $\mathbb{x}$ as an explicit input, oracle access to implicit input y, and oracle access to a proof string π. The verifier also recieves (explicitly) a proximity parameter $\delta > 0$. For proximity parameter $\delta \in [0,1]$ and input $(\mathbb{x}, y)$ the following holds:*

Completness. *For every $(\mathbb{x}, y) \in L$ it holds that*

$$\Pr\left[\ \mathsf{PCPP.V}^{y,\pi}(\mathbb{x}, |y|, |\pi|, \delta) = 1 \ \middle|\ \pi \leftarrow \mathsf{PCPP.P}(\mathbb{x}, y)\ \right] = 1\ .$$

Soundness. *For every $\Delta(y, L(\mathbb{x})) \geq \delta$, and for every oracle $\tilde{\pi}$ it holds that*

$$\Pr\left[\mathsf{PCPP.V}^{y,\tilde{\pi}}(\mathbb{x}, |y|, |\pi|, \delta) = 1\right] < \frac{1}{2}$$

The length of π as a function of $|\mathbb{x}|$ and $|y|$ is called the proof length. In order to verify its oracles, the verifier $\mathsf{PCPP.V}$ *tosses $\mathsf{r} = \mathsf{r}(|\mathbb{x}|, |y|, \delta)$ random coins, and makes $\mathsf{q} = \mathsf{q}(|\mathbb{x}|, |y|, \delta)$ queries to y and π. The functions r and q are called the randomness complexity and query complexity, respectively.*

Theorem 3.4 ([BGHSV05])**.** *Let L be a pair language, with instances $(\mathbb{x}, y)$, computable in time $t = t(|\mathbb{x}|, |y|)$, and let $\delta \in [0,1]$. There exists a* PCPP *for L with respect to proximity parameter δ, with query complexity $O(1/\delta)$, randomness complexity $O(\log \frac{t}{\delta})$, and proof length $\mathrm{poly}(t)$. The verifier runs in time $\mathrm{poly}(|\mathbb{x}|, \log t, \frac{1}{\delta})$ and the prover runs in time $\mathrm{poly}(t)$.*

3.3 Probabilistically checkable arguments

In what follows, we define probabilistically checkable argument (PCA). The definition is taken from [BR22].

Much like a PCPs , PCAs are a special format in which the verifier only has to read a few bits from the proof. Unlike PCPs, in which any proof for a false statement is rejected with high probability, for PCAs, accepting proofs may exist, but we require that it is computationally hard to find them.

Definition 3.5 (publicly-verifiable PCA.). *A publicly-verifiable probabilistically checkable argument (*PCA*) for an* NP *relation* $\mathcal{R}$ *is a triplet of* $\mathrm{poly}(n, m, \lambda)$*-time algorithms* $(\mathsf{PCA.G}, \mathsf{PCA.P}, \mathsf{PCA.V})$*, with deterministic* $\mathsf{PCA.P}$ *and probabilistic* $\mathsf{PCA.G}$ *and* $\mathsf{PCA.V}$*, such that for every instance length* n *and witness length* m *the following holds:*

Completness. *For every* $\mathbb{x} \in \{0,1\}^n$ *and* $\mathbb{w} \in \{0,1\}^m$*, such that* $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$*, every* $\lambda \in \mathbb{N}$*, and every* $\mathsf{crs} \leftarrow \mathsf{PCA.G}(1^n, 1^m, 1^\lambda)$ *it holds that:*

$$\Pr\left[\ \mathsf{PCA.V}^{\pi}(\mathsf{crs}, \mathbb{x}) = 1 \ \middle|\ \pi \leftarrow \mathsf{PCA.P}(\mathsf{crs}, \mathbb{x}, \mathbb{w})\ \right] = 1\ .$$

Computational adaptive soundness. *For every* $\lambda \in \mathbb{N}$*, and poly-size adversary* $\tilde{\mathsf{P}}$*, with all but* $\mathsf{negl}(\lambda)$ *probability over the choice of* $\mathsf{crs} \leftarrow \mathsf{PCA.G}(1^n, 1^m, 1^\lambda)$ *it holds that:*

$$\Pr\left[\begin{array}{l} \mathsf{PCA.V}^{\tilde{\pi}}(\mathsf{crs}, \mathbb{x}) = 1 \\ \mathcal{R}(\mathbb{x}) = \emptyset \end{array} \ \middle|\ (\mathbb{x}, \tilde{\pi}) \leftarrow \tilde{\mathsf{P}}(\mathsf{crs})\ \right] < \frac{1}{2}\ .$$

The length of π*, as a function of* n*,* m*, and* λ *is called the proof length. In order to verify its oracle, the verifier* $\mathsf{PCA.V}$ *tosses* $\mathsf{r} = \mathsf{r}(n, m, \lambda)$ *random coins, and makes* $\mathsf{q} = \mathsf{q}(n, m, \lambda)$ *queries to* π*. The functions* r *and* q *are called the randomness complexity and query complexity, respectively.*

Note that in Definition 3.5 we distinguish between the randomness used for generating the crs and the randomness of the verifier. This separation allows us to define the crs as "good" with overwhelming probability while the verifier only guarantees constant soundness (since the verifier only makes a small number of queries).

Succinct PCAs. A PCA for an NP relation $\mathcal{R}$, which is decidable in some time $t = t(n, m) \geq n$, is said to be succinct if the PCA proof is of length $\mathrm{poly}(m, \lambda, \log t)$, where poly refers to a fixed universal polynomial (that does not depend on the relation $\mathcal{R}$).

PCA with a succinct prover. We say that a PCA scheme has a *succinct prover* if for any NP relation $\mathcal{R}$ that can be verified in time $t = \mathrm{poly}(n, m)$ and space $s = \mathrm{poly}(n, m)$, the PCA prover (given the instance and the witness) generates the proof in time $t \cdot \mathrm{poly}(\lambda, m, \log t)$ and space $s \cdot \mathrm{poly}(\lambda, m, \log t)$. Here, poly refers to a fixed universal polynomial (that does not depend on the relation $\mathcal{R}$).

3.4 Batch arguments

In what follows, we define batch argument (BARG) scheme for $\mathsf{BatchCSAT}$. The definition is taken from [KLVW22].

Let CSAT be the following language:

$$\mathsf{CSAT} = \{(C, \mathbb{x}) \mid \exists \mathbb{w} \in \{0,1\}^m \text{ s.t. } C(\mathbb{x}, \mathbb{w}) = 1\}$$

where $C : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ is a Boolean circuit and $\mathbb{x} \in \{0,1\}^n$ is an instance.
Let $\mathsf{BatchCSAT}$ be the following language:

$$\mathsf{BatchCSAT} = \{C \mid \exists \mathbb{w}_1, \ldots, \mathbb{w}_k \in \{0,1\}^m \text{ s.t. } \forall i \in [k],\ C(i, \mathbb{w}_i) = 1\}$$

Definition 3.6 (BARG for $\mathsf{BatchCSAT}$)**.** *A non-interactive batch argument for the index language* $\mathsf{BARG} = (\mathsf{BARG.G}, \mathsf{BARG.P}, \mathsf{BARG.V})$ *has the following syntax:*

$\mathsf{BARG.G}(1^\lambda, k, s, i^*) \to (\mathsf{crs}, \mathsf{td})$**.** *This is a randomized algorithm that takes as input a security parameter 1^λ, a number of instances k, a circuit size 1^s, and an index $i^* \in [k]$. It outputs a common reference string* crs*, and a trapdoor* td*.*

$\mathsf{BARG.P}(\mathsf{crs}, C, \mathbb{w}_1, \dots, \mathbb{w}_k) \to \pi$**.** *This is a deterministic poly-time algorithm that takes as input a common reference string* crs*, a circuit $C : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ and a witnesses $\mathbb{w}_1, \dots, \mathbb{w}_k \in \{0,1\}^m$. It outputs a proof π.*

$\mathsf{BARG.V}(\mathsf{crs}, C, \pi) \to \{0,1\}$**.** *This is a deterministic poly-time algorithm that takes as input a circuit $C : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$, and a proof π. It outputs an acceptance bit.*

An $\mathcal{L}(\cdot,\cdot)$-succinct BARG *protocol for the relation $\mathcal{R}$ satisfies the following requirements:*

$\mathcal{L}$-Succinct. *The running time of* $\mathsf{BARG.G}$ *is at most $\mathcal{L}(k,\lambda) \cdot \mathrm{poly}(s)$, and the length of the* crs *and the proof π is at most $\mathcal{L}(k,\lambda) \cdot \mathrm{poly}(s)$.*

$\mathcal{L}$-Verifier efficiency. *The running time of* $\mathsf{BARG.V}$ *is at most $\mathcal{L}(k,\lambda) \cdot \mathrm{poly}(s)$.*

Prover efficiency. *The running time of* $\mathsf{BARG.P}$ *is polynomial in its input.*

Completeness. *For every $k = k(\lambda), m = m(\lambda), s = s(\lambda)$ of size at most 2^λ, witnesses $\mathbb{w}_1, \dots, \mathbb{w}_k \in \{0,1\}^m$, and circuit $C \in \{0,1\}^s$ such that $\forall i \in [k]\ C(i, \mathbb{w}_i) = 1$, and for every index $i^* \in [k]$, there exists a negligible function μ such that for any $\lambda \in \mathbb{N}$:*

$$\Pr\left[\ \mathsf{BARG.V}(\mathsf{crs}, C, \pi) = 1 \ \middle|\ \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{BARG.G}(1^\lambda, s, k, m, i^*) \\ \pi \leftarrow \mathsf{BARG.P}(\mathsf{crs}, C, (\mathbb{w}_1, \dots, \mathbb{w}_k)) \end{array} \right] = 1 - \mu(\lambda) \ .$$

Indistiguishability. *For every poly-size adversary A, and any polynomials $k = k(\lambda), m = m(\lambda), s = s(\lambda)$, there exists a function μ such that for every $\lambda \in \mathbb{N}$ and two indexes $i_1, i_2 \in [k]$:*

$$\left| \begin{array}{l} \Pr\left[A(\mathsf{crs}) = 1 \mid (\mathsf{crs}, \cdot) \leftarrow \mathsf{BARG.G}(1^\lambda, s, k, m, i_1)\right] \\ -\Pr\left[A(\mathsf{crs}) = 1 \mid (\mathsf{crs}, \cdot) \leftarrow \mathsf{BARG.G}(1^\lambda, s, k, m, i_2)\right] \end{array} \right| \le \mu(\lambda) \ .$$

Semi-adaptive soundness. *For every poly-size adversary A, and any polynomials $k = k(\lambda), m = m(\lambda), s = s(\lambda)$, there exists a function μ such that for every $\lambda \in \mathbb{N}$ and index $i^* \in [k]$:*

$$\Pr\left[\ \begin{array}{l} \mathsf{BARG.V}(\mathsf{crs}, C, \pi) = 1 \\ (C, i^*) \notin \mathsf{CSAT} \end{array} \ \middle|\ \begin{array}{l} (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{BARG.G}(1^\lambda, s, k, m, i^*) \\ (C, \pi) \leftarrow A(\mathsf{crs}) \end{array} \right] \le \mu(\lambda) \ .$$

Definition 3.7 (seBARG for $\mathsf{BatchCSAT}$)**.** *A somewhere extractable non-interactive batch argument for the index language* $\mathsf{seBARG} = (\mathsf{BARG.G}, \mathsf{BARG.P}, \mathsf{BARG.V}, \mathsf{BARG.E})$ *is a* BARG *with the following augmented syntax:*

$\mathsf{BARG.E}(\mathsf{td}, C, \pi) \to \mathbb{w}^*$**.** *This is a deterministic poly-time algorithm that takes as input a trapdoor key* td*, a circuit $C : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$, and a proof π. It outputs a witness $\mathbb{w} \in \{0,1\}^m$.*

An $\mathcal{L}(\cdot,\cdot)$-succinct seBARG *protocol for the relation $\mathcal{R}$ satisfies the following additional requirements to that of an $\mathcal{L}(\cdot,\cdot)$-succinct* BARG

Somewhere argument of knowelege. *For every poly-size adversary A, and any polynomials $k = k(\lambda), m = m(\lambda), s = s(\lambda)$, there exists a function μ such that for every $\lambda \in \mathbb{N}$ and index $i^* \in [k]$:*

$$\Pr\left[\begin{array}{l|l} \mathsf{BARG.V}(\mathsf{crs}, C, \pi) = 1 & (\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{BARG.G}(1^\lambda, s, k, m, i^*) \\ C(i^*, \mathbb{w}^*) \neq 1 & (C, \pi) \leftarrow A(\mathsf{crs}) \\ & \mathbb{w}^* \leftarrow \mathsf{BARG.E}(\mathsf{td}, \pi) \end{array}\right] \leq \mu(\lambda) \ .$$

The following theorem follows by implying the [KLVW22] BARG transformation on the BARG constructions from [CJJ21; WW22; CGJJZ23].

Theorem 3.8 ([KLVW22; CJJ21; WW22; CGJJZ23]). *There exist* $\mathrm{poly}(\lambda, \log k)$*-succinct index BARGs for* $\mathsf{BatchCSAT}$ *with somewhere argument of knowledge under any of the following assumptions:*

1. *The* $O(1)$-LIN *assumption on a pair of cryptographic groups with efficient bilinear map.*
2. *The hardness of Learning with errors* (LWE) *problem against polynomial time adversaries.*
3. *The sub-exponential Decisional Diffie-Hellman (*DDH*) assumptions.*

3.5 RAM delegation

In what follows we define $\mathsf{RAM\ Delegation}$ scheme for *read-write* RAM machines. The definition is taken from [KLVW22].

A *read-write* RAM machine is modeled as a deterministic machine with random access to a *read-write* memory of large size. In its standard definition, the machine has a local state of length logarithmic in the memory size. At each time step, the machine reads or writes to a single memory cell and updates its local state. Often it is assumed that the machine has no input outside of its memory.

We say that without loss of generality, the RAM machine has random access to a *read-only* memory of size n and a large *read-write* memory initially filled with zeros. We refer to the specific memory locations where the machine performs write operations as the *write-memory* of the machine. We denote $t(n)$ as the number of steps executed by the machine, and $w(n)$ as the size of the write-memory. The input to the RAM machine is represented as a pair $x = (x^{\mathsf{imp}}, x^{\mathsf{exp}})$, where x^{imp} is large and stored in the random access memory, and x^{exp} is a compact explicit input. Having defined this model, we can now proceed to formalize the concept of $\mathsf{RAM\ Delegation}$.

Definition 3.9. *An $\mathcal{L}$-succinct* $\mathsf{RAM\ Delegation} = (\mathsf{RAM.G}, \mathsf{RAM.D}, \mathsf{RAM.P}, \mathsf{RAM.V})$ *for a* RAM *computation M with local state of size* $\mathsf{L_{st}} \geq |x^{\mathsf{exp}}| + \log |x^{\mathsf{imp}}|$*, satisfies the following properties:*

$\mathcal{L}$-succinct. *The running time of* $\mathsf{RAM.G}$ *is at most* $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L_{st}})$*, and the length of a proof π is at most* $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L_{st}})$.

$\mathcal{L}$-Verifier efficiency. *The running time of* $\mathsf{RAM.V}$ *is at most* $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L_{st}})$.

Prover efficiency. *The running time of* $\mathsf{RAM.P}$ *is polynomial in its input and the read-write memory of the machine.*

Digest efficiency. *The running time of* $\mathsf{RAM.D}$ *is linear in its input,* $|x| \cdot \mathrm{poly}(\lambda)$.

Completeness. *For any* $\lambda, n \in \mathbb{N}$ *such that* $n \leq w(n) \leq t(n) \leq 2^\lambda$*, and for any* $x = (x^{\mathsf{imp}}, x^{\mathsf{exp}}) \in \{0,1\}^n$ *such that* $M(x)$ *halts within* t *time steps, we have that*

$$\Pr\left[\begin{array}{l} \mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, b, \pi) = 1 \\ M(x) = 1 \end{array} \middle| \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{RAM.G}(1^\lambda, t) \\ \mathsf{d}^{\mathsf{imp}} = \mathsf{RAM.D}(\mathsf{crs}, x^{\mathsf{imp}}) \\ (b, \pi) \leftarrow \mathsf{RAM.P}(\mathsf{crs}, x^{\mathsf{imp}}, x^{\mathsf{exp}}) \end{array}\right] = 1 - \mathsf{negl}(\lambda) \ .$$

Collistion resistance of RAM **digest.** *For any poly-size adversary* A*, and polynomial* $t = t(\lambda)$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathsf{RAM.D}(\mathsf{crs}, x) = \mathsf{RAM.D}(\mathsf{crs}, x') \\ x \neq x' \end{array} \middle| \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{RAM.G}(1^\lambda, t) \\ (x, x') \leftarrow A(\mathsf{crs}) \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

Weak soundness. *For any poly-size adversary* A*, and polynomial* $t = t(\lambda)$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, 0, \pi_0) = 1 \\ \mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, 1, \pi_1) = 1 \end{array} \middle| \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{RAM.G}(1^\lambda, t) \\ (x = (x^{\mathsf{imp}}, x^{\mathsf{exp}}), \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{d}^{\mathsf{imp}} \leftarrow \mathsf{RAM.D}(\mathsf{crs}, x^{\mathsf{imp}}) \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

We say that the $\mathsf{RAM\ Delegation}$ *scheme has a* strong soundness *if it also satisfies the following strong soundness definition:*

Strong soundness. *For any poly-size adversary* A*, and polynomial* $t = t(\lambda)$*, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for every* $\lambda \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, 0, \pi_0) = 1 \\ \mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, 1, \pi_1) = 1 \end{array} \middle| \begin{array}{l} \mathsf{crs} \leftarrow \mathsf{RAM.G}(1^\lambda, t) \\ (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

4 PCA from RAM Delegation

In this section we construct a publicly-verifiable and adaptively sound succinct PCA with constant query complexity for any relation in NP. The construction is introduced in Section 4.1. We then proceed to prove the completeness, efficiency, and soundness of the scheme in Section 4.2. By combining these results, we establish the following theorem.

Theorem 4.1. *Assume there exists a* $\mathrm{poly}(\log t, \lambda)$*-succinct* $\mathsf{RAM\ Delegation}$ *with weak soundness for any read-write* RAM *machine, then for any relation* $\mathcal{R}$ *in* NP *there exists a publicly-verifiable and adaptively sound succinct* PCA *with constant query complexity.*

Furthermore,

- *The verifier runs in time* $n \cdot \mathrm{poly}(\lambda) + \mathrm{polylog}(\lambda, m, \log n)$.
- *The verifier has randomness size* $O(\log m + \mathrm{loglog} n + \log \lambda)$.

- *The prover runs in time $T + n \cdot \mathrm{poly}(\lambda) + \mathrm{poly}(\lambda, m, \log n)$, and uses space $S + n \cdot \mathrm{poly}(\lambda) + \mathrm{poly}(\lambda, m, \log n)$, where T, S are the running time and space complexity of the* RAM Delegation *prover.*

Where n is the instance size, and m is the witness size.

By combining Theorem 4.1 with Corollary 7.2 we get the following corollary.

Corollary 4.2. *For any relation $\mathcal{R}$ in* NP *there exists a publicly-verifiable and adaptively sound succinct* PCA *with constant query complexity under any of the following assumptions:*

1. *The $O(1)$-*LIN *assumption on a pair of cryptographic groups with efficient bilinear map.*
2. *The hardness of Learning with errors* (LWE) *problem against polynomial time adversaries.*
3. *The sub-exponential Decisional Diffie-Hellman (*DDH*) assumptions.*

Furthermore,

- *The verifier runs in time $n \cdot \mathrm{poly}(\lambda) + \mathrm{polylog}(\lambda, m, \log n)$.*
- *The verifier has randomness size $O(\log m + \mathrm{loglog} n + \log \lambda)$.*
- *The prover runs in time $T \cdot \mathrm{poly}(\lambda, m, \log T)$, and uses space $S \cdot \mathrm{poly}(\lambda, m, \log T)$, where T, S are the time and space required for verifying the language.*

4.1 Construction

In this section, we present the construction of a publicly-verifiable and adaptively sound succinct PCA with constant query complexity for any relation in NP. We begin by introducing relevant notations that will be used throughout the construction.

Notations. Fix a relation $\mathcal{R}$ in NP. Let M' be a deterministic Turing machine that can verify whether an instance of size n and a witness of size m belong to $\mathcal{R}$ in time $t = \mathrm{poly}(n)$. We denote M as a RAM machine that, given an an implicit input $x^{\mathsf{imp}} = \mathbb{x}$ and an explicit input $\mathbb{w}$, emulates $M'(\mathbb{x}, \mathbb{w})$. The machine will output 1 if and only if $M'(\mathbb{x}, \mathbb{w})$ accepts. The running time of M is t. The read-write memory of the machine is of size $S = t + n + m$. The local state of the machine includes the description of the state of the Turing machine and the last location read from memory, which requires at most $\mathrm{polylog}(t, n, m)$. We define the size of the local state of the machine to be $\mathsf{L_{st}} = |x^{\mathsf{exp}}| + \log |x^{\mathsf{imp}}| + \mathrm{polylog}(t, n, m) = m + \mathrm{polylog}(t, n, m)$.

Let $(\mathsf{RAM.G}, \mathsf{RAM.D}, \mathsf{RAM.P}, \mathsf{RAM.V})$ be a RAM Delegation scheme for the machine M, and let Enc be an efficiently encodable and decodable error correcting code ensemble with relative distance $\delta > 0$. Denote by Enc^{-1} the (poly-time) decoding algorithm for Enc. We require the decoding algorithm to work only for valid codewords. The language L' is defined as follows:

$$L' = \left\{ \left(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}\right), \mathsf{Enc}(\mathbb{w}||\pi) \mid \mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, \mathbb{w}, 1, \pi) \right\}$$

In other words, the language L' is the set of pairs of the following structure:

1. The first component consists of inputs known to RAM.V explicitly, namely, the common reference string crs and a digest of the implicit input.

2. The second component consists of inputs given to $\mathsf{RAM.V}$ by the prover, namely, the witness $\mathbb{w}$, and the $\mathsf{RAM\ Delegation}$ proof π. All of these are encoded by the error correcting code Enc.

We define $\mathsf{PCPP} = (\mathsf{PCPP.P}, \mathsf{PCPP.V})$ scheme to be a PCPP scheme for the language L'.

Construction 4.3. The construction of the PCA scheme is as follows:

- $\underline{\mathsf{PCA.G}(1^n, 1^m, 1^\lambda).}$
  1. Output $\mathsf{crs} \leftarrow \mathsf{RAM.G}(1^\lambda, t(n, m))$.
- $\underline{\mathsf{PCA.P}(\mathsf{crs}, \mathbb{x}, \mathbb{w}).}$
  1. Set $\pi_1 = \mathsf{RAM.P}(\mathsf{crs}, \mathbb{x}, \mathbb{w})$.
  2. Set $\mathsf{d}^{\mathsf{imp}} = \mathsf{RAM.D}(\mathsf{crs}, \mathbb{x})$.
  3. Set $\pi_2 = \mathsf{PCPP.P}((\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}), \mathsf{Enc}(\mathbb{w}||\pi_1))$.
  4. Output a proof $\pi = (\mathsf{Enc}(\mathbb{w}||\pi_1)||\pi_2)$.
- $\underline{\mathsf{PCA.V}^{\pi}(\mathsf{crs}, \mathbb{x}).}$
  1. Set $\mathsf{d}^{\mathsf{imp}} = \mathsf{RAM.D}(\mathsf{crs}, \mathbb{x})$.
  2. Parse π as $\mathsf{Enc}(\mathbb{w}||\pi_1)||\pi_2$.
  3. Output $\mathsf{PCPP.V}^{\mathsf{Enc}(\mathbb{w}||\pi_1), \pi_2}((\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}), \frac{\delta}{2})$.

4.2 Analysis

In what follows, we will provide proofs for completeness, efficiency, and soundness for the PCA construction in section Section 4.1.

Completeness. Follows directly from the completeness properties of the underlying encoding, $\mathsf{RAM\ Delegation}$ and PCPP schemes.

Efficiency. We start by defining the efficiency properties of the underlying $\mathsf{RAM\ Delegation}$ scheme. Let L_{st} be the size of the local state of M, let vt be the running time of $\mathsf{RAM.V}$, let $\mathsf{L}_{\mathsf{crs}}$ be the size of crs, and let L_{π_1} be the size of the proof π_1. By the $\mathrm{poly}(\lambda, \log t)$-verifier efficiency and the $\mathrm{poly}(\lambda, \log t)$-succinctness of the underlying $\mathsf{RAM\ Delegation}$ scheme, we get that for $\mathsf{L}_{\mathsf{st}} \geq |x^{\mathsf{exp}}| + \log |x^{\mathsf{imp}}|$ the following holds (Definition 3.9),

$$\mathsf{vt}, \mathsf{L}_{\mathsf{crs}}, \mathsf{L}_{\pi_1} \leq \mathrm{poly}(\lambda, \log t) \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}}) \ .$$

Given that $\mathsf{L}_{\mathsf{st}} = m + \mathrm{polylog}(t, n, m)$, we get that $\mathsf{L}_{\mathsf{st}} \geq |x^{\mathsf{exp}}| + \log |x^{\mathsf{imp}}|$. Therefore, by the above equation,

$$\begin{aligned} \mathsf{vt}, \mathsf{L}_{\mathsf{crs}}, \mathsf{L}_{\pi_1} &\leq \mathrm{poly}(\lambda, \log t) \cdot \mathrm{poly}(\log t, \log n, m) \\ &\leq \mathrm{poly}(\lambda, \log n, m) \ . \end{aligned} \tag{1}$$

Furthermore, we define the running time required for verifying the language L' to be the t'. By the efficiency of the underlying $\mathsf{RAM\ Delegation}$ scheme and the efficiency of the underlying encoding scheme, we get that,

$$t' = \mathsf{vt} + \mathrm{poly}(m, \mathsf{L}_{\pi_1}) \ .$$

By combining the above equation with Equation 1 we get that:

$$t' \leq \mathrm{poly}(\lambda, \log n, m) \ . \tag{2}$$

Now, we can proceed to prove the succinctness of our scheme.

- Proof size: By the efficiency of the underlying encoding scheme, and by the size of π_1 described in Equation 1, we get that,

$$|\mathsf{Enc}(\mathbb{w}||\pi_1)| \leq \mathrm{poly}(\lambda, \log n, m) \ . \tag{3}$$

By the running time of RAM.V described in Equation 1, combining with Theorem 3.4, we get that,

$$|\pi_2| \leq \mathrm{poly}(\lambda, \log n, m) \ . \tag{4}$$

Overall, by combining Equations 3, 4 we get that,

$$|\pi| = |(\mathsf{Enc}(\mathbb{w}||\pi_1)||\pi_2)| \leq \mathrm{poly}(\lambda, \log n, m) \ ,$$

as required.

- Prover complexity: We start by analyzing the running time of all the steps except for the RAM Delegation prover execution. The PCA prover first computes π_1, the RAM Delegation proof. Then, the PCA prover computes the digest of $\mathbb{x}$ using RAM.D. By the efficiency of the underlying RAM Delegation, this step takes time:

$$|\mathbb{x}| \cdot \mathrm{poly}(\lambda) = n \cdot \mathrm{poly}(\lambda) \ . \tag{5}$$

Finally, the prover computes π_2 by encoding $(\mathbb{w} \mid\mid \pi_1)$, followed by the computation of a PCPP proof for the language L'. Since the running time of verifying L' is t', this step takes $\mathrm{poly}(t')$ (Theorem 3.4). By equation Equation 2, we get that this step takes time:

$$\mathrm{poly}(\lambda, \log n, m) \ . \tag{6}$$

By Equation 1 we get that,

$$\mathrm{poly}(\mathsf{vt}, m, \mathsf{L}_{\pi_1}) \leq \mathrm{poly}(\lambda, \log n, m) \ . \tag{7}$$

By Equation 5 to Equation 7, we get that all the steps of the PCA prover, except for the execution of the RAM Delegation prover, takes time:

$$n \cdot \mathrm{poly}(\lambda) + \mathrm{poly}(\lambda, \log n, m) \ . \tag{8}$$

Overall, for a RAM Delegation prover with running time bounded by T, we get that the running time of the PCA prover is bounded by:

$$T + n \cdot \mathrm{poly}(\lambda) + \mathrm{poly}(\lambda, \log n, m) \ .$$

Regarding the space complexity of the PCA prover, it is determined by the space used by the $\mathsf{RAM\ Delegation}$ prover and the runtime of the remaining steps. Therefore, referring to equation 8, for a $\mathsf{RAM\ Delegation}$ prover with space complexity bounded by S, the space complexity of the PCA prover is bounded by:

$$S + n \cdot \mathrm{poly}(\lambda) + \mathrm{poly}(\lambda, \log n, m)$$

as required.

- Verifier running time: The PCA verifier starts by computing the digest of $\mathbb{x}$ using $\mathsf{RAM.D}$. By the efficiency of the underlying $\mathsf{RAM\ Delegation}$ scheme, this step takes time

  $$|\mathbb{x}| \cdot \mathrm{poly}(\lambda) = n \cdot \mathrm{poly}(\lambda) \ . \tag{9}$$

  Then, the PCA verifier validates the PCPP proof. By Theorem 3.4, verifying the PCPP proof takes time:

  $$\mathrm{polylog}(\mathsf{vt}, m, \mathsf{L}_{\pi_1}) \ . \tag{10}$$

  By Equation 1 we get that,

  $$\mathrm{polylog}(\mathsf{vt}, m, \mathsf{L}_{\pi_1}) \leq \mathrm{polylog}(\lambda, m, \log n) \ . \tag{11}$$

  Overall, by combining Equation 9 to Equation 11 we get that the verifier runs in time $n \cdot \mathrm{poly}(\lambda) + \mathrm{polylog}(\lambda, m, \log n)$, as required.

- Verifier's randomness size: The PCA verifier only uses randomness required for the PCPP verifier. Therefore, by Theorem 3.4, the randomness complexity is $O(\log t')$. By Equation 2 we get that the randomness complexity is

  $$O(\log(\mathrm{poly}(\lambda, \log n, m))) \leq O(\log(\lambda) + \mathrm{loglog}(n) + \log(m)),$$

  as requires.

- Query complexity: The PCA verifier only queries its oracle when running the PCPP verifier. Therefore, for a fixed proximity parameter, by Theorem 3.4, the query complexity is $O(1)$, as required.

Soundness. Fix some $n, m \in \mathbb{N}$, security parameter $\lambda \in \mathbb{N}$, and deterministic malicious prover $\tilde{\mathsf{P}}$ of size $s(\lambda) = \mathrm{poly}(\lambda)$.

Fix a crs generated by $\mathsf{PCA.G}(1^n, 1^m, 1^\lambda)$, a malicious prover message $(\mathbb{x}, \tilde{\pi}) = \tilde{\mathsf{P}}(\mathsf{crs})$ such that $\mathcal{R}(\mathbb{x}) = \emptyset$, and a digest of the input $\mathsf{d}^{\mathsf{imp}} = \mathsf{RAM.D}(\mathsf{crs}, \mathbb{x})$. Parse the proof as $\tilde{\pi} = \tilde{\tau}_1 || \tilde{\tau}_2$. Consider the case where:

$$\Delta(\tilde{\tau}_1, \mathsf{Image}(\mathsf{Enc})) \geq \frac{\delta}{2} \ .$$

In this case, since $L'(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}) \subseteq \mathsf{Image}(\mathsf{Enc})$, it holds that:

$$\Delta(\tilde{\tau}_1, L'(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}})) \geq \frac{\delta}{2} \ .$$

By the soundness property of the PCPP, we get that:

$$\Pr\left[\ \mathsf{PCPP.V}^{\tilde{\tau}_1,\tilde{\tau}_2}((\mathsf{crs},\mathsf{d}^{\mathsf{imp}}),\tfrac{\delta}{2})=1\ \middle|\ \begin{array}{l}(\mathbb{x},\tilde{\pi})\leftarrow\tilde{\mathsf{P}}(\mathsf{crs})\\ \mathsf{d}^{\mathsf{imp}}\leftarrow\mathsf{RAM.D}(\mathsf{crs},\mathbb{x})\\ \tilde{\pi}:=\tilde{\tau}_1\|\tilde{\tau}_2\\ \Delta(\tilde{\tau}_1,L'(\mathsf{crs},\mathsf{d}^{\mathsf{imp}}))\geq\tfrac{\delta}{2}\end{array}\right]<\frac{1}{2}\ .$$

By the above equation, for the case where $\Delta(\tilde{\tau}_1,\mathsf{Image}(\mathsf{Enc}))\geq\frac{\delta}{2}$, the following holds:

$$\Pr\left[\ \mathsf{PCA.V}^{\tilde{\pi}}(\mathsf{crs},\mathbb{x})=1\ \wedge\ \mathcal{R}(\mathbb{x})\ \middle|\ (\mathbb{x},\tilde{\pi})\leftarrow\tilde{\mathsf{P}}(\mathsf{crs})\ \right]<\frac{1}{2}\ ,$$

as required.

Therefore, moving forward, we will only consider the case where:

$$\Delta(\tilde{\tau}_1,\mathsf{Image}(\mathsf{Enc}))<\frac{\delta}{2}\ .$$

In this case, the prefix $\tilde{\tau}_1$ can be decoded efficiently into $\tilde{\mathbb{w}}\|\tilde{\pi}_1=\mathsf{Enc}^{-1}(\tilde{\tau}_1)$.

Given a crs we define the event BAD_b to be the following event:

$$\mathsf{BAD}_b:=\left[\ \begin{array}{l}\mathsf{RAM.V}(\mathsf{crs},\mathsf{d}^{\mathsf{imp}},\tilde{\mathbb{w}},1,\tilde{\pi}_1)=b\\ \mathcal{R}(\mathbb{x})=\emptyset\end{array}\ \middle|\ \begin{array}{l}(\mathbb{x},\tilde{\pi})\leftarrow\tilde{\mathsf{P}}(\mathsf{crs})\\ \mathsf{d}^{\mathsf{imp}}\leftarrow\mathsf{RAM.D}(\mathsf{crs},\mathbb{x})\\ \tilde{\pi}:=\tilde{\tau}_1\|\tilde{\tau}_2\\ \tilde{\mathbb{w}}\|\tilde{\pi}_1\leftarrow\mathsf{Enc}^{-1}(\tilde{\tau}_1)\end{array}\right]\ .$$

In other words, the occurrence of event BAD_1 implies that the proof $\tilde{\pi}$ consists of a false witness $\tilde{\mathbb{w}}$ and an accepting $\mathsf{RAM\ Delegation}$ proof $\tilde{\pi}_1$. We'll argue that (1) the probability of sampling a PCA common reference string such that the event BAD_1 occurs is negligible, and (2) the probability of sampling a PCA common reference string such that the event BAD_0 occurs and that the PCA verifier accepts is at most $\frac{1}{2}$. Combining both claims will give us the required soundness guarantee for our PCA.

By the soundness property of the underlying $\mathsf{RAM\ Delegation}$ scheme, we get that:

$$\Pr\left[\ \mathsf{BAD}_1\ \middle|\ \mathsf{crs}\leftarrow\mathsf{PCA.G}(1^n,1^m,1^\lambda)\ \right]\leq\mathsf{negl}(\lambda)\ .$$

Otherwise, we could use $(\mathbb{x},\tilde{\pi})\leftarrow\tilde{\mathsf{P}}(\mathsf{crs})$ to construct an adversary for the $\mathsf{RAM\ Delegation}$ scheme that breaks the weak soundness with the same probability that the event BAD_1 occurs. The adversary for the $\mathsf{RAM\ Delegation}$ scheme will extract the witness $\tilde{\mathbb{w}}$ and the proof $\tilde{\pi}_1$ from $\tilde{\pi}$. Then, the adversary will generate an honest proof π_0 for the statement that $M(\mathbb{x},\mathbb{w})=0$ (which can be done by the completeness of the $\mathsf{RAM\ Delegation}$ scheme). The adversary will output $(x=(\mathbb{x},\mathbb{w}),\pi_0,\pi_1=\tilde{\pi}_1)$.

Next, we want to bound the following probability of BAD_0. By the definition of BAD_0, this means that $\mathsf{RAM.V}$ rejects its input. Therefore, the claim proven by $\tilde{\tau}_2$ is false. In this case, by the soundness of the PCPP verifier for L' it holds that the probability of $\mathsf{PCPP.V}$ accepting input $((\mathsf{crs},\mathsf{d}^{\mathsf{imp}}),\mathsf{Enc}(\tilde{\mathbb{w}}\|\tilde{\pi}_1))$ with proof $\tilde{\tau}_2$ is at most $\frac{1}{2}$. Therefore, we get that:

$$\Pr\left[\ \begin{array}{l}\mathsf{PCA.V}^{\tilde{\pi}}(\mathsf{crs},\mathbb{x})=1\ \wedge\ \mathcal{R}(\mathbb{x})\\ \mathsf{BAD}_0\end{array}\ \middle|\ \mathsf{crs}\leftarrow\mathsf{PCA.G}(1^n,1^m,1^\lambda)\ \right]<\frac{1}{2}\ .$$

Overall, with all but $\mathsf{negl}(\lambda)$ probability over the choice of crs, the probability of $\mathsf{PCPP.V}$ accepting $\mathbb{x}$ with proof $\tilde{\pi}$ is at most $\frac{1}{2}$.

5 Preliminaries - Hash Families

In this section, we present a collection of diverse hash family definitions. Each hash family we introduce includes the desirable property of local opening and a unique combination of soundness and efficiency attributes.

5.1 Hash Tree

In this section we recall the definition of a hash tree scheme taken from [KPY19]. A hash tree scheme $\mathsf{HT} = (\mathsf{HT.G}, \mathsf{HT.H}, \mathsf{HT.R}, \mathsf{HT.VR})$ has the following syntax:

$\mathsf{HT.G}(\lambda) \to \mathsf{hk}$. The probabilistic setup algorithm $\mathsf{HT.G}$ takes as input the security parameter λ and outputs a hash key hk.

$\mathsf{HT.H}(\mathsf{hk}, x) \to (\mathsf{T}, \mathsf{rt})$. The deterministic hash algorithm $\mathsf{HT.H}$ takes as input a key hk, and an input $x \in \{0,1\}^*$. It outputs a hash tree T, and a root rt.

$\mathsf{HT.R}(\mathsf{T}, j) \to (b, \pi)$. The deterministic read algorithm $\mathsf{HT.R}$ takes as input a tree T and an index j. It outputs a bit b, and a proof π.

$\mathsf{HT.W}(\mathsf{T}, j, b) \to (\mathsf{T}', \mathsf{rt}', \pi)$ The deterministic write algorithm $\mathsf{HT.W}$ takes as input a tree T, an index j, and a bit b. It outputs a new tree T', a new root rt' and a proof π.

$\mathsf{HT.VR}(\mathsf{hk}, \mathsf{rt}, j, b, \pi) \to 0/1$. The deterministic verify read algorithm $\mathsf{HT.VR}$ takes as input a key hk, a root rt, an index j, a bit b, and a proof π. It outputs an acceptance bit.

$\mathsf{HT.VW}(\mathsf{hk}, \mathsf{rt}, j, b, \mathsf{rt}', \pi) \to 0/1$. The deterministic verify write algorithm $\mathsf{HT.VW}$ takes as input a key hk, a root rt, an index j, a bit b, a new root rt', and a proof π. It outputs an acceptance bit.

Definition 5.1 (Hash Tree). *A secure hash tree scheme satisfies the following requirements:*

Completeness of read. *For every $\lambda \in \mathbb{N}$, $N \le 2^\lambda$, $x \in \{0,1\}^N$, and $j \in [N]$:*

$$\Pr\left[\begin{array}{l} \mathsf{HT.VR}(\mathsf{hk}, \mathsf{rt}, j, b, \pi) = 1 \\ x[j] = b \end{array} \;\middle|\; \begin{array}{l} \mathsf{hk} \leftarrow \mathsf{HT.G}(\lambda) \\ (\mathsf{T}, \mathsf{rt}) \leftarrow \mathsf{HT.H}(\mathsf{hk}, x) \\ (b, \pi) \leftarrow \mathsf{HT.R}(\mathsf{T}, j) \end{array}\right] = 1 \ .$$

Completeness of write. *For every $\lambda \in \mathbb{N}$, $N \le 2^\lambda$, $x \in \{0,1\}^N$, $j \in [N]$ and $b \in \{0,1\}$, let x' be the string x with its j-th location set to b. We have that:*

$$\Pr\left[\begin{array}{l} \mathsf{HT.VW}(\mathsf{hk}, \mathsf{rt}, j, b, \mathsf{rt}', \pi) = 1 \\ (\mathsf{T}', \mathsf{rt}') = \mathsf{HT.H}(\mathsf{hk}, x') \end{array} \;\middle|\; \begin{array}{l} \mathsf{hk} \leftarrow \mathsf{HT.G}(\lambda) \\ (\mathsf{T}, \mathsf{rt}) \leftarrow \mathsf{HT.H}(\mathsf{hk}, x) \\ (\mathsf{T}', \mathsf{rt}', \pi) \leftarrow \mathsf{HT.W}(\mathsf{T}, j, b) \end{array}\right] = 1 \ .$$

Efficiency. *In the completeness experiment above:*

- *The setup, read, write, verify read, and verify write algorithms run in time* $\log N \cdot \mathrm{poly}(\lambda)$.
- *The hash algorithm runs in time* $N \cdot \mathrm{poly}(\lambda)$

- *The length of the root is* $\mathrm{poly}(\lambda)$.
- *The length of the proof is* $\log N \cdot \mathrm{poly}(\lambda)$.

Soundness of read. *For every* $\mathrm{poly}(\lambda)$*-size adversary A there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:*

$$\Pr\left[\begin{array}{l} b_1 \neq b_2 \\ \mathsf{HT.VR}(\mathsf{hk}, \mathsf{rt}, j, b_1, \pi_1) = 1 \\ \mathsf{HT.VR}(\mathsf{hk}, \mathsf{rt}, j, b_2, \pi_2) = 1 \end{array} \middle| \begin{array}{l} \mathsf{hk} \leftarrow \mathsf{HT.G}(\lambda) \\ (\mathsf{rt}, j, b_1, \pi_1, b_2, \pi_2) \leftarrow A(\mathsf{hk}) \end{array}\right] \leq \mu(\lambda) \ .$$

Soundness of write. *For every* $\mathrm{poly}(\lambda)$*-size adversary A there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:*

$$\Pr\left[\begin{array}{l} \mathsf{rt}_1 \neq \mathsf{rt}_2 \\ \mathsf{HT.VW}(\mathsf{hk}, \mathsf{rt}, j, b, \mathsf{rt}_1, \pi_1) = 1 \\ \mathsf{HT.VW}(\mathsf{hk}, \mathsf{rt}, j, b, \mathsf{rt}_2, \pi_2) = 1 \end{array} \middle| \begin{array}{l} \mathsf{hk} \leftarrow \mathsf{HT.G}(\lambda) \\ (\mathsf{rt}, j, b, \mathsf{rt}_1, \pi_1, \mathsf{rt}_2, \pi_2) \leftarrow A(\mathsf{hk}) \end{array}\right] \leq \mu(\lambda) \ .$$

Theorem 5.2 ([Mer87])**.** *A hash-tree scheme can be constructed from any family of collision-resistant hash functions.*

5.2 Somewhere Extractable Hash

In this section we recall the definition of a hash tree scheme. The definition is taken from. A somewhere extractable hash

$$\mathsf{SEH} = (\mathsf{SEH.G}, \mathsf{SEH.H}, \mathsf{SEH.O}, \mathsf{SEH.V}, \mathsf{SEH.E})$$

has the following syntax:

$\mathsf{SEH.G}(1^\lambda, N, i) \to (\mathsf{hk}, \mathsf{td})$**.** This is a randomized algorithm that takes as input a security parameter 1^λ, input size N, and an index $i \in [N]$. It outputs a hash key hk, and a trapdoor td.

$\mathsf{SEH.O}(\mathsf{hk}, x, j) \to (b, \rho)$**.** This is a deterministic poly-time algorithm that takes as input a hash key hk, an input x, and an index $j \in [N]$. It outputs a bit b, and an opening ρ.

$\mathsf{SEH.V}(\mathsf{hk}, \mathsf{v}, j, b, \rho) \to 0/1$**.** This is a deterministic poly-time algorithm that takes as input a hash key hk, a hash value v, an index $j \in [N]$, a bit b, and an opening ρ. It outputs an acceptance bit.

$\mathsf{SEH.E}(\mathsf{td}, \mathsf{v}) \to b$**.** This is a deterministic poly-time algorithm that takes as input a trapdoor hk, and a hash value v. It outputs a bit b.

Definition 5.3 (SEH)**.** *An $\mathcal{L}$-succinct SEH hash family*

$$(\mathsf{SEH.G}, \mathsf{SEH.H}, \mathsf{SEH.O}, \mathsf{SEH.V}, \mathsf{SEH.E}) \ ,$$

is required to satisfy the following properties:

$\mathcal{L}$-succinct. *The runtime of* $\mathsf{SEH.G}(1^\lambda, N, i)$ *is bounded by $\mathcal{L}(N, \lambda)$, and the runtime of* $\mathsf{SEH.V}$ *(and hence the size of* v, ρ*) is at most $\mathcal{L}(N, \lambda)$.*

Opening completness. *For any $\lambda \in \mathbb{N}$, any $N \leq 2^\lambda$, any indices $i, j \in [N]$, and $x \in \{0,1\}^N$,*

$$\Pr\left[\begin{array}{l} b = x_j \\ \mathsf{SEH.V}(\mathsf{hk}, \mathsf{v}, i, b, \rho) = 1 \end{array} \middle| \begin{array}{l} (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{SEH.G}(1^\lambda, N, i) \\ (\mathsf{v}, \cdot) \leftarrow \mathsf{SEH.H}(\mathsf{hk}, x) \\ (b, \rho) \leftarrow \mathsf{SEH.O}(\mathsf{hk}, x, j) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda) \ .$$

$$\Pr\left[\begin{array}{l} \rho_j = \rho \end{array} \middle| \begin{array}{l} (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{SEH.G}(1^\lambda, N, i) \\ (\mathsf{v}, \rho_1, \dots, \rho_{|x|}) \leftarrow \mathsf{SEH.H}(\mathsf{hk}, x) \\ (b, \rho) \leftarrow \mathsf{SEH.O}(\mathsf{hk}, x, j) \end{array}\right] = 1 \ .$$

Index hiding. *For any poly-size adversary $A = (A_1, A_2)$, and indexes $i_0, i_1 \in \mathbb{N}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\ A_2(\mathsf{hk}) = b \ \middle| \begin{array}{l} (i_0, i_1, N) \leftarrow A_1(1^\lambda), b \leftarrow \{0,1\} \\ (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{SEH.G}(1^\lambda, N, i_b) \end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda) \ .$$

Somewhere statistical (resp. computational) extractability *for any all powerful (resp. poly-size) adversary $A = (A_1, A_2)$ there exists a negligible function $\mathsf{negl}(\cdot)$ such that for every $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{l} b \neq \mathsf{SEH.E}(\mathsf{td}, \mathsf{v}) \\ \mathsf{SEH.V}(\mathsf{hk}, \mathsf{v}, i, b, \rho) = 1 \end{array} \middle| \begin{array}{l} (i, N) \leftarrow A_1(1^\lambda) \\ (\mathsf{hk}, \mathsf{td}) \leftarrow \mathsf{SEH.G}(1^\lambda, N, i) \\ (\mathsf{v}, b, \rho) \leftarrow A_2(\mathsf{hk}) \end{array}\right] \leq \mathsf{negl}(\lambda) \ .$$

We say that a SEH *scheme has succinct local openings if*

- SEH.G *runs in time* $\mathrm{poly}(\lambda)$.
- $\mathcal{L}(N, \lambda) = \mathrm{poly}(\lambda, \log N)$.

Remark 5.4 (m-SEH)**.** As remarked in [KLVW22], any SEH hash family can be converted into one that is extractable on m indices $i_1, \dots, i_m$ by simply running all the algorithms in parallel m times. Under this transformation, if the original SEH family had ℓ-local openings, the new family will have $\ell \cdot m$-local openings. Thus, more generally, we think of SEH.G as taking as input $(1^\lambda, N, s)$ where $I \subseteq [N]$, in which case $\mathsf{SEH.E}(\mathsf{td}, \mathsf{v})$ outputs $|I|$ bits $(b_i)_{i \in I}$. We sometimes refer to this an m-SEH hash family, and sometimes we omit m, and simply refer to it as an SEH hash family.

Theorem 5.5 ([KLVW22])**.** SEH *schemes with succinct local openings exist under any of the following cryptographic hardness assumptions: (1) $O(1)$-*LIN *assumption; (2) The* LWE *assumption; or (3) Decisional Diffie-Hellman (*DDH*).*

Definition 5.6 (Two-mode SEH)**.** *We say that a* SEH *is a two-mode* SEH *if for a fixed input length $N = 2$ and flexible block size m, the output length of* SEH.H *is $m \cdot (1 + 1/\Omega(\lambda)) + \mathrm{poly}(\lambda)$.*

Theorem 5.7 ([KLVW22])**.** *Two-mode* SEH *schemes exist under any of the following cryptographic hardness assumptions: (1) $O(1)$-*LIN *assumption; (2) The* LWE *assumption; or (3) Decisional Diffie-Hellman (*DDH*).*

5.3 Sparse Hash Tree

In this section we recall the definition of a hash tree scheme ([DPP16]). A sparse hash tree scheme $\mathsf{SparseHT} = (\mathsf{SparseHT.G}, \mathsf{SparseHT.H}, \mathsf{SparseHT.R}, \mathsf{SparseHT.VR})$ has the same syntax as for the HT scheme except for the following syntax:

$\mathsf{SparseHT.H}(\mathsf{hk}, N, I) \to (\mathsf{T}, \mathsf{rt})$. The deterministic hash algorithm $\mathsf{SparseHT.H}$ takes as input a key hk, an input size $N \in \mathbb{N}$, and a set of indexes $I \subseteq [N]$. It outputs a description of a hash tree T, and a root rt.

Definition 5.8 (Sparse Hash Tree). *A secure hash tree scheme satisfies the following requirements:*

Completeness of read. *For every security parameter $\lambda \in \mathbb{N}$, input size $N \in \mathbb{N}$, input $x \in \{0,1\}^N$, a set of indexes $I = \{j \in [N] \mid x[j] = 1\}$, and index $j \in [N]$:*

$$\Pr\left[\begin{array}{l|l} \mathsf{SparseHT.VR}(\mathsf{hk}, \mathsf{rt}, j, b, \pi) = 1 & \mathsf{hk} \leftarrow \mathsf{SparseHT.G}(\lambda) \\ x[j] = b & (\mathsf{T}, \mathsf{rt}) \leftarrow \mathsf{SparseHT.H}(\mathsf{hk}, N, I) \\ & (b, \pi) \leftarrow \mathsf{SparseHT.R}(\mathsf{T}, j) \end{array}\right] = 1 \ .$$

Completeness of write. *For every security parameter $\lambda \in \mathbb{N}$, input size $N \in \mathbb{N}$, input $x \in \{0,1\}^N$, a set of indexes $I = \{j \in [N] \mid x[j] = 1\}$, index $j \in [N]$, $b \in \{0,1\}$, let x' be the string x with its j-th location set to b, and let $I' = \{j \in [N] \mid x'[j] = 1\}$. We have that:*

$$\Pr\left[\begin{array}{l|l} \mathsf{SparseHT.VW}(\mathsf{hk}, \mathsf{rt}, j, b, \mathsf{rt}', \pi) = 1 & \mathsf{hk} \leftarrow \mathsf{SparseHT.G}(\lambda) \\ (\mathsf{T}', \mathsf{rt}') = \mathsf{SparseHT.H}(\mathsf{hk}, N, I') & (\mathsf{T}, \mathsf{rt}) \leftarrow \mathsf{SparseHT.H}(\mathsf{hk}, N, I) \\ & (\mathsf{T}', \mathsf{rt}', \pi) \leftarrow \mathsf{SparseHT.W}(\mathsf{T}, j, b) \end{array}\right] = 1 \ .$$

Efficiency. *In the completeness experiment above:*

- *The setup, read, write, verify read, and verify write algorithms run in time* $\log N \cdot \mathrm{poly}(\lambda)$.
- *The hash algorithm runs in time* $|I| \cdot \log N \cdot \mathrm{poly}(\lambda)$.
- *The length of the root is* $\mathrm{poly}(\lambda)$.
- *The length of the proof is* $\log N \cdot \mathrm{poly}(\lambda)$.

Soundness of read. *For every* $\mathrm{poly}(\lambda)$*-size adversary A there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:*

$$\Pr\left[\begin{array}{l|l} b_1 \neq b_2 & \\ \mathsf{SparseHT.VR}(\mathsf{hk}, \mathsf{rt}, j, b_1, \pi_1) = 1 & \mathsf{hk} \leftarrow \mathsf{SparseHT.G}(\lambda) \\ \mathsf{SparseHT.VR}(\mathsf{hk}, \mathsf{rt}, j, b_2, \pi_2) = 1 & (\mathsf{rt}, j, b_1, \pi_1, b_2, \pi_2) \leftarrow A(\mathsf{hk}) \end{array}\right] \leq \mu(\lambda) \ .$$

Soundness of write. *For every* $\mathrm{poly}(\lambda)$*-size adversary A there exists a negligible function μ such that for every $\lambda \in \mathbb{N}$:*

$$\Pr\left[\begin{array}{l|l} \mathsf{rt}_1 \neq \mathsf{rt}_2 & \\ \mathsf{SparseHT.VW}(\mathsf{hk}, \mathsf{rt}, j, b, \mathsf{rt}_1, \pi_1) = 1 & \mathsf{hk} \leftarrow \mathsf{SparseHT.G}(\lambda) \\ \mathsf{SparseHT.VW}(\mathsf{hk}, \mathsf{rt}, j, b, \mathsf{rt}_2, \pi_2) = 1 & (\mathsf{rt}, j, b, \mathsf{rt}_1, \pi_1, \mathsf{rt}_2, \pi_2) \leftarrow A(\mathsf{hk}) \end{array}\right] \leq \mu(\lambda) \ .$$

Theorem 5.9. *If there exists a secure family of collision-resistant hash functions, then there exists a secure sparse hash tree scheme.*

Proof Sketch. We construct a Merkle tree with an efficient representation. Fix any $N \in \mathbb{N}$, and let $\mathsf{d} := \log N$ be the depth of the tree. We observe that when $I = \emptyset$, all the leaf nodes in the tree have the value $H(0)$. As each node in the tree is computed by hashing its two child nodes, it follows that all the nodes within a given layer of the tree will have the same value. We leverage this information, and we compute only one value for each layer in the tree. We define $v_0 = \mathsf{H}(0)$, and for each $i \in [\mathsf{d}]$ we define $v_i := \mathsf{H}(v_{i-1} \mid\mid v_{i-1})$. This computation can be done using $\mathrm{poly}(\lambda, \log N)$ time and space.

Our next step is to address the values in the locations I. We start with just a null root that does not point to anything. At every step of the algorithm we will think of the tree as a virtual complete binary tree, where a null node at level i means that the values of that node is v_i. We now gradually add values to the tree according to I. The main idea is to use the well-known algorithm for updating a Merkle tree. The values of the path from the leaf to the root can be computed in time $\mathrm{poly}(\lambda, \log N)$. We update the values of the path from the leaf to the root, while adding the nodes that are absent from that path.

Overall, the running time for constructing the tree is $(|I| + 1) \cdot \log N \cdot \mathrm{poly}(\lambda)$.

□

6 RAM Delegation from BARGs

In this section we construct a $\mathrm{poly}(\lambda, \log t)$-succinct RAM Delegation scheme for any read-write RAM machine. The construction is introduced in Section 6.1. We then proceed to prove the completeness, efficiency, and soundness of the scheme in Section 6.2. By combining these results, we establish the following theorem.

Theorem 6.1. *Assume there exists a secure hash tree* HT*, a secure sparse hash tree* SparseHT*, a secure somewhere extractable hash* SEH *with succinct local opening, and an $\mathcal{L}$-succinct somewhere extractable batch argument* seBARG *for* BatchCSAT*, then there exists an $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\lambda, \log t)$-succinct* RAM Delegation *scheme with strong soundness for any read-write* RAM *machine.*

By combining Theorem 6.1 with Theorems 5.2, 5.9, 5.5, and 3.8 we get the following corollary.

Corollary 6.2. *There exist $\mathrm{poly}(\lambda, \log t)$-succinct* RAM Delegation *scheme for any read-write* RAM *machine under any of the following assumptions:*

1. *The $O(1)$-*LIN *assumption on a pair of cryptographic groups with efficient bilinear map.*

2. *The hardness of Learning with errors* (LWE) *problem against polynomial time adversaries.*

3. *The sub-exponential Decisional Diffie-Hellman (*DDH*) assumptions.*

6.1 Construction

In this section, we present the construction of a $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\lambda, \log t)$-succinct RAM Delegation scheme with strong soundness for read-write RAM machines. To lay the foundation, we first provide a formal definition of a single step in a RAM machine. We then introduce relevant notations that will facilitate the subsequent construction process.

The RAM machine. Fix RAM machine M. We assume for simplicity, and without loss of generality that in every step RAM reads from one input location, then reads from one memory location, and then writes to one memory location. Moreover, without loss of generality, we assume that M zeros its memory before accepting or rejecting. Let $(\mathsf{StepInputRead}, \mathsf{StepMemRead}, \mathsf{StepMemWrite})$ be the following deterministic polynomial-time algorithms:

- $\mathsf{StepInputRead}(\mathsf{st}) \rightarrow (\mathsf{type}, \ell^{\mathsf{inp}})$: Given an input st, the algorithm $\mathsf{StepInputRead}$ outputs an input type $\mathsf{type} \in \{\mathsf{imp}, \mathsf{exp}\}$ together with a location ℓ^{inp} such that RAM in state st reads the type input at location ℓ^{inp}.

- $\mathsf{StepMemRead}(\mathsf{st}, z^{\mathsf{inp}}) \rightarrow \ell^{\mathsf{mr}}$: Given an input $(\mathsf{st}, z^{\mathsf{inp}})$, the algorithm $\mathsf{StepMemRead}$ outputs a memory location ℓ^{mr} such that RAM in state st, after reading the bit z^{inp} from the input, reads from location ℓ^{mr} in the memory.

- $\mathsf{StepMemWrite}(\mathsf{st}, z^{\mathsf{inp}}, z^{\mathsf{mr}}) \rightarrow (z^{\mathsf{mw}}, \ell^{\mathsf{mw}}, \mathsf{st}')$: Given an input $(\mathsf{st}, z^{\mathsf{inp}}, z^{\mathsf{mr}})$, the algorithm $\mathsf{StepMemWrite}$ outputs a bit z^{mw}, memory location ℓ^{mw} and a state st', such that RAM in state st, after reading the bit z^{inp} from the input and reading the bit z^{mr} from the memory, writes the bit z^{mw} to location ℓ^{mw} in the memory and transitions to state st'.

Notations. Let M be some RAM machine, let $\mathsf{L_{st}}$ be its local state size, and let $N = |x^{\mathsf{imp}}|$ be the implicit input size. We denote $\mathsf{L_{rt}}$ to be the root size of a Merkle tree using $\mathsf{HT.G}(1^\lambda)$. Note that $\mathsf{L_{rt}} = \mathrm{poly}(\lambda)$. Furthermore, we define $C = C_{\mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, x^{\mathsf{exp}}, \mathsf{d}^{\mathsf{imp}}, \mathsf{rt}_h, \mathsf{rt_{st}}, h_0, out}$ to be the circuit that on input $(i, \mathbb{w}_i)$ outputs 1 if and only if the following conditions hold:

1. Parse $\mathbb{w}_i := (\mathsf{st}, \mathsf{st}', h, h', z^{\mathsf{mr}}, \mathsf{auth}^{\mathsf{mr}}, \mathsf{auth}^{\mathsf{mw}}, z^{\mathsf{inp}}, \mathsf{auth}^{\mathsf{inp}}, \rho_{\mathsf{st}}, \rho_{\mathsf{st}'}, \rho_h, \rho_{h'})$.

2. Check that the initial and end state, and the initial and end memory hash are correct:
   (a) If $i = 1$ check that st is the initial state.
   (b) If $i = t$ and $out = 1$ then check that st' is an accepting state.
   (c) If $i = t$ and $out = 0$ then check that st' is a rejecting state.
   (d) If $i = 1$ or $i = t$ then check that $h = h_0$. (we assume WLOG that M zeros its memory).

3. Check that the state and memory hash are consistent with the hash values:
   (a) For all $i \in [t]$, we define $I_i^{\mathsf{st}} = \{(i-1) \cdot \mathsf{L_{st}} + 1, \ldots, i \cdot \mathsf{L_{st}}\}$.
   (b) For all $i \in [t]$, we define $I_i^{h} = \{(i-1) \cdot \mathsf{L_{rt}} + 1, \ldots, i \cdot \mathsf{L_{rt}}\}$.
   (c) If $i > 1$ then check that $1 = \mathsf{SEH.V}(\mathsf{SEH.hk}_1, \mathsf{rt_{st}}, I_{i-1}^{\mathsf{st}}, \mathsf{st}, \rho_{\mathsf{st}})$.
   (d) If $i > 1$ then check that $1 = \mathsf{SEH.V}(\mathsf{SEH.hk}_2, \mathsf{rt}_h, I_{i-1}^{h}, h, \rho_h)$.
   (e) Check that $1 = \mathsf{SEH.V}(\mathsf{SEH.hk}_1, \mathsf{rt_{st}}, I_i^{\mathsf{st}}, \mathsf{st}', \rho_{\mathsf{st}'})$.
   (f) Check that $1 = \mathsf{SEH.V}(\mathsf{SEH.hk}_2, \mathsf{rt}_h, I_i^{h}, h', \rho_{h'})$.

4. Check that M transitions from (st, h) to (st', h') correctly:
   (a) Set $(\mathsf{type}, \ell^{\mathsf{inp}}) \leftarrow \mathsf{StepInputRead}(\mathsf{st})$.
   (b) Set $\ell^{\mathsf{mr}} \leftarrow \mathsf{StepMemRead}(\mathsf{st}, z^{\mathsf{inp}})$.
   (c) Set $(z^{\mathsf{mw}}, \ell^{\mathsf{mw}}, \mathsf{st}'') \leftarrow \mathsf{StepMemWrite}(\mathsf{st}, z^{\mathsf{inp}}, z^{\mathsf{mr}})$.
   (d) Check that $\mathsf{st}' = \mathsf{st}''$.
   (e) Check that $1 = \mathsf{SparseHT.VR}(\mathsf{SparseHT.hk}, h, \ell^{\mathsf{mr}}, z^{\mathsf{mr}}, \mathsf{auth}^{\mathsf{mr}})$.
   (f) Check that $1 = \mathsf{SparseHT.VW}(\mathsf{SparseHT.hk}, h, \ell^{\mathsf{mw}}, z^{\mathsf{mw}}, h', \mathsf{auth}^{\mathsf{mw}})$.

(g) If $\mathsf{type} = \mathsf{imp}$ then check that $1 = \mathsf{HT.VR}(\mathsf{HT.hk}, \mathsf{d}^{\mathsf{imp}}, \ell^{\mathsf{inp}}, z^{\mathsf{mw}}, \mathsf{auth}^{\mathsf{inp}})$.

(h) If $\mathsf{type} = \mathsf{exp}$ then check that $z^{\mathsf{inp}} = (x^{\mathsf{exp}})_{\ell^{\mathsf{inp}}}$.

Construction 6.3. The construction of the RAM Delegation scheme is as follows:

<u>$\mathsf{RAM.G}(1^\lambda, t)$:</u>

```
1. Set $i := 1$ (arbitrarily).
2. Set $(\mathsf{BARG.crs}, \mathsf{BARG.td}) \leftarrow \mathsf{BARG.G}(1^\lambda, t, \mathsf{L}_C, i)$, where $\mathsf{L}_C = |C|$.
3. Set $\mathsf{HT.hk} \leftarrow \mathsf{HT.G}(\lambda)$.
4. Set $\mathsf{SparseHT.hk} \leftarrow \mathsf{SparseHT.G}(\lambda)$.
5. Sample $(\mathsf{SEH.hk}_1, \mathsf{SEH.td}_1) \leftarrow \mathsf{SEH.G}(1^\lambda, t \cdot \mathsf{L}_{\mathsf{st}}, I_i^{\mathsf{st}})$ where $I_i^{\mathsf{st}} = \{(i-1) \cdot \mathsf{L}_{\mathsf{st}} + 1, \ldots, i \cdot \mathsf{L}_{\mathsf{st}}\}$.
6. Sample $(\mathsf{SEH.hk}_2, \mathsf{SEH.td}_2) \leftarrow \mathsf{SEH.G}(1^\lambda, t \cdot \mathsf{L}_{\mathsf{rt}}, I_i^h)$ where $I_i^h = \{(i-1) \cdot \mathsf{L}_{\mathsf{rt}} + 1, \ldots, i \cdot \mathsf{L}_{\mathsf{rt}}\}$.
7. Output $\mathsf{crs} := (\mathsf{BARG.crs}, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, t)$.
```

<u>$\mathsf{RAM.D}(\mathsf{crs}, x^{\mathsf{imp}})$:</u>

```
1. Parse $\mathsf{crs} := (\mathsf{BARG.crs}, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, t)$.
2. Output $\mathsf{d}^{\mathsf{imp}}$ where $(\cdot, \mathsf{d}^{\mathsf{imp}}) := \mathsf{HT.H}(\mathsf{HT.hk}, x^{\mathsf{imp}})$.
```

<u>$\mathsf{RAM.P}(\mathsf{crs}, x^{\mathsf{imp}}, x^{\mathsf{exp}})$:</u>

```
1. Parse $\mathsf{crs} = (\mathsf{BARG.crs}, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, t)$.
2. Set $(\mathsf{T}^{\mathsf{imp}}, \mathsf{d}^{\mathsf{imp}}) := \mathsf{HT.H}(\mathsf{HT.hk}, x^{\mathsf{imp}})$.
3. Set $(\mathsf{T}_0^{\mathsf{mem}}, h_0) := \mathsf{SparseHT.H}(\mathsf{SparseHT.hk}, 2^{\mathsf{L}_{\mathsf{st}}}, \emptyset)$.
4. Set $\mathsf{st}_0$ to be the initialize state of $M$.
5. For every $i \in [t]$ set the following values:
   • Set $(\mathsf{type}_i, \ell_i^{\mathsf{inp}}) := \mathsf{StepInputRead}(\mathsf{st}_{i-1})$.
   • If $\mathsf{type} = \mathsf{imp}$ then set $(z_i^{\mathsf{inp}}, \mathsf{auth}_i^{\mathsf{inp}}) := \mathsf{HT.R}(\mathsf{T}^{\mathsf{imp}}, \ell_i^{\mathsf{inp}})$.
   • If $\mathsf{type} = \mathsf{exp}$ then set $z_i^{\mathsf{inp}} := (x^{\mathsf{exp}})_{\ell_i^{\mathsf{inp}}}$, $\mathsf{auth}_i^{\mathsf{inp}} := \perp$.
   • Set $\ell_i^{\mathsf{mr}} := \mathsf{StepMemRead}(\mathsf{st}_{i-1}, z_i^{\mathsf{inp}})$.
   • Set $(z_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mr}}) := \mathsf{SparseHT.R}(\mathsf{T}_{i-1}^{\mathsf{mem}}, \ell_i^{\mathsf{mr}})$.
   • Set $(z_i^{\mathsf{mw}}, \ell_i^{\mathsf{mw}}, \mathsf{st}_i) := \mathsf{StepMemWrite}(\mathsf{st}_{i-1}, z_i^{\mathsf{inp}}, z_i^{\mathsf{mr}})$.
   • Set $(\mathsf{T}_i^{\mathsf{mem}}, h_i, \mathsf{auth}_i^{\mathsf{mw}}) := \mathsf{SparseHT.W}(\mathsf{T}_{i-1}^{\mathsf{mem}}, \ell_i^{\mathsf{mw}}, z_i^{\mathsf{mw}})$.
6. Set $\mathsf{rt}_{\mathsf{st}} := \mathsf{SEH.H}(\mathsf{SEH.hk}_1, (\mathsf{st}_1, \ldots, \mathsf{st}_t))$.
7. Set $\mathsf{rt}_h := \mathsf{SEH.H}(\mathsf{SEH.hk}_2, (h_1, \ldots, h_t))$.
8. Set $\rho_{h,0}, \rho_{\mathsf{st},0} = \perp$.
9. For every $i \in [t]$:
   • Set $(\cdot, \rho_{\mathsf{st},i}) := \mathsf{SEH.O}(\mathsf{SEH.hk}_1, (\mathsf{st}_1, \ldots, \mathsf{st}_t), I_i^{\mathsf{st}})$.
   • Set $(\cdot, \rho_{h,i}) := \mathsf{SEH.O}(\mathsf{SEH.hk}_2, (h_1, \ldots, h_t), I_i^h)$.
10. For every $i \in [t]$:
    set $\mathrm{w}_i := (\mathsf{st}_{i-1}, \mathsf{st}_i, h_{i-1}, h_i, z_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mw}}, z_i^{\mathsf{inp}}, \mathsf{auth}_i^{\mathsf{inp}}, \rho_{\mathsf{st},i-1}, \rho_{\mathsf{st},i}, \rho_{h,i-1}, \rho_{h,i})$.
```

11. If st_t is an accepting state then set $out = 1$, and if st_t is a rejecting state then set $out = 0$.
12. Set $\mathsf{BARG}.\pi := \mathsf{BARG.P}(\mathsf{BARG.crs}, C, (\mathbb{w}_1, \dots, \mathbb{w}_t))$.
13. Output $(out, \pi = (\mathsf{BARG}.\pi, \mathsf{rt}_{\mathsf{st}}, \mathsf{rt}_h))$.

$\underline{\mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, b, \pi)}$:

1. Parse $\mathsf{crs} := (\mathsf{BARG.crs}, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, t)$, $\pi = (\mathsf{BARG}.\pi, \mathsf{rt}_{\mathsf{st}}, \mathsf{rt}_h)$.
2. Set $(\cdot, h_0) := \mathsf{SparseHT.H}(\mathsf{SparseHT.hk}, 2^{\mathsf{L}_{\mathsf{st}}}, \emptyset)$.
3. Output $\mathsf{BARG.V}(\mathsf{BARG.crs}, C_{\mathsf{HT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, x^{\mathsf{exp}}, \mathsf{d}^{\mathsf{imp}}, \mathsf{rt}_h, \mathsf{rt}_{\mathsf{st}}, h_0, out}, \mathsf{BARG}.\pi)$.

6.2 Analysis

In what follows, we will provide proofs for completeness, efficiency, and soundness for the RAM Delegation construction in section Section 6.1.

Completeness. Follows directly from the completeness properties of the underlying HT, SEH and seBARG schemes.

Complexity. We bound the following functions with $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\lambda, \log t) \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}})$.

- To bound the running time of RAM.G we use the following facts:
  - The underlying SEH is succinct, and therefore the $\mathsf{SEH.G}(1^\lambda, t \cdot \mathsf{L}_{\mathsf{st}}, I_i^{\mathsf{st}})$ takes $\mathrm{poly}(\lambda, \log t) \cdot \mathsf{L}_{\mathsf{st}}$ time.
  - The underlying SEH is succinct, and therefore the $\mathsf{SEH.G}(1^\lambda, t \cdot \mathsf{L}_{\mathsf{rt}}, I_i^h)$ takes $\mathrm{poly}(\lambda, \log t)$ time.
  - The underlying seBARG is $\mathcal{L}$-succinct, and therefore the $(\mathsf{BARG.crs}, \mathsf{BARG.td}) \leftarrow \mathsf{BARG.G}(1^\lambda, t, \mathsf{L}_C, i)$ takes $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L}_C)$ time, where $\mathsf{L}_C = |C| \leq \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- To bound the proof size we use the following facts:
  - The underlying SEH is succinct, and therefore $|\mathsf{rt}_{\mathsf{st}}| \leq \mathrm{poly}(\lambda, \log t) \cdot \mathsf{L}_{\mathsf{st}}$.
  - The underlying SEH is succinct, and therefore $|\mathsf{rt}_h| \leq \mathrm{poly}(\lambda, \log t)$.
  - The underlying seBARG is $\mathcal{L}$-succinct, and therefore $\mathsf{BARG}.\pi \leq \mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L}_C)$, where $\mathsf{L}_C = |C| \leq \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- To bound the running time of RAM.V we use the following facts:
  - By bounding RAM.G we got that $|\mathsf{crs}| \leq \mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}})$.
  - We got that the proof size is at most $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}})$.
  - $\mathsf{SparseHT.H}(\mathsf{SparseHT.hk}, 2^{\mathsf{L}_{\mathsf{st}}}, \emptyset)$ takes $\mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}})$ time.
  - The underlying seBARG is $\mathcal{L}$-succinct, and therefore the BARG.V takes $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L}_C)$ time, where $\mathsf{L}_C = |C| \leq \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- To bound the running time of RAM.P we use the following facts:
  - By bounding RAM.G we got that $|\mathsf{crs}| \leq \mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}})$.
  - SparseHT.H operations on a tree with at most t elements takes $(t + 1) \cdot \mathrm{poly}(\lambda)$.
  - The algorithms $\mathsf{StepInputRead}, \mathsf{StepMemRead}, \mathsf{StepMemWrite}$ takes poly-time in L_{st}.

– The underlying seBARG prover runs in time polynomial in its input.

Soundness. Fix a read-write RAM machine M. Assume there exists:

- a secure hash tree,

$$\mathsf{HT} = (\mathsf{HT.G}, \mathsf{HT.H}, \mathsf{HT.R}, \mathsf{HT.W}, \mathsf{HT.VR}, \mathsf{HT.VW}) \ .$$

- a secure sparse hash tree,

$$\begin{aligned}&\mathsf{SparseHT} = \\ &(\mathsf{SparseHT.G}, \mathsf{SparseHT.H}, \mathsf{SparseHT.R}, \mathsf{SparseHT.W}, \mathsf{SparseHT.VR}, \mathsf{SparseHT.VW}) \ .\end{aligned}$$

- a secure somewhere extractable hash with succinct local opening,

$$\mathsf{SEH} = (\mathsf{SEH.G}, \mathsf{SEH.H}, \mathsf{SEH.O}, \mathsf{SEH.V}, \mathsf{SEH.E}) \ .$$

- an $\mathcal{L}$-succinct somewhere extractable batch argument for $\mathsf{BatchCSAT}$,

$$\mathsf{seBARG} = (\mathsf{BARG.G}, \mathsf{BARG.P}, \mathsf{BARG.V}, \mathsf{BARG.E}) \ .$$

Suppose toward contradiction that there exists a poly-size adversary A, polynomial $t = t(\lambda)$, and non-negligible function $\epsilon(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l}\forall b \in \{0,1\} \\ \mathsf{RAM.V}(\mathsf{crs}, \mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, b, \pi_b) = 1\end{array} \middle| \begin{array}{l}\mathsf{crs} \leftarrow \mathsf{RAM.G}(1^\lambda, t) \\ (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs})\end{array}\right] \geq \epsilon(\lambda) \ .$$

Parse $\mathsf{crs} = (\mathsf{BARG.crs}, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{hk}_1, \mathsf{hk}_2, t)$, and $\pi_b = (\mathsf{BARG}.\pi_b, \mathsf{rt}_{b,\mathsf{st}}, \mathsf{rt}_{b,h})$. By the definition of $\mathsf{RAM.V}$, we get that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l}\forall b \in \{0,1\} \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1\end{array} \middle| \begin{array}{l}\mathsf{crs} \leftarrow \mathsf{RAM.G}(1^\lambda, t) \\ (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs})\end{array}\right] \geq \epsilon(\lambda) \ .$$

where $C_b = C_{\mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{hk}_1, \mathsf{hk}_2, x^{\mathsf{exp}}, \mathsf{d}^{\mathsf{imp}}, \mathsf{rt}_{b,h}, \mathsf{rt}_{b,\mathsf{st}}, h_0, b}$.

For every $j \in [t]$, let $\mathsf{RAM.G}_j$ be identical to $\mathsf{RAM.G}$, except that rather than setting $i = 1$ it sets $i = j$. By the index hiding property of SEH and the index hiding property of seBARG, the above equation implies that there exists a negligible function $\mu_1(\cdot)$ such that for every $i \in [t]$ and every $\lambda \in \mathbb{N}$,

$$\begin{aligned}&\Pr\left[\begin{array}{l}\forall b \in \{0,1\} \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1\end{array} \middle| \begin{array}{l}\mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs})\end{array}\right] \\ &\geq \epsilon(\lambda) - \mu_1(\lambda) \ .\end{aligned}$$

In the equations below, to avoid lengthy equations, we omit the prefix of $\mathsf{BARG.td}, \mathsf{SEH.td}_1, \mathsf{SEH.td}_2$ and use it as $\mathsf{td}, \mathsf{td}_1, \mathsf{td}_2$ accordingly.

By the somewhere argument of knowledge property of the underlying seBARG scheme, the above equation implies that there exists a negligible function $\mu_2(\cdot)$ such that for every $i \in [t]$ and every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ C_b(i, \mathbb{w}_b) = 1 & \forall b \in \{0,1\} \\ & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \epsilon(\lambda) - \mu_2(\lambda) \ . \tag{12}$$

For every $b \in \{0,1\}$ parse

$$\mathbb{w}_b = (\mathsf{st}_{b,i-1}, \mathsf{st}_{b,i}, h_{b,i-1}, h_{b,i}, z_b^{\mathsf{mr}}, \mathsf{auth}_b^{\mathsf{mr}}, \mathsf{auth}_b^{\mathsf{mw}}, z_b^{\mathsf{inp}}, \mathsf{auth}_b^{\mathsf{inp}}, \rho_{b,\mathsf{st}_{i-1}}, \rho_{b,\mathsf{st}_i}, \rho_{b,h_{i-1}}, \rho_{b,h_i}).$$

We next argue that Equation 12 implies that there exists a negligible function $\xi(\cdot)$ such that for every $i \in [t]$ and every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ C_b(i, \mathbb{w}_b) = 1 & \forall b \in \{0,1\} \\ \mathsf{st}_{0,i} = \mathsf{st}_{1,i}, h_{0,i} = h_{1,i} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right] \geq \epsilon(\lambda) - i \cdot \xi(\lambda) \ . \tag{13}$$

Fix $i = t$. By the definition of C_b, the above equation implies that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_t(1^\lambda, t) \\ C_b(t, \mathbb{w}_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_{0,t} = \text{reject}, \ \mathsf{st}_{1,t} = \text{accept} & \forall b \in \{0,1\} \\ \mathsf{st}_{0,t} = \mathsf{st}_{1,t} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right] \geq \epsilon(\lambda) - i \cdot \xi(\lambda) \ .$$

Which is a contradiction, since $\mathsf{st}_{0,t} = \text{reject}, \ \mathsf{st}_{1,t} = \text{accept}$ implies that $\mathsf{st}_{0,t} \neq \mathsf{st}_{1,t}$.

In order to prove Equation 13 , we start by stating a useful claim, which we will prove later on.

Claim 6.4. There exists a negligible function $\eta(\cdot)$ such that for every $i \in [t]$ and every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ C_b(i, \mathbb{w}_b) = 1 & \forall b \in \{0,1\} \\ \mathsf{st}_{0,i-1} = \mathsf{st}_{1,i-1}, h_{0,i-1} = h_{1,i-1} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$- \Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_{0,i-1} = \mathsf{st}_{1,i-1}, h_{0,i-1} = h_{1,i-1} & \forall b \in \{0,1\} \\ \mathsf{st}_{0,i} = \mathsf{st}_{1,i}, h_{0,i} = h_{1,i} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right] \leq \eta(\lambda) \ .$$

Base case. $i = 1$. Follows directly from Claim 6.4 together with the definition of C_b that includes a unique initial state st_0 and a unique initial memory root h_0.

Inductive step. Supposing Equation 13 holds for $i-1$, we proceed to prove that it holds for i. By the inductive assumption together with the somewhere extractability property of the underlying SEH scheme, it holds that there exists a negligible function $\nu_1(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG.}\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_{i-1}(1^\lambda, t) \\ C_b(i-1, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_{0,i-1} = \mathsf{st}_{1,i-1}, h_{0,i-1} = h_{1,i-1} & \forall b \in \{0,1\} \\ \forall b \in \{0,1\} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG.}\pi_b) \\ \mathsf{st}_{b,i-1} = \mathsf{SEH.E}(\mathsf{td}_1, \mathsf{rt}_{b,\mathsf{st}}) & \\ h_{b,i-1} = \mathsf{SEH.E}(\mathsf{td}_2, \mathsf{rt}_{b,h}) & \end{array}\right]$$
$$\geq \epsilon(\lambda) - (i-1)\cdot\xi(\lambda) - \nu_1(\lambda) \ .$$

Let $\mathsf{RAM.G}'_i$ be the algorithm that on input $(1^\lambda, t)$ generates $\mathsf{BARG.crs}$ w.r.t. index i, but generates $\mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2$ w.r.t. $I^{\mathsf{st}}_{i-1}, I^h_{i-1}$ (as opposed to I^{st}_i, I^h_i). More formally, $\mathsf{RAM.G}'_i$ generates $(\mathsf{BARG.crs}, \mathsf{BARG.td}) \leftarrow \mathsf{BARG.G}(1^\lambda, t, \mathsf{L}_C, i)$, $(\mathsf{SEH.hk}_1, \mathsf{SEH.td}_1) \leftarrow \mathsf{SEH.G}(1^\lambda, t\cdot\mathsf{L}_{\mathsf{st}}, I^{\mathsf{st}}_{i-1})$ for $I^{\mathsf{st}}_i = \{(i-2)\cdot\mathsf{L}_{\mathsf{st}}+1, \dots, (i-1)\cdot\mathsf{L}_{\mathsf{st}}\}$ and $(\mathsf{SEH.hk}_2, \mathsf{SEH.td}_2) \leftarrow \mathsf{SEH.G}(1^\lambda, t\cdot\mathsf{L}_{\mathsf{rt}}, I^h_{i-1})$ for $I^h_i = \{(i-2)\cdot\mathsf{L}_{\mathsf{rt}}+1, \dots, (i-1)\cdot\mathsf{L}_{\mathsf{rt}}\}$.

The equation above, together with the index hiding of the underlying seBARG scheme, implies that there exists a negligible function $\nu_2(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG.}\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}'_i(1^\lambda, t) \\ \mathsf{SEH.E}(\mathsf{td}_1, \mathsf{rt}_{0,\mathsf{st}}) = \mathsf{SEH.E}(\mathsf{td}_1, \mathsf{rt}_{1,\mathsf{st}}) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{SEH.E}(\mathsf{td}_2, \mathsf{rt}_{0,h}) = \mathsf{SEH.E}(\mathsf{td}_2, \mathsf{rt}_{1,h}) & \end{array}\right]$$
$$\geq \epsilon(\lambda) - (i-1)\cdot\xi(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) \ .$$

By the somewhere argument of knowledge property of the underlying seBARG scheme, the above equation implies that there exists a negligible function $\nu_3(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \mathsf{crs} \leftarrow \mathsf{RAM.G}'_i(1^\lambda, t) \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG.}\pi_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ C_b(i, \mathbb{w}_b) = 1 & \forall b \in \{0,1\} \\ \mathsf{SEH.E}(\mathsf{td}_1, \mathsf{rt}_{0,\mathsf{st}}) = \mathsf{SEH.E}(\mathsf{td}_1, \mathsf{rt}_{1,\mathsf{st}}) & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG.}\pi_b) \\ \mathsf{SEH.E}(\mathsf{td}_2, \mathsf{rt}_{0,h}) = \mathsf{SEH.E}(\mathsf{td}_2, \mathsf{rt}_{1,h}) & \end{array}\right]$$
$$\geq \epsilon(\lambda) - (i-1)\cdot\xi(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) - \nu_3(\lambda) \ .$$

By the somewhere extractability property of the underlying SEH scheme, together with the definition of C_b, there exists a negligible function $\nu_4(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \mathsf{crs} \leftarrow \mathsf{RAM.G}'_i(1^\lambda, t) \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG.}\pi_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ C_b(i, \mathbb{w}_b) = 1 & \forall b \in \{0,1\} \\ \mathsf{st}_{0,i-1} = \mathsf{st}_{1,i-1}, h_{0,i-1} = h_{1,i-1} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG.}\pi_b) \end{array}\right]$$
$$\geq \epsilon(\lambda) - (i-1)\cdot\xi(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) - \nu_3(\lambda) - \nu_4(\lambda) \ .$$

By the index hiding property of the underlying SEH scheme, there exists a negligible function $\nu_5(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ C_b(i, \mathbb{w}_b) = 1 & \forall b \in \{0,1\} \\ \mathsf{st}_{0,i-1} = \mathsf{st}_{1,i-1}, h_{0,i-1} = h_{1,i-1} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \epsilon(\lambda) - (i-1)\cdot\xi(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) - \nu_3(\lambda) - \nu_4(\lambda) - \nu_5(\lambda) \ .$$

By the above, and by Claim 6.4 we get that:

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_{0,i-1} = \mathsf{st}_{1,i-1}, h_{0,i-1} = h_{1,i-1} & \forall b \in \{0,1\} \\ \mathsf{st}_{0,i} = \mathsf{st}_{1,i}, h_{0,i} = h_{1,i} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \epsilon(\lambda) - (i-1)\cdot\xi(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) - \nu_3(\lambda) - \nu_4(\lambda) - \nu_5(\lambda) - \eta(\lambda) \ .$$

By setting $\xi(\lambda) = \eta(\lambda) + \sum_{j=1}^{5}\nu_j(\lambda)$, the above proves the inductive step, as desired. We now left to prove Claim 6.4.

Proof of Claim 6.4. Fix some $i \in [t]$. Let $\varepsilon(\cdot)$ be a function such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ C_b(i, \mathbb{w}_b) & \forall b \in \{0,1\} \\ \mathsf{st}_0 = \mathsf{st}_1, h_0 = h_1 & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right] = \varepsilon(\lambda) \ . \tag{14}$$

where for every $b \in \{0,1\}$ parse

$$\mathbb{w}_b = (\mathsf{st}_b, \mathsf{st}'_b, h_b, h'_b, z_b^{\mathsf{mr}}, \mathsf{auth}_b^{\mathsf{mr}}, \mathsf{auth}_b^{\mathsf{mw}}, z_b^{\mathsf{inp}}, \mathsf{auth}_b^{\mathsf{inp}}, \rho_{b,\mathsf{st}}, \rho_{b,\mathsf{st}'}, \rho_{b,h_{i-1}}, \rho_{b,h'_i}) \ .$$

For the experiment above, for every $b \in \{0,1\}$ let:

$$\ell_b^{\mathsf{inp}} \leftarrow \mathsf{StepInputRead}(\mathsf{st}_b)$$
$$\ell_b^{\mathsf{mr}} \leftarrow \mathsf{StepMemRead}(\mathsf{st}_b, z_b^{\mathsf{inp}})$$
$$(z_b^{\mathsf{mw}}, \ell_b^{\mathsf{mw}}, \mathsf{st}''_b) \leftarrow \mathsf{StepMemWrite}(\mathsf{st}_b, z_b^{\mathsf{inp}}, z_b^{\mathsf{mr}})$$

By the definition of ℓ_b^{inp}, Equation 14 implies that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_0 = \mathsf{st}_1, h_0 = h_1, & \forall b \in \{0,1\} \\ \ell_0^{\mathsf{inp}} = \ell_1^{\mathsf{inp}} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right] = \varepsilon(\lambda) \ .$$

By the soundness of read property of the underlying HT scheme, together with the definition of C_b, the above equation implies that there exists a negligible function $\nu_1(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_0 = \mathsf{st}_1, h_0 = h_1, & \forall b \in \{0,1\} \\ z_0^{\mathsf{inp}} = z_1^{\mathsf{inp}} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \varepsilon(\lambda) - \nu_1(\lambda) \ .$$

By the definition of ℓ_b, the above equation implies that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_0 = \mathsf{st}_1, h_0 = h_1, & \forall b \in \{0,1\} \\ z_0^{\mathsf{inp}} = z_1^{\mathsf{inp}}, \ell_0^{\mathsf{mr}} = \ell_1^{\mathsf{mr}} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \varepsilon(\lambda) - \nu_1(\lambda) \ .$$

By the soundness of read property of the underlying $\mathsf{SparseHT}$ scheme, together with the definition of C_b, and since $(h_0 = h_1)$, the above equation implies that there exists a negligible function $\nu_2(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_0 = \mathsf{st}_1, h_0 = h_1, & \forall b \in \{0,1\} \\ z_0^{\mathsf{inp}} = z_1^{\mathsf{inp}}, z_0^{\mathsf{mr}} = z_1^{\mathsf{mr}} & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \varepsilon(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) \ .$$

By the definition of $(z_b^{\mathsf{mw}}, \ell_b^{\mathsf{mw}}, \mathsf{st}_b'')$, we get that the above equation implies that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_0 = \mathsf{st}_1, h_0 = h_1, & \forall b \in \{0,1\} \\ (z_0^{\mathsf{mw}}, \ell_0^{\mathsf{mw}}, \mathsf{st}_0'') = (z_1^{\mathsf{mw}}, \ell_1^{\mathsf{mw}}, \mathsf{st}_1'') & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \varepsilon(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) \ .$$

By the soundness of write property of the underlying $\mathsf{SparseHT}$ scheme, together with the definition of C_b, and since in the above equation $(z_0^{\mathsf{mw}}, \ell_0^{\mathsf{mw}}, \mathsf{st}_0'') = (z_1^{\mathsf{mw}}, \ell_1^{\mathsf{mw}}, \mathsf{st}_1'')$, the above equation implies that there exists a negligible function $\nu_3(\cdot)$ such that for every $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l|l} \forall b \in \{0,1\} & \\ \mathsf{BARG.V}(\mathsf{BARG.crs}, C_b, \mathsf{BARG}.\pi_b) = 1 & \mathsf{crs} \leftarrow \mathsf{RAM.G}_i(1^\lambda, t) \\ C_b(i, \mathbb{w}_b) & (\mathsf{d}^{\mathsf{imp}}, x^{\mathsf{exp}}, \pi_0, \pi_1) \leftarrow A(\mathsf{crs}) \\ \mathsf{st}_0 = \mathsf{st}_1, h_0 = h_1, & \forall b \in \{0,1\} \\ \mathsf{st}'_0 = \mathsf{st}'_1, h'_0 = h'_1 & \mathbb{w}_b = \mathsf{BARG.E}(\mathsf{td}, C_b, \mathsf{BARG}.\pi_b) \end{array}\right]$$
$$\geq \varepsilon(\lambda) - \nu_1(\lambda) - \nu_2(\lambda) - \nu_3(\lambda) \ .$$

By setting $\eta(\lambda) = \sum_{j=1}^{3} \nu_j(\lambda)$, the above proves the claim, as desired. □

7 Complexity-Preserving RAM Delegation Scheme

In this section, we present an alternative implementation of the RAM Delegation prover introduced in Section 6.1. We begin by introducing time- and space-efficient algorithms for Merkle trees in Section 7.1. These algorithms are designed to operate on a stream as input, ensuring space efficiency. In Section 7.2 we define and construct a data structure that serves as an efficient memory manager. In Section 7.3, we use the techniques and algorithms presented in Section 7.1 and the efficient data structure provided in Section 7.2 to develop a complexity-preserving prover for our RAM Delegation scheme. By replacing the original RAM Delegation prover with our efficient implementation from Section 6.1, we establish the following theorem.

Theorem 7.1. *Assume there exists a secure hash tree* HT*, a secure sparse hash tree* SparseHT*, a secure two-mode somewhere extractable hash* SEH *with succinct local opening, and an $\mathcal{L}$-succinct somewhere extractable batch argument* seBARG *for* BatchCSAT*, then there exists an $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\lambda, \log t)$-succinct* RAM Delegation *scheme* RAM *with strong soundness for any read and write RAM machine M. Moreover, the* RAM *prover has the following properties:*
- *The prover runs in time $\big(t \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}}, \log t) + |x^{\mathsf{imp}}|\big) \cdot \mathrm{poly}(\lambda)$.*
- *The prover uses sapce $\big(w \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}}, \log t) + |x^{\mathsf{imp}}|\big) \cdot \mathrm{poly}(\lambda)$.*

where $(x^{\mathsf{imp}}, x^{\mathsf{exp}})$ is the input to M, t is the running time of $M(x^{\mathsf{imp}}, x^{\mathsf{exp}})$, and w is the space used by $M(x^{\mathsf{imp}}, x^{\mathsf{exp}})$.

By combining Theorem 7.1 with Theorems 5.2, 5.9, 5.7, and 3.8 we get the following corollary.

Corollary 7.2. *There exist* $\mathrm{poly}(\lambda, \log t)$*-succinct* RAM Delegation *scheme for any read-write* RAM *machine under any of the following assumptions:*

1. *The $O(1)$-*LIN *assumption on a pair of cryptographic groups with efficient bilinear map.*
2. *The hardness of Learning with errors* (LWE) *problem against polynomial time adversaries.*
3. *The sub-exponential Decisional Diffie-Hellman (*DDH*) assumptions.*

Moreover, the RAM *prover has the following properties:*
- *The prover runs in time $\big(t \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}}, \log t) + |x^{\mathsf{imp}}|\big) \cdot \mathrm{poly}(\lambda)$.*
- *The prover uses sapce $\big(w \cdot \mathrm{poly}(\mathsf{L}_{\mathsf{st}}, \log t) + |x^{\mathsf{imp}}|\big) \cdot \mathrm{poly}(\lambda)$.*

where $(x^{\mathsf{imp}}, x^{\mathsf{exp}})$ is the input to M, t is the running time of $M(x^{\mathsf{imp}}, x^{\mathsf{exp}})$, and w is the space used by $M(x^{\mathsf{imp}}, x^{\mathsf{exp}})$.

7.1 Efficient Merkle tree for streaming data

In this subsection, we will introduce a set of procedures for Merkle trees that are designed to be efficient both in terms of time and space. To accommodate the space constraints, these procedures operate on a stream of elements rather than requiring the entire input to be stored in memory. To ensure efficiency in terms of time, we will employ a specific stream scheme that enables us to back up the state of the stream and then return to that state at any given time. For a more detailed explanation of this stream scheme, please refer to Definition 7.3.

A stateful (deterministic) stream scheme

$$\mathsf{Stm} = (\mathsf{Init}, \mathsf{Next}, \mathsf{Backup}, \mathsf{DeleteBackup}, \mathsf{Jump}, \mathsf{GetData})$$

has the following syntax:

$\mathsf{Init}(x, \mathsf{k})$**.** The algorithm takes as input a setup data x, and the maximum number of saves that the scheme can store. The algorithm initialize the parameters of the stream accordingly.

$\mathsf{Next}()$**.** The algorithm advances the stream's state to point to the subsequent element in the stream.

$\mathsf{Backup}() \to \mathsf{i}$**.** The algorithm creates a backup for the current state of the stream. It outputs the index of the current element in the stream.

$\mathsf{DeleteBackup}(\mathsf{i})$**.** The algorithm deletes the backup of for the i-th element in the stream.

$\mathsf{Jump}(\mathsf{i})$**.** This algorithm accepts an input index i. If there are at most k backups in the stream, and a backup was created when the stream was at index i (and has not been subsequently deleted), then the algorithm restores the stream to index i.

$\mathsf{GetData}() \to \mathsf{data}$**.** The algorithm outputs the data data corresponding to the element currently referenced by the stream.

Definition 7.3 (Stm)**.** *A stateful stream has the following requirements,*

Efficiency. *We bound the efficiency parameters of the stream as follows:*

- *Let* T_{Stm} *be the bound on the running time of one stream operation* ($\mathsf{Next}, \mathsf{Backup}, \mathsf{DeleteBackup}, \mathsf{Jump}, \mathsf{GetData}$)*.*
- *Let* $\overrightarrow{T}_{\mathsf{Stm}} : \mathbb{N} \to \mathbb{N}$ *be a function that gets as input a bound on the number of times that one stream element is accessed, and outputs the total running time of the stream.*
- *Let* S_{Stm} *be a bound on the space used in the stream.*
- *Let* S_{data} *be the output size of* $\mathsf{GetData}$*.*

The five Merkle tree (deterministic) procedure are as follows:

$\underline{\mathsf{ComputeTreeHash}}(\mathsf{H}, \mathsf{stm}, n) \to \mathsf{rt}$**.** This algorithm takes as input a hash function H, a reference to a stream stm, and a number n. The algorithm outputs the Merkle root rt of n elements in the stream stm.

$\underline{\mathsf{ComputeAuthPath}}(\mathsf{H}, \mathsf{stm}, i, n) \to \mathsf{auth}$**.** This algorithm takes as input a hash function H, a reference to a stream stm, an index i, and a number n. The algorithm computes the Merkle tree of n elements in the stream stm, and outputs the authentication path of the i-th element.

$\underline{\mathsf{ComputeTreeRootUsingAuthPath}}(\mathsf{H}, i, \mathsf{Leaf}, \mathsf{auth}, n) \to \mathsf{rt}$**.** This algorithm takes as input a hash function H, an index i, a leaf in the tree Leaf, an authentication path auth, and the number of leafs in the tree n. It outputs the root of the tree rt.

$\underline{\mathsf{ComputeNextAuthPath}}(\mathsf{H}, \mathsf{stm}, i, \mathsf{auth}_i, n) \to \mathsf{auth}$**.** This algorithm takes as input a hash function H, a reference to a stream stm, an index i, an authentication path auth_i to the i-th leaf in the tree, and a number n. The algorithm outputs an authentication path auth of the $i+1$ leaf in the tree, where the leafs are the elements in the stream stm.

$\underline{\mathsf{StreamAllAuthPaths}}(\mathsf{H}, \mathsf{stm}, n) \to (\mathsf{auth}_1, \ldots, \mathsf{auth}_n)$. This algorithm takes as input a hash function H, a reference to a stream stm, and a number n. The algorithm outputs a stream of authentication paths $\mathsf{auth}_1, \ldots, \mathsf{auth}_n$ of all the leafs in a Merkle tree, where the leafs are the elements in the stream stm.

Each of the described procedures leaves the stream's state unaltered. To ensure this, we implicitly follow a specific sequence: We initiate the procedure by saving the current stream state using $\mathsf{i} := \mathsf{stm.Backup}()$. We then proceed to execute the relevant procedure. Once the procedure is completed, we restore the stream's state by employing $\mathsf{stm.Jump}(\mathsf{i})$. Finally, we remove the saved state using $\mathsf{stm.DeleteBackup}(\mathsf{i})$.

Remark 7.4. We note that, since each backup is invoked at the beginning of the procedure, and discarded by the end, the number of backups stored at any point is determined by the size of the function-calling stack. As we will see, each function has a function-calling stack of size at most 3, which results in at most 3 backups stored at any time.

We start by describing the procedure $\mathsf{ComputeTreeHash}$. While this algorithm is already well-established, with a mention in [Szy04], we provide a detailed description to adapt it for stream input access instead of random access to the input.

$\underline{\mathsf{ComputeTreeHash}}(\mathsf{H}, \mathsf{stm}, n)$**:**

```
1. If the top two nodes on the stack have the same height in the tree,
   1.1. Pop $v^{\text{right}}$.
   1.2. Pop $v^{\text{left}}$.
   1.3. Execute $v^{\text{parent}} := \mathsf{H}(v^{\text{left}} \,||\, v^{\text{right}})$.
   1.4. If the height of $v^{\text{parent}}$ is $\log n$, output $v^{\text{parent}}$, destroy the stack, and stop.
   1.5. Push $v^{\text{parent}}$ onto the stack.
2. Else,
   2.1. Push $\mathsf{stm.GetData}()$ onto the stack.
   2.2. Execute $\mathsf{stm.Next}()$.
3. Loop to step 1.
```

Complexity: The above algorithm does $T = O(n)$ steps, go over each element in the stream at most $N_{\mathsf{elem}} = 1$ times, and stores a maximum of $S = \log n + 1$ hashed values at once.

The next algorithm computes an authentication path of some leaf in the tree. The algorithm is implemented the same as $\mathsf{ComputeTreeHash}$, except for collecting the relevant nodes for the authentication path while going over the tree. Note that for each height $\mathsf{d}' < \mathsf{d}$, we define $\mathsf{auth}_{\mathsf{d}'}$ to be the value of the sibling of the height d' node on the path from the leaf to the root. The authentication path is then the set $\mathsf{auth} = \{\mathsf{auth}_j \mid 0 \leq j < \mathsf{d}\}$.

$\underline{\mathsf{ComputeAuthPath}}(\mathsf{H}, \mathsf{stm}, i, n)$**:**

```
1. If the top two nodes on the stack have the same height in the tree,
   1.1. Set $i$ to be the height of the top two nodes.
   1.2. Pop $v^{\text{right}}$.
   1.3. Pop $v^{\text{left}}$.
   1.4. If $\lceil \frac{i}{2^j} \rceil$ is odd then $\mathsf{auth}_j := v^{\text{right}}$.
   1.5. If $\lceil \frac{i}{2^j} \rceil$ is even then $\mathsf{auth}_j := v^{\text{left}}$.
   1.6. If $\mathsf{d}' = \log n - 1$, output $\{\mathsf{auth}_0, \ldots, \mathsf{auth}_{\log n - 1}\}$, destroy the stack, and stop.
   1.7. Compute $v^{\text{parent}} := \mathsf{H}(v^{\text{left}} \mid\mid v^{\text{right}})$.
   1.8. Push $v^{\text{parent}}$ onto the stack.
2. Else,
   2.1. Push $\mathsf{stm.GetData}()$ onto the stack.
   2.2. Execute $\mathsf{stm.Next}()$.
3. Loop to step 1.
```

Complexity: Similarly to the $\mathsf{ComputeTreeHash}$, the above algorithm does $T = O(n)$ steps, and go over each element in the stream at most $N_{\mathsf{elem}} = 1$ times. The algorithm stores a maximum of $S = \log n + 1$ hash values on the stack and additional $\log n$ elements for the authentication path.

The next algorithm computes the root of a tree using a leaf in the tree, the authentication path of the root, and the number of leafs in the tree.

$\underline{\mathsf{ComputeTreeRootUsingAuthPath}}(\mathsf{H}, i, \mathsf{Leaf}, \mathsf{auth}, n)$**:**

```
1. Set $v := \mathsf{Leaf}$.
2. For every $j \in \{0, \ldots, \log n - 1\}$ do the following:
   2.1. If $\lceil \frac{i}{2^j} \rceil$ is odd then $v := \mathsf{H}(v \mid\mid \mathsf{auth}_j)$.
   2.2. Else if $\lceil \frac{i}{2^j} \rceil$ is even then $v := \mathsf{H}(\mathsf{auth}_j \mid\mid v)$.
3. Output $v$ as the root.
```

Complexity: The algorithm does $T = O(\log n)$ steps, and stores constant number of hashed values at every step.

$\underline{\mathsf{ComputeNextAuthPath}}(\mathsf{H}, \mathsf{stm}, i, \mathsf{auth}, n)$**:**

```
1. Parse $\mathsf{auth} := \{\mathsf{auth}_0, \ldots, \mathsf{auth}_{\log n - 1}\}$.
2. Set $\mathsf{d}'$ to be the height of the first common ancestor of the $i$ and $(i+1)$ leafs in the tree.
3. For every $\mathsf{d}' \leq j < \log n$, set $\mathsf{auth}^{\mathsf{next}}_j := \mathsf{auth}_j$.
4. Set $\mathsf{auth}^{\mathsf{next}}_{\mathsf{d}'-1} :=$
   $\mathsf{ComputeTreeRootUsingAuthPath}\left(\mathsf{H}, \mathsf{stm.GetData}(), \{\mathsf{auth}_0, \ldots, \mathsf{auth}_{\mathsf{d}'-2}\}, 2^{\mathsf{d}'-1}\right)$.
```

5. Set $\{\mathsf{auth}_0, \ldots, \mathsf{auth}_{\mathsf{d}'-2}\} := \mathsf{ComputeAuthPath}(\mathsf{H}, \mathsf{stm}, 1, 2^{\mathsf{d}'-1})$.

Complexity: Let d' be the height of the first common ancestor of the input element and the next element. The algorithm does $T = O(\log n + 2^{\mathsf{d}'})$, go over each element in the stream at most $N_{\mathsf{elem}} = O(1)$ times, and stores at most $S = O(\log n)$ hashed values in the memory at each step:

- Step 3 has $O(\log n - \mathsf{d}')$ inner steps, and stores $O(\log n - \mathsf{d}')$ elements in the memory.
- By the complexity of ComputeTreeRootUsingAuthPath, step 4 has $O(\mathsf{d}')$ inner steps, and stores $O(\mathsf{d}')$ hashed values in the memory.
- By the complexity of ComputeAuthPath, step 5 has $O(2^{\mathsf{d}'})$ inner steps, and stores $O(\mathsf{d}')$ hashed values in the memory.

Note that in the worst case, the algorithm does $O(n)$ steps, but as we will see in the next algorithm, on the average case, the algorithm does $O(\log n)$ steps. In addition, the algorithm go over each element in the stream at most $N_{\mathsf{elem}} = O(1)$ times, but specifically, it goes only over elements in the smallest common sub-tree of the i-th and $(i+1)$-th elements.

```
StreamAllAuthPaths(H, stm, n):
1. Set auth := ComputeAuthPath(H, stm, start, 1, $n$).
2. Output auth.
3. For $i \in [n-1]$ do as follows:
    3.1. Set $\mathsf{auth}^{\mathsf{next}}$ := ComputeNextAuthPath(H, stm, elem, $i$, auth, $n$).
    3.2. Set auth := $\mathsf{auth}^{\mathsf{next}}$.
    3.3. Output auth, and continue to the next iteration.
```

Complexity: The algorithm does $T = O(n \cdot \log n)$ steps, go over each element in the stream at most $N_{\mathsf{elem}} = O(\log n)$ times, and stores at most $S = O(\log n)$ hashed values in the memory at each step.

The value of N_{elem} is directly implied by the complexity of ComputeNextAuthPath, ComputeAuthPath. In what follows, we focus on proving the value of T, N_{elem}. To prove $T = O(n \cdot \log n)$, we define a new function $\mathsf{c} : [n] \times [n] \to [\log n]$. This function is given two indexes of leafs in the tree $i, j \in [n]$, and outputs the height of the first common ancestor of the i-th and j-th leafs in the tree. Let $T : \mathbb{N} \to \mathbb{N}$ be a function that gets as an input the number of leafs in the tree, and outputs the number of steps the algorithm does. Note that by the complexity of ComputeNextAuthPath and ComputeAuthPath, for any $n \in \mathbb{N}$:

$$\begin{aligned} f(n) &\leq O(n) + \sum_{i=1}^{n-1} O\left(\log n + 2^{\mathsf{c}(i,i+1)}\right) \\ &\leq O(n \cdot \log n) + \sum_{i=1}^{n-1} O\left(2^{\mathsf{c}(i,i+1)}\right) \quad . \end{aligned}$$

Note that the output values of c for the leafs in the left sub-tree is the same as the values of c for the leafs in the right sub-tree:

$$\left\{\mathsf{c}(1,2), \ldots, \mathsf{c}\left(\frac{n}{2}-1, \frac{n}{2}\right)\right\} = \left\{\mathsf{c}\left(\frac{n}{2}+1, \frac{n}{2}+2\right), \ldots, \mathsf{c}\left(n-1, n\right)\right\} \quad .$$

Therefore, the function f can be computed as:

$$f(n) \leq O(n \cdot \log n) + O\left(2^{\mathsf{c}\left(\frac{n}{2},\frac{n}{2}+1\right)}\right) + 2 \cdot \sum_{i=1}^{\frac{n}{2}} O\left(2^{\mathsf{c}(i,i+1)}\right) \ .$$

We have that the first ancestor of the of $\frac{n}{2}, \frac{n}{2}+1$ is the root. Therefore, $\mathsf{c}\left(\frac{n}{2}, \frac{n}{2}+1\right) = \log n$, and we get that:

$$f(n) \leq O(n \cdot \log n) + O(n) + 2 \cdot \sum_{i=1}^{\frac{n}{2}} O\left(2^{\mathsf{c}(i,i+1)}\right) \ .$$

It can be shown by induction that:

$$f(n) \leq O(n \cdot \log n) + \sum_{i=0}^{\log n - 1} 2^i \cdot O(2^{\log n - i}) \ .$$

Which implies that:

$$\begin{aligned} f(n) &\leq O(n \cdot \log n) + \sum_{i=0}^{\log n-1} O(n) \\ &= O(n \cdot \log n) \ . \end{aligned}$$

As required.

We left to prove the value of N_{elem}. The value of N_{elem} is dominated by the complexity of $\mathsf{ComputeNextAuthPath}$. As mentioned in the complexity of $\mathsf{ComputeNextAuthPath}$, the elements in the stream accessed by the function $\mathsf{ComputeNextAuthPath}(\cdot, i, \cdot)$ are only the elements that are under the smallest common sub-tree of $(i, i+1)$. Note that when going over the pairs of leafs in the tree $(1,2), (2,3), \ldots$, each node in the tree gets to be the smallest common ancestor only once. Fix some element in the stream. This element exists in at most $\log n$ sub-trees (one for each level in the tree). Since each node in the tree is a smallest common ancestor only once, we get that overall, the element is accessed $\log n$ times, as required.

7.2 Efficient memory scheme

We define a stateful data structure that represents the memory of a deterministic program, allowing for reading, writing, backing up, and restoration of the memory. A stateful (deterministic) *memory scheme*,

$$\mathsf{Mem} = (\mathsf{Init}, \mathsf{Read}, \mathsf{Write}, \mathsf{Backup}, \mathsf{DeleteBackup}, \mathsf{Restore}) \ ,$$

is a data structure with the following syntax:

$\mathsf{Init}(S, M, \mathsf{k})$**.** This algorithm takes as input the memory size S, the initial state of the memory M, and the maximum number of backups k that the scheme can store. The algorithm initializes the parameters of the scheme.

$\mathsf{Read}(\ell) \to b$**.** This algorithm takes as input a memory location ℓ. The algorithm outputs the value of the memory in location ℓ. If no write operations were performed on location ℓ, $\mathsf{Read}(\ell)$ outputs $M[\ell]$, where M is the initial memory.

$\mathsf{Write}(\ell, b)$**.** This algorithm takes as input a memory location ℓ, and a bit b. The algorithm updates the memory at a location ℓ with the value b.

$\mathsf{Backup}() \to \mathsf{ts}$**.** This algorithm creates a backup for the current state of the memory, and returns the time stamp in which the backup created.

$\mathsf{DeleteBackup}(\mathsf{ts})$**.** This algorithm takes as input a time stamp ts. The algorithm deletes the backup created in time ts.

$\mathsf{Restore}(\mathsf{ts})$**.** This algorithm takes as input a time stamp ts. If there are at most k backups and a backup was preformed in time ts (and was not subsequently deleted), then the algorithm restores the memory to its state at time ts. This operation does not delete the backups preformed after time ts.

Definition 7.5 (Mem)**.** *A stateful memory scheme satisfies the following requirements,*

Completeness. *The restore functionality holds only for memory changes that can be described by a deterministic* RAM *machine. In simpler terms, we expect the same write operation to occur even after a restore has been performed at time* ts.

Efficiency. *We bound the efficiency parameters of the stream as follows:*

- *The initialization algorithm runs in time* $T^{\mathsf{init}}_{\mathsf{Mem}}(\mathsf{k}, S)$.
- *The read, write, backup, and restore algorithms take time* $T_{\mathsf{Mem}}(\mathsf{k}, S)$.
- *The total memory required for the scheme is* $S_{\mathsf{Mem}}(\mathsf{k}, S)$.

Claim 7.6. There exists a Mem scheme with efficiency parameters

$$T^{\mathsf{init}}_{\mathsf{Mem}} = O(\mathsf{k} \cdot S),\ T_{\mathsf{Mem}} = \mathsf{k} \cdot \mathrm{polylog}(\mathsf{w}, S),\ S_{\mathsf{Mem}} = \mathsf{k} \cdot S \cdot \mathrm{polylog}(\mathsf{w})\ ,$$

Here, S is the size of the memory, k is the maximum number of stored backups, and w is the number of write operations performed.

Proof. The primary challenge in our proof arises from the efficiency requirements we aim to meet. These requirements dictate that the backup operation cannot directly create a copy of the memory, as doing so would take time proportional to the size of the memory itself.

Our construction takes a high-level approach as follows: We consider a sequence of memory writes as occurring within intervals, each consisting of S write operations, where S corresponds to the memory's size. Over the course of these S write operations, we record all updates made to the memory in a designated block. This block contains the initial state of the memory within the relevant interval and an array called $\mathsf{updates}$ of S elements. Each element at location ℓ in the $\mathsf{updates}$ array contains a binary search tree, representing the updates to the memory at location ℓ. These trees contain key-value pairs, where the key signifies the time when the write operation occurred, and the value denotes the bit that was written to the memory. At the start of each interval, all elements in the $\mathsf{updates}$ array are initialized as empty binary search trees. With each subsequent write operation, the update to the memory is added to the corresponding tree. This construction facilitates the ability to restore the memory to any point within the interval efficiently.

To enable memory restoration at any time, we use additional blocks. The scheme commences by initializing a certain number of blocks before the process begins. As previously described, the scheme stores all pertinent data for the interval within a specific block. When the Backup function is executed, the scheme marks the block as saved. After completing S write operations, the scheme searches for an available (unsaved) block and begins updating it. This allows to restore any memory backup at any time.

The problem with this approach lies in the initiation of a new block when transitioning to a new interval, which incurs a time cost of S. To address this issue, we initialize the next block in advance. With each write operation, we also take one step in the initialization process for the next block. Given that we have S write operations within an interval, by the interval's conclusion, the next block is fully initialized and ready for use.

Notation. Let B be a block of data containing $(I, \mathsf{saved}, M, \mathsf{updates}, p^{\mathsf{next}})$, where I is the index of the interval of writes that this block represents, saved is a counter represents how many times the block has been saved, $\mathsf{updates}$ is an array of S binary search trees, and p^{next} is a pointer to some block in the memory. Let Blocks be an array of $\mathsf{m} = 3 \cdot (\mathsf{k}+1)$ blocks, where k is the maximum number of backups that the scheme supports. Let ts be a timestamp indicating that the memory's state is aligned with the state it had at time ts. Note that $\mathsf{Blocks}, \mathsf{ts}, \mathsf{B}_1, \mathsf{B}_2, \mathsf{B}_3$ are global parameters in the stream.

Construction 7.7. The construction of the Mem scheme is as follows:

Global Parameters.

```
– $\mathsf{Blocks} := \perp$.
– $\mathsf{B}_1, \mathsf{B}_2, \mathsf{B}_3 := \perp$.
– $\mathsf{ts} := \perp$.
```

The following function initializes the scheme's global parameters. It iterates through each block in Blocks, setting the value of saved to 0 and initializing $\mathsf{updates}$ as an empty array of binary search trees. Furthermore, specific blocks, namely B_1, B_2, and B_3, are set to be a reference to three specific blocks within Blocks. These three blocks function as the initial storage for memory changes and are flagged as "saved". Additionally, they are initialized with the input memory state M.

$\mathsf{Mem.Init}(S, M, \mathsf{k})$.

```
1. Set $\mathsf{ts} := 0$.
2. Set $\mathsf{Blocks}$ as an array of $\mathsf{m}$ blocks.
3. For each $i \in \{0, \dots, \mathsf{m}-1\}$,
    3.1. Set $\mathsf{Blocks}[i].\mathsf{saved} := 0$.
    3.2. Set $\mathsf{Blocks}[i].\mathsf{updates}$ to be an array of $S$ empty binary search trees.
4. For each $i \in [3]$
    4.1. Set $\mathsf{B}_i := \mathsf{Blocks}[i-1]$ (soft copy).
    4.2. Set $\mathsf{B}_i.\mathsf{saved} := 1$.
    4.3. Set $\mathsf{B}_i.M := M$.
    4.4. Set $I := i$.
    4.5. Set $\mathsf{B}_i.p^{\mathsf{next}} := \mathsf{Blocks}[i \pmod{\mathsf{m}}]$ (soft copy).
```

The following function outputs the memory value at location ℓ. This value is expected to correspond with the memory's state at time ts. To accomplish this, the function identifies the most recent memory update at ℓ that occurred before ts.

$\underline{\mathsf{Mem.Read}(\ell).}$

```
1. Find the largest key in $\mathsf{B}_1.\mathsf{updates}[j]$ that is $\leq \mathsf{ts}$, and return its value.
2. If such key does not exist, return $\mathsf{B}_1.M[j]$.
```

The following function is responsible for modifying the memory at location ℓ with the value b. To achieve this, the function adds the update to the $\mathsf{updates}$ array within B_1. Additionally, the scheme takes a step in initializing the next available block. In the event that B_1 already contains S updates, B_1 is updated to reference the newly initialized block, and the function identifies the new next block accordingly.

$\underline{\mathsf{Mem.Write}(\ell, b).}$

```
1. Set $\mathsf{ts} := \mathsf{ts} + 1$.
2. If $\mathsf{ts} \pmod{S} \equiv 0$, then:
   2.1. Set $\mathsf{B}_1.\mathsf{saved} := \mathsf{B}_1.\mathsf{saved} - 1$.
   2.2. Set $\mathsf{B}_1 := \mathsf{B}_2$ (soft copy).
   2.3. Set $\mathsf{B}_2 := \mathsf{B}_3$ (soft copy).
   2.4. Set $\mathsf{B}_3 := \mathsf{FindFreeBlock}()$ (soft copy).
   2.5. Set $\mathsf{B}_3.I := \mathsf{B}_2.I + 1$.
   2.6. Set $\mathsf{B}_2.p^{\mathsf{next}} := \mathsf{B}_3$ (soft copy).
3. Add the key-value pair $\langle \mathsf{ts}, b \rangle$ to the binary search tree $\mathsf{B}_1.\mathsf{updates}[\ell]$ (ignore duplicates).
4. Set $\mathsf{B}_2.M[\ell] := b$.
5. Do one step in initializing $\mathsf{B}_3$,
   5.1. Set $\mathsf{B}_3.M[\mathsf{ts} - 1 \pmod{S}] := \mathsf{B}_2.M[\mathsf{ts} - 1 \pmod{S}]$.
   5.2. Set $\mathsf{B}_3.\mathsf{updates}[\mathsf{ts} - 1 \pmod{S}]$ to be an empty binary search tree.
```

The following functions has a trivial implementation.

$\underline{\mathsf{Mem.Backup}().}$

```
1. $\mathsf{B}_1.\mathsf{saved} := \mathsf{B}_1.\mathsf{saved} + 1$.
2. $\mathsf{B}_2.\mathsf{saved} := \mathsf{B}_2.\mathsf{saved} + 1$.
3. $\mathsf{B}_3.\mathsf{saved} := \mathsf{B}_3.\mathsf{saved} + 1$.
4. Output $\mathsf{ts}$.
```

$\underline{\mathsf{Mem.DeleteBackup}(\mathsf{ts}').}$

```
1. Set $\mathsf{B}_1' := \mathsf{FindBlock}(\mathsf{ts}')$ (soft copy).
2. Set $\mathsf{B}_2' := \mathsf{B}_1.p^{\mathsf{next}}$ (soft copy).
3. Set $\mathsf{B}_3' := \mathsf{B}_2.p^{\mathsf{next}}$ (soft copy).
4. $\mathsf{B}_1'.\mathsf{saved} := \mathsf{B}_1.\mathsf{saved} - 1$.
5. $\mathsf{B}_2'.\mathsf{saved} := \mathsf{B}_2.\mathsf{saved} - 1$.
6. $\mathsf{B}_3'.\mathsf{saved} := \mathsf{B}_3.\mathsf{saved} - 1$.
```

$\underline{\mathsf{Mem.Restore}(\mathsf{ts}').}$

1. Set $\mathsf{ts} := \mathsf{ts}'$.
2. Set $\mathsf{B}_1 := \mathsf{FindBlock}(\mathsf{ts})$ (soft copy).
3. Set $\mathsf{B}_2 := \mathsf{B}_1.p^{\mathsf{next}}$ (soft copy).
4. Set $\mathsf{B}_3 := \mathsf{B}_2.p^{\mathsf{next}}$ (soft copy).

$\underline{\mathsf{FindFreeBlock}().}$

1. For each block B in Blocks, if $\mathsf{B.saved} = 0$ return B (soft copy).

$\underline{\mathsf{FindBlock}(\mathsf{ts}').}$

1. For each block B of Blocks, if $\mathsf{B}.I = \left\lceil \frac{\mathsf{ts}'}{S} \right\rceil$ return B (soft copy).

Both the efficiency and the correctness comes directly from the construction. In what follows, we will describe some key considerations in the efficiency analysis.

Efficiency. The efficiency parameters are mainly dominated by the following consideration:

- The Init function initializes $O(\mathsf{k})$ blocks of size $O(S)$. This takes time $T^{\mathsf{init}}_{\mathsf{Mem}} = (\mathsf{k} \cdot S)$.
- In one step of the scheme, the dominant operation is a single Write operation, which takes $T_{\mathsf{Mem}} = \mathsf{k} \cdot \mathrm{polylog}(\mathsf{ts}, S)$ time. This includes:
  - $\mathrm{polylog}(\mathsf{ts})$ time for updating the timestamp.
  - $O(\mathsf{k})$ time for finding free block.
  - $\mathrm{polylog}(S)$ time for updating the $\mathsf{updates}$ array. (adding a single element to a binary search tree of size at most S).
  - $\mathrm{polylog}(\mathsf{ts})$ time is spent on one step of the initialization process for the next free block.
- The memory of the scheme is bounded by $S_{\mathsf{Mem}} = \mathsf{k} \cdot S \cdot \mathrm{polylog}(\mathsf{ts})$. This includes $O(\mathsf{k})$ blocks, each of size $S \cdot \mathrm{polylog}(\mathsf{ts})$.

□

7.3 RAM delegation with complexity-preserving prover

In Section 6, we introduced an $\mathcal{L}(\lambda, t) \cdot \mathrm{poly}(\lambda, \log t)$-succinct $\mathsf{RAM\ Delegation}$ scheme with strong soundness for any read and write RAM machine. In this section, we introduce an alternative implementation to the prover from Construction 6.3. The alternative implementation will generate the same proof but in a complexity-preserving way.

One of the challenges in converting the original prover (see Construction 6.3) into a complexity-preserving prover is that the prover stores in memory the list $(\mathbb{w}_1, \ldots, \mathbb{w}_t)$, while our goal is to construct a prover with space that depends poly-logarithmic on t. To overcome this issue, we first construct a stream that will allow a stream rewind access to the witnesses in a complexity preserving way, and then we will construct a seBARG with prover that can generate the proof using a restricted stream rewind access to the witness. In addition to the list of witness, our prover holds a list $(\mathsf{st}_1, \ldots, \mathsf{st}_t)$, and a list $(h_1, \ldots, h_t)$. These t size lists has a similar issue. To overcome this issue, we will describe a stream with a stream rewind access to $(\mathsf{st}_1, \ldots, \mathsf{st}_t)$, and a stream with a stream rewind access to $(h_1, \ldots, h_t)$, and we'll use those streams in order to construct the required data in the proof.

Let $\mathsf{SEH} = (\mathsf{SEH.G}, \mathsf{SEH.H}, \mathsf{SEH.O}, \mathsf{SEH.V}, \mathsf{SEH.E})$ be a secure two-mode somewhere extractable hash with succinct local opening Assume there exists:

- a secure hash tree,

$$\mathsf{HT} = (\mathsf{HT.G}, \mathsf{HT.H}, \mathsf{HT.R}, \mathsf{HT.W}, \mathsf{HT.VR}, \mathsf{HT.VW}) \ .$$

- a secure sparse hash tree,

$$\mathsf{SparseHT} = \\ (\mathsf{SparseHT.G}, \mathsf{SparseHT.H}, \mathsf{SparseHT.R}, \mathsf{SparseHT.W}, \mathsf{SparseHT.VR}, \mathsf{SparseHT.VW}) \ .$$

- a secure *two-mode* somewhere extractable hash with succinct local opening,

$$\mathsf{SEH} = (\mathsf{SEH.G}, \mathsf{SEH.H}, \mathsf{SEH.O}, \mathsf{SEH.V}, \mathsf{SEH.E}) \ .$$

- an $\mathcal{L}$-succinct somewhere extractable batch argument for $\mathsf{BatchCSAT}$,

$$\mathsf{seBARG} = (\mathsf{BARG.G}, \mathsf{BARG.P}, \mathsf{BARG.V}, \mathsf{BARG.E}) \ .$$

In what follows, we assume without loss of generality that t is a power of 2, and let $\mathsf{d} \in \mathbb{N}$ be such that $t = 2^{\mathsf{d}}$.

7.3.1 Constructing rewindable streams.

In this subsection, we construct the required streams for the $\mathsf{RAM\ Delegation}$ prover construction. This includes a rewindable stream for each one of the following lists:

- The list of witnesses: $\mathbb{w}_1, \ldots, \mathbb{w}_t$.
- The list of local states: $\mathsf{st}_1, \ldots, \mathsf{st}_t$.
- The list of the hash of the intermediate configuration of the memory: $h_1, \ldots, h_t$.

Recall that the $\mathsf{RAM\ Delegation}$ prover from Section 6 uses $\mathsf{SparseHT}$ to emulate the machine's memory. As a first step stone, for allowing the rewindable property to the streams that we construct, we wrap the memory of the $\mathsf{SparseHT}$ using the memory scheme presented in Section 7.2 to provide additional functionality of jumping back to a previous state of the memory.

Soft copy. Throughout the subsequent constructions, when we mention a copy of the parameters $\mathsf{T}^{\mathsf{imp}}$, $\mathsf{T}^{\mathsf{Mem}}$ or x^{exp}, we are essentially referring to a *soft-copy*, meaning that only a reference to the instance is duplicated.

Memory management. In the $\mathsf{RAM.P}$ construction, detailed in Section 6, the parameter $\mathsf{T}^{\mathsf{mem}}$ contains sparse-tree representation of the RAM machine's memory. Within this section, we manage the memory of the variable $\mathsf{T}^{\mathsf{mem}}$ using the Mem scheme. The new parameter responsible for handling $\mathsf{T}^{\mathsf{mem}}$'s memory is known as $\mathsf{T}^{\mathsf{Mem}}$. Essentially, $\mathsf{T}^{\mathsf{Mem}}$ is an instance of Mem that encapsulates the memory of $\mathsf{T}^{\mathsf{mem}}$. To access the tree $\mathsf{T}^{\mathsf{mem}}$, one must utilize the read/write functions of the scheme. For instance, in the following code, when executing the command $(\mathsf{T}^{\mathsf{Mem}\prime}, h, \mathsf{auth}) := \mathsf{SparseHT.W}(\mathsf{T}^{\mathsf{Mem}}, \ell, b)$, we provide to $\mathsf{SparseHT.W}$ only a reference to $\mathsf{T}^{\mathsf{Mem}}$, and $\mathsf{T}^{\mathsf{Mem}\prime}$ is simply another reference to the same $\mathsf{T}^{\mathsf{Mem}}$. Consequently, $\mathsf{SparseHT.W}$ effectively modifies the memory contained within $\mathsf{T}^{\mathsf{Mem}}$. Note that $\mathsf{SparseHT.R}, \mathsf{SparseHT.W}$ anticipates a sparse hash tree as input, not an instance of a memory scheme. Therefore, when $\mathsf{SparseHT}$ attempts to read from or write to the memory in $\mathsf{T}^{\mathsf{Mem}}$, we implicitly consider it as if the function were attempting to read from or write to the memory that $\mathsf{T}^{\mathsf{Mem}}$ represents. Thus, we replace the command with $\mathsf{T}^{\mathsf{Mem}}.\mathsf{Read}(\cdot)$ or $\mathsf{T}^{\mathsf{Mem}}.\mathsf{Write}(\cdot)$ as appropriate.

Regarding the efficiency parameters of the memory scheme, to simplify the complexity analysis of all the streaming algorithms later on in the proof, we predefine the efficiency parameter of the Mem scheme as it will be used that way later in our construction. The size of the memory that Mem needs to manage is $|\mathsf{T}^{\mathsf{mem}}|$. Therefore, based on the efficiency parameters of $\mathsf{SparseHT}$, the size of the memory is $s = w \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}})$. Since this data structure essentially manages the memory of the RAM machine, where each read/write operation is emulated by $\mathsf{SparseHT}$, the maximum number of write operations is bounded by $\mathsf{w} \le t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}})$. Later on in the proof, we will observe that the number of backups required for the scheme at a specific time is constant. Consequently, the maximum number of backups in the scheme is $\mathsf{k} = O(1)$. Overall, by the complexity parameters in Claim 7.6, we can determine that the efficiency parameters of $\mathsf{T}^{\mathsf{Mem}}$ are as follows:

$$T^{\mathsf{init}}_{\mathsf{Mem}} = w \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}),\ T_{\mathsf{Mem}} = \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t),\ S_{\mathsf{Mem}} = w \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)\ ,$$

Our initial step in constructing our $\mathsf{RAM.P}$ involves explaining a single operation of the RAM machine, which will be useful when constructing the necessary streams.

```
$\mathsf{RAMSingleStep}(\mathsf{st}_{i-1}, \mathsf{T}^{\mathsf{Mem}}, \mathsf{T}^{\mathsf{imp}}, x^{\mathsf{exp}})$.
    1. Set $(\mathsf{type}_i, \ell_i^{\mathsf{inp}}) := \mathsf{StepInputRead}(\mathsf{st}_{i-1})$.
    2. If $\mathsf{type} = \mathsf{imp}$ then set $(z_i^{\mathsf{inp}}, \mathsf{auth}_i^{\mathsf{inp}}) := \mathsf{HT.R}(\mathsf{T}^{\mathsf{imp}}, \ell_i^{\mathsf{inp}})$.
    3. If $\mathsf{type} = \mathsf{exp}$ then set $z_i^{\mathsf{inp}} := (x^{\mathsf{exp}})_{\ell_i^{\mathsf{inp}}}$, $\mathsf{auth}_i^{\mathsf{inp}} := \perp$.
    4. Set $\ell_i^{\mathsf{mr}} := \mathsf{StepMemRead}(\mathsf{st}_{i-1}, z_i^{\mathsf{inp}})$.
    5. Set $(z_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mr}}) := \mathsf{SparseHT.R}(\mathsf{T}^{\mathsf{Mem}}, \ell_i^{\mathsf{mr}})$.
    6. Set $(z_i^{\mathsf{mw}}, \ell_i^{\mathsf{mw}}, \mathsf{st}_i) := \mathsf{StepMemWrite}(\mathsf{st}_{i-1}, z_i^{\mathsf{inp}}, z_i^{\mathsf{mr}})$.
    7. Set $(\mathsf{T}^{\mathsf{Mem}}, h_i, \mathsf{auth}_i^{\mathsf{mw}}) := \mathsf{SparseHT.W}(\mathsf{T}^{\mathsf{Mem}}, \ell_i^{\mathsf{mw}}, z_i^{\mathsf{mw}})$.
    8. Output $(\mathsf{st}_i, h_i, \mathsf{T}^{\mathsf{Mem}}, z_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mw}}, z_i^{\mathsf{inp}}, \mathsf{auth}_i^{\mathsf{inp}})$.
```

Complexity: The running time of the function is dominated by $\mathsf{HT.R}$ (Item 2) and $\mathsf{SparseHT.W}$ (Item 7). By the efficiency of the HT scheme we get that Item 2 takes time $\log(|x^{\mathsf{imp}}|) \cdot \mathrm{poly}(\lambda)$. By the efficiency of the $\mathsf{SparseHT}$ scheme, $\mathsf{SparseHT.W}$ has $\mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}})$ steps. Since Mem is handling some of the steps, by Claim 7.6 we can bound the running time of Item 7 by $\mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$. Overall, we get that the running and space are bounded by $\mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.

The following stream is a stream for a single step of the RAM machine. Each element in the stream essentially contains all the information required for validating that a single RAM step was executed correctly.

Construction 7.8. The construction of the $\mathsf{Stm}^{\mathsf{step}}$ scheme is as follows:

<u>Global Parameters:</u>

- $\mathsf{backups}$ (dictionary).
- $\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}$.
- x^{exp} (string).
- $\mathsf{T}^{\mathsf{Mem}}$ (instance of Mem).
- $\mathsf{T}^{\mathsf{imp}}$ (Merkle-tree).
- elem.

<u>$\mathsf{Init}(x, \mathsf{k})$:</u>

```
1. Parse $x := (\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s)$.
2. Compute $(\mathsf{T}^{\mathsf{imp}}, \mathsf{d}^{\mathsf{imp}}) := \mathsf{HT.H}(\mathsf{HT.hk}, x^{\mathsf{imp}})$.
3. Compute $(\mathsf{T}^{\mathsf{mem}}, h_0) := \mathsf{SparseHT.H}(\mathsf{SparseHT.hk}, 2^{\mathsf{L}_{\mathsf{st}}}, \emptyset)$.
4. Execute $\mathsf{T}^{\mathsf{Mem}}.\mathsf{Init}(s, \mathsf{T}^{\mathsf{mem}}, \mathsf{k})$.
5. Execute $\mathsf{backups}$ as an empty dictionary.
6. Set $\mathsf{st}_0$ to be the initialize state of $M$.
7. Compute $(\mathsf{st}_1, h_1, \mathsf{T}^{\mathsf{Mem}}, z_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mw}}, z_1^{\mathsf{inp}}, \mathsf{auth}_1^{\mathsf{inp}}) :=$
   $\mathsf{RAMSingleStep}(\mathsf{st}_i, \mathsf{T}^{\mathsf{Mem}}, \mathsf{T}^{\mathsf{imp}}, x^{\mathsf{exp}})$.
8. Set $\mathsf{elem} := (1, \mathsf{st}_0, \mathsf{st}_1, h_0, h_1, z_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mw}}, z_1^{\mathsf{inp}}, \mathsf{auth}_1^{\mathsf{inp}})$.
```

<u>$\mathsf{Next}()$:</u>

```
1. Parse $\mathsf{elem} := (i, \mathsf{st}_{i-1}, \mathsf{st}_i, h_{i-1}, h_i, z_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mw}}, z_i^{\mathsf{inp}}, \mathsf{auth}_i^{\mathsf{inp}})$.
2. Compute $(\mathsf{st}_{i+1}, h_{i+1}, \mathsf{T}^{\mathsf{Mem}}, z_{i+1}^{\mathsf{mr}}, \mathsf{auth}_{i+1}^{\mathsf{mr}}, \mathsf{auth}_{i+1}^{\mathsf{mw}}, z_{i+1}^{\mathsf{inp}}, \mathsf{auth}_{i+1}^{\mathsf{inp}}) :=$
   $\mathsf{RAMSingleStep}(\mathsf{st}_i, \mathsf{T}^{\mathsf{Mem}}, \mathsf{T}^{\mathsf{imp}}, x^{\mathsf{exp}})$.
3. Set $\mathsf{elem} := (i+1, \mathsf{st}_i, \mathsf{st}_{i+1}, h_i, h_{i+1}, z_{i+1}^{\mathsf{mr}}, \mathsf{auth}_{i+1}^{\mathsf{mr}}, \mathsf{auth}_{i+1}^{\mathsf{mw}}, z_{i+1}^{\mathsf{inp}}, \mathsf{auth}_{i+1}^{\mathsf{inp}})$.
```

<u>$\mathsf{Backup}()$:</u>

```
1. Parse $\mathsf{elem} := (\mathsf{i}, \mathsf{st}_{i-1}, \mathsf{st}_i, h_{i-1}, h_i, z_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mw}}, z_i^{\mathsf{inp}}, \mathsf{auth}_i^{\mathsf{inp}})$.
2. Execute $\mathsf{ts} := \mathsf{T}^{\mathsf{Mem}}.\mathsf{Backup}()$.
3. Set $\mathsf{backups[i]} := (\mathsf{elem}, \mathsf{ts})$.
4. Output $i$.
```

<u>$\mathsf{DeleteBackup(i)}$:</u>

```
1. Parse $\mathsf{backups[i]} := (\mathsf{elem}', \mathsf{ts})$.
2. Execute $\mathsf{T}^{\mathsf{Mem}}.\mathsf{DeleteBackup}(\mathsf{ts})$.
3. Delete $\mathsf{backups[i]}$.
```

<u>$\mathsf{Jump(i)}$:</u>

```
1. Parse $\mathsf{backups[i]} := (\mathsf{elem}', \mathsf{ts})$.
2. Execute $\mathsf{T}^{\mathsf{Mem}}.\mathsf{Restore}(\mathsf{ts})$.
3. Set $\mathsf{elem} := \mathsf{elem}'$.
```

$\underline{\textsf{GetData}():}$

1. Output $\textsf{elem}$.

To simplify the complexity analysis of all the streaming algorithms, we treat $\textsf{k}$ as a constant, as it will be used in that way later in our construction.

Complexity: The running time of a single step is dominated by the running time of $\textsf{RAMSingleStep}$, and the space is dominated by the global parameters $T_{\textsf{Mem}}$ and $\textsf{T}^{\textsf{imp}}$. Taking into account the efficiency parameters of the $\textsf{HT.H}$, $\textsf{Mem}$ schemes, and the efficiency of $\textsf{RAMSingleStep}$, we can deduce the following:

- The running time of $\textsf{Init}$ is bounded by $|x^{\textsf{imp}}| \cdot \mathrm{poly}(\lambda) + s \cdot \mathrm{poly}(\lambda, \textsf{L}_{\textsf{st}}, \log t)$.
- The running time of one stream operation is bounded by $T^{\textsf{step}}_{\textsf{Stm}} = \mathrm{poly}(\lambda, \textsf{L}_{\textsf{st}}, \log t)$.
- The running time of one pass over the entire stream is bounded by $\overrightarrow{T}^{\textsf{step}}_{\textsf{Stm}} = t \cdot \mathrm{poly}(\lambda, \textsf{L}_{\textsf{st}}, \log t)$.
- The space used in the stream is bounded by $S^{\textsf{step}}_{\textsf{Stm}} = |x^{\textsf{imp}}| \cdot \mathrm{poly}(\lambda) + s \cdot \mathrm{poly}(\lambda, \textsf{L}_{\textsf{st}}, \log t)$.

Let $\textsf{Stm}^{\textsf{st}}$ be the stream for the list $(\textsf{st}_1, \ldots, \textsf{st}_t)$, and let $\textsf{Stm}^h$ be the stream for the list $(h_1, \ldots, h_t)$. We omit the construction of $\textsf{Stm}^{\textsf{st}}, \textsf{Stm}^h$ since it does essentially the same as the stream $\textsf{Stm}^{\textsf{step}}$. The only difference is that $\textsf{Stm}^{\textsf{st}}.\textsf{GetData}()$ will only output the local state $\textsf{st}$, and $\textsf{Stm}^h.\textsf{GetData}()$ will only output the hash of the memory h (rather than the entire element's data).

Another challenge we face in constructing our complexity-preserving prover, which arises from the inability to hold the data $(\textsf{st}_1, \ldots, \textsf{st}_t)$ in memory, is the computation of the hash of $(\textsf{st}_1, \ldots, \textsf{st}_t)$. We need to compute $\textsf{SEH}$ for the data using only streaming access to the data. Furthermore, due to the same limitation that prevents us from holding $(\textsf{st}_1, \ldots, \textsf{st}_t)$ in memory, we are also unable to store the openings $(\rho_{\textsf{st},1}, \ldots, \rho_{\textsf{st},t})$ to the hash. To address these challenges, we need $\textsf{SEH}$ with the following properties:

1. The hash of the input can be efficiently computed using only streaming access to the input.
2. A stream of openings to the hash can be efficiently computed using only streaming access to the input.

Fortunately, Okamoto et al. [OPWW15] constructed $\textsf{SEH}$ that satisfies those properties. In more detail, they demonstrated that given a two-mode hash family, using a Merkle tree with a two-mode hash function as its gates, we obtain a succinct $\textsf{SEH}$ (Lemma 3.2). The hash function outputs the root of the Merkle tree, and the openings are represented by the authentication paths in the tree. Now, we need to show that this construction fulfills the aforementioned requirements.

To meet our requirements, we first need to ensure that the hash function can be computed efficiently. The hash function outputs the root of the Merkle tree, and this can be computed using the $\textsf{ComputeTreeHash}$ introduced in Section 7.1. In this case, the hash function sent to the function is a two-mode hash function. Secondly, we need to compute a stream of authentication paths efficiently, while adhering to the constraint of streaming access to the input. This can be achieved by first generating the authentication path for the first bit in the input using $\textsf{ComputeAuthPath}$. Subsequently, we can emulate $\textsf{ComputeNextAuthPath}$ for all the other authentication paths. For more details on the complexity of $\textsf{ComputeTreeHash}$, $\textsf{ComputeAuthPath}$, and $\textsf{ComputeNextAuthPath}$, please refer to Section 7.1.

We are now ready to construct a stream of $\textsf{SEH}$ openings with respect to the hashed data $(\textsf{st}_1, \ldots, \textsf{st}_t)$.

Construction 7.9. The construction of the $\mathsf{Stm}^{\rho_{\mathsf{st}}}$ scheme is as follows:
Global Parameters:

- $\mathsf{backups}$ (dictionary).
- $\mathsf{stm}^{\mathsf{st}}$ (instance of $\mathsf{Stm}^{\mathsf{st}}$).
- t.
- H (hash function).
- elem.

Init(x, k):

```
1. Parse $x := (\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, t)$.
2. Set $\mathsf{H} := \mathsf{SEH.H}(\mathsf{SEH.hk}, \cdot)$.
3. Execute $\mathsf{stm}^{\mathsf{st}}.\mathsf{Init}(\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, \mathsf{k})$.
4. Compute $\mathsf{auth}_1 := \mathsf{ComputeAuthPath}(\mathsf{H}, \mathsf{stm}^{\mathsf{st}}, 1, t)$ (soft-copy).
5. Set $\mathsf{elem} := (1, \mathsf{auth}_1)$.
```

Next():

```
1. Parse $\mathsf{elem} := (\mathsf{i}, \mathsf{auth}_i)$.
2. Compute $\mathsf{auth}_{i+1} := \mathsf{ComputeNextAuthPath}(\mathsf{H}, \mathsf{stm}^{\mathsf{st}}, \mathsf{i}, \mathsf{auth}_i, t)$ (soft-copy).
3. Compute $\mathsf{stm}^{\mathsf{st}}.\mathsf{Next}()$.
4. Set $\mathsf{elem} := (\mathsf{i} + 1, \mathsf{auth}_{i+1})$.
```

Backup():

```
1. Parse $\mathsf{elem} := (\mathsf{i}, \mathsf{auth}_i)$.
2. Execute $\mathsf{stm}^{\mathsf{st}}.\mathsf{Backup}()$.
3. Set $\mathsf{backups}[\mathsf{i}] := \mathsf{elem}$.
4. Output $\mathsf{i}$.
```

DeleteBackup(i):

```
1. Execute $\mathsf{stm}^{\mathsf{st}}.\mathsf{DeleteBackup}(\mathsf{i})$.
2. Delete $\mathsf{backups}[\mathsf{i}]$.
```

Jump(i):

```
1. Execute $\mathsf{stm}^{\mathsf{st}}.\mathsf{Jump}(\mathsf{i})$.
2. Set $\mathsf{elem} := \mathsf{backups}[\mathsf{i}]$.
```

GetData():

```
1. Parse $\mathsf{elem} := (\mathsf{i}, \mathsf{auth}_i)$.
2. Output $\mathsf{auth}_i$.
```

Complexity: For one pass over the entire stream, we follow the complexity analysis of StreamAllAuthPaths (see Section 7.1). The rest of the analysis follows directly from the complexity of the algorithms we use. The complexity we get is:

- The running time of Init is bounded by $|x^{\mathsf{imp}}| \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- The running time of one stream operation is bounded by $T^{\rho_{\mathsf{st}}}_{\mathsf{Stm}} = t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- The running time of one pass over the entire stream is bounded by $\overrightarrow{T}^{\rho_{\mathsf{st}}}_{\mathsf{Stm}} = t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.

- The space used in one stream operation is bounded by $S^{\rho_{\mathsf{st}}}_{\mathsf{Stm}} = |x^{\mathsf{imp}}| \cdot \text{poly}(\lambda) + s \cdot \text{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.

Let Stm^{ρ_h} denote the stream of SEH openings corresponding to the hashed data $(h_1, \ldots, h_t)$. We will not provide the detailed construction of Stm^{ρ_h} since it follows the same approach as $\mathsf{Stm}^{\rho_{\mathsf{st}}}$, with the only difference being the use of Stm^h instead of $\mathsf{Stm}^{\mathsf{st}}$ in the construction.

We will now proceed to construct a stream of witnesses to be used when generating the seBARG proof. Each individual witness has the following structure:

$$\mathbb{w}_i := (\mathsf{st}_{i-1}, \mathsf{st}_i, h_{i-1}, h_i, z_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mr}}, \mathsf{auth}_i^{\mathsf{mw}}, z_i^{\mathsf{inp}}, \mathsf{auth}_i^{\mathsf{inp}}, \rho_{\mathsf{st},i-1}, \rho_{\mathsf{st},i}, \rho_{h,i-1}, \rho_{h,i}) \ .$$

Construction 7.10. The construction of the $\mathsf{Stm}^{\mathbb{w}}$ scheme is as follows:

Global Parameters:

- $\mathsf{stm}^{\mathsf{step}}$ (instance of $\mathsf{Stm}^{\mathsf{step}}$).
- $\mathsf{stm}^{\rho_{\mathsf{st}}}$ (instance of $\mathsf{Stm}^{\rho_{\mathsf{st}}}$).
- stm^{ρ_h} (instance of Stm^{ρ_h}).

$\mathsf{Init}(x, \mathsf{k})$:

1. Parse $x := (\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, t)$.
2. Execute $\mathsf{stm}^{\mathsf{step}}.\mathsf{Init}(\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, \mathsf{k})$.
3. Execute $\mathsf{stm}^{\rho_{\mathsf{st}}}.\mathsf{Init}(\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, t, \mathsf{k})$.
4. Execute $\mathsf{stm}^{\rho_h}.\mathsf{Init}(\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_2, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, t, \mathsf{k})$.
5. Set $(i, \mathsf{st}_1, h_1, z_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mw}}, z_1^{\mathsf{inp}}, \mathsf{auth}_1^{\mathsf{inp}}) := \mathsf{stm}^{\mathsf{step}}.\mathsf{GetData}()$.
6. Set $\rho_{\mathsf{st},0}, \rho_{h,0} := \perp$.
7. Set $\mathsf{elem} := (1, \mathsf{st}_0, \mathsf{st}_1, h_0, h_1, z_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mr}}, \mathsf{auth}_1^{\mathsf{mw}}, z_1^{\mathsf{inp}}, \mathsf{auth}_1^{\mathsf{inp}}, \rho_{\mathsf{st},0}, \rho_{h,0})$.

$\mathsf{Next}()$:

1. Execute $\mathsf{stm}^{\mathsf{step}}.\mathsf{Next}()$.
2. Execute $\mathsf{stm}^{\rho_{\mathsf{st}}}.\mathsf{Next}()$.
3. Execute $\mathsf{stm}^{\rho_h}.\mathsf{Next}()$.

$\mathsf{Backup}()$:

1. Execute $\mathsf{i} := \mathsf{stm}^{\mathsf{step}}.\mathsf{Backup}()$.
2. Execute $\mathsf{stm}^{\rho_{\mathsf{st}}}.\mathsf{Backup}()$.
3. Execute $\mathsf{stm}^{\rho_h}.\mathsf{Backup}()$.
4. Output i.

$\mathsf{DeleteBackup}(\mathsf{i})$:

1. Execute $\mathsf{stm}^{\mathsf{step}}.\mathsf{DeleteBackup}(\mathsf{i})$.
2. Execute $\mathsf{stm}^{\rho_{\mathsf{st}}}.\mathsf{DeleteBackup}(\mathsf{i})$.
3. Execute $\mathsf{stm}^{\rho_h}.\mathsf{DeleteBackup}(\mathsf{i})$.

$\mathsf{Jump}(\mathsf{i})$:

1. Execute $\mathsf{stm}^{\mathsf{step}}.\mathsf{Jump}(\mathsf{i})$.
2. Execute $\mathsf{stm}^{\rho_{\mathsf{st}}}.\mathsf{Jump}(\mathsf{i})$.
3. Execute $\mathsf{stm}^{\rho_h}.\mathsf{Jump}(\mathsf{i})$.

GetData():

1. Output ($\mathsf{stm}^{\mathsf{step}}.\mathsf{GetData}(), \mathsf{stm}^{\rho_{\mathsf{st}}}.\mathsf{GetData}(), \mathsf{stm}^{\rho_h}.\mathsf{GetData}()$).

Complexity: The complexity of the stream is dominated by the complexity of the streams $\mathsf{Stm}^{\rho_{\mathsf{st}}}$, Stm^{ρ_h}. Therefore, by the complexity of the streams $\mathsf{Stm}^{\rho_{\mathsf{st}}}, \mathsf{Stm}^{\rho_h}$, we get that:

- The running time of Init is bounded by $|x^{\mathsf{imp}}| \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- The running time of one stream operation is bounded by $T^{\mathtt{w}}_{\mathsf{Stm}} = t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- The running time of one pass over the entire stream is bounded by $\overrightarrow{T}^{\mathtt{w}}_{\mathsf{Stm}} = t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.
- The space used in one stream operation is bounded by $S^{\mathtt{w}}_{\mathsf{Stm}} = |x^{\mathsf{imp}}| \cdot \mathrm{poly}(\lambda) + s \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.

Note that the efficiency parameter $\overrightarrow{T}^{\rho_{\mathsf{st}}}_{\mathsf{Stm}}$ implies that, if we bound the number of accesses to a single stream element to be polylog(t), the average time required for a Next operation is bounded by $\mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$. This is a significant improvement compared to the original estimate, where the time for a single stream operation is $t \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}}, \log t)$.

7.3.2 Constructing complexity-preserving prover.

In this subsection, we construct our $\mathsf{RAM\ Delegation}$ prover and use the rewindable streams from Appendix 7.3.1 to achieve complexity-preserving.

In our alternative $\mathsf{RAM.P}$, we use a special seBARG with an efficient prover that receives the witness list as a stream. While [KLVW22] constructed a succinct seBARG, their primary focus was not on the efficiency of the prover. We use the seBARG constructed by [KLVW22] but provide an alternative prover implementation that achieves the same output.

Claim 7.11. Assume there exists a $\mathrm{poly}(\lambda, \log k)$-succinct somewhere extractable batch argument for $\mathsf{BatchCSAT}$, then there exists a $\mathrm{poly}(\lambda, \log k)$-succinct somewhere extractable batch argument for $\mathsf{BatchCSAT}$ with a prover that has streaming access to the witness list (rather than explicit access), and has the following properties:

1. The prover's time complexity is $\overrightarrow{T}_{\mathsf{Stm}} \cdot \mathrm{poly}(\lambda, |C|, \log k)$.

2. The prover's space complexity is $S_{\mathsf{Stm}} \cdot \mathrm{poly}(\lambda, |C|, \log k)$.

3. The maximum number of stream backups that the prover requires the stream to hold at any time is at most 3.

Where $\overrightarrow{T}_{\mathsf{Stm}}$ and S_{Stm} are the efficiency parameters of the stream, and m is the size of the witness.

We only provide an intuition for the new prover implementation since it uses the same streaming tools and algorithms we already discussed in the construction of our alternative RAM prover.

The initial step of the [KLVW22] prover involves computing k distinct RAM proofs. For each $j \in [k]$, the RAM prover, denoted as $\mathsf{RAM.P}$, generates a proof π^0_j demonstrating that $C(j, \mathtt{w}_j) = 1$. In order to maintain our space complexity, we construct a stream of proofs, rather than holding the proofs explicitly in the memory. Next, the prover hashes the proofs $(\pi^0_1, \dots, \pi^0_k)$ into v_0, and computes a small number of openings in the hash. This we can do using similar ideas to what we demonstrated in $\mathsf{ComputeTreeHash}$ and $\mathsf{ComputeAuthPath}$ algorithms. Then, for every $\ell \in [\log k]$, they define a circuit C^ℓ such that $C^\ell(\pi^{\ell-1}_{2j-1}, \pi^{\ell-1}_{2j}) = 1$ if and only if the $\mathsf{RAM.V}$ accepts both of the proofs, relative to the hashed input $v_{\ell-1}$. Next, $\mathsf{RAM.P}$ generates a proof π^ℓ_j demonstrating that $C^{\ell}(\pi^{\ell-1}_{2j-1}, \pi^{\ell-1}_{2j}) = 1$. The prover hashes the proofs $(\pi^{\ell}_1, \ldots, \pi^{\ell}_{k/2^{\ell}})$. Here, again, we construct a stream of proofs $(\pi^{\ell}_1, \ldots, \pi^{\ell}_{k/2^{\ell}})$ rather than holding the proofs in memory, and compute the hash of the proofs efficiently using the stream. The BARG proof is then $\pi = (v_1, \rho_1 \ldots, v_{\log k}, \rho_{\log k}, \pi^{\log k}_1)$.

The main steps are as follows:

Generating RAM proofs: Initially, the prover computes k distinct RAM proofs. For each $j \in [k]$, the RAM prover, denoted as $\mathsf{RAM.P}$, generates a proof π^1_j to demonstrate that $C(j, \mathbb{w}_j) = 1$. To maintain our space complexity, these proofs are not held explicitly in memory but are instead managed as a stream.

Hashing proofs: The prover hashes the proofs $(\pi^1_1, \ldots, \pi^1_k)$ into v_1 and computes a small number of openings in the hash ρ_1. This step uses similar ideas to what we demonstrated in the $\mathsf{ComputeTreeHash}$ and $\mathsf{ComputeAuthPath}$ algorithms.

Iterative stream generation: For each $\ell \in [\log k]$, the prover defines a circuit $C^{\ell+1}$ such that $C^{\ell+1}(\pi^{\ell}_{2j-1}, \pi^{\ell}_{2j}) = 1$ if and only if the $\mathsf{RAM.V}$ accepts both of the proofs concerning the hashed input v_{ℓ}. Then, $\mathsf{RAM.P}$ generates a proof π^{ℓ}_j demonstrating that $C^{\ell+1}(\pi^{\ell}_{2j-1}, \pi^{\ell}_{2j}) = 1$. Similar to the previous step, these proofs are managed as a stream, and their hash is computed efficiently using the stream.

Final BARG Proof: The BARG proof is $\pi = (v_1, \rho_1 \ldots, v_{\log k}, \rho_{\log k}, \pi^{\log k}_1)$.

It is important to clarify that at each level $\ell \in [\log k]$, $\mathsf{RAM.P}$ conceptually considers all the proofs $(\pi^{\ell}_1, \ldots, \pi^{\ell}_{k/2^{\ell}})$ as implicit inputs and an index j as an explicit input. However, in practice, the prover only requires two proofs, specifically π^{ℓ}_{2j-1} and π^{ℓ}_{2j}, along with the openings to these proofs relative to the hash v_{ℓ} of the implicit input. Therefore, for simplicity, we assume that $\mathsf{RAM.P}$ takes only the relevant proofs as input, along with the corresponding openings, and this is sufficient for emulating the original $\mathsf{RAM.P}$ used in the [KLVW22] construction. Back to constructing $\mathsf{RAM.P}$, in the prover's construction, we use the seBARG from Claim 7.11.

$\underline{\mathsf{RAM.P}(\mathsf{crs}, x^{\mathsf{imp}}, x^{\mathsf{exp}}).}$

```
1. Parse $\mathsf{crs} := (\lambda, \mathsf{BARG.crs}, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, t)$.
2. Set $\mathsf{H}_1 := \mathsf{SEH.H}(\mathsf{SEH.hk}_1, \cdot)$.
3. Set $\mathsf{H}_2 := \mathsf{SEH.H}(\mathsf{SEH.hk}_2, \cdot)$.
4. Set $s = w \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}})$, the size of $\mathsf{T}^{\mathsf{mem}}$. [11]
5. Set $\mathsf{k} = 3$. [12]
6. Compute the following:
   6.1. $\mathsf{stm}^{\mathsf{st}} := \mathsf{Stm}^{\mathsf{st}}.\mathsf{Init}(\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, \mathsf{k})$.
   6.2. $\mathsf{stm}^{h} := \mathsf{Stm}^{h}.\mathsf{Init}(\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, \mathsf{k})$.
   6.3. $\mathsf{stm}^{\mathbb{w}} :=$
        $\mathsf{Stm}^{\mathbb{w}}.\mathsf{Init}(\lambda, \mathsf{HT.hk}, \mathsf{SparseHT.hk}, \mathsf{SEH.hk}_1, \mathsf{SEH.hk}_2, x^{\mathsf{imp}}, x^{\mathsf{exp}}, s, t, \mathsf{k})$.
   6.4. $\mathsf{rt}_{\mathsf{st}} := \mathsf{ComputeTreeHash}(\mathsf{H}_1, \mathsf{stm}^{\mathsf{st}}, t)$.
   6.5. $\mathsf{rt}_{h} := \mathsf{ComputeTreeHash}(\mathsf{H}_2, \mathsf{stm}^{h}, t)$.
```

[11]Recall that s represents the memory size required for emulating the machine's large read-write memory using a sparse hash tree. In fact, s can be computed by emulating the RAM machine using a sparse hash tree and required space $w \cdot \mathrm{poly}(\lambda, \mathsf{L}_{\mathsf{st}})$.

[12]Note that the maximum number of stored backups is determined by the backup operations performed in the Merkle tree procedures from Section 7.1. As mentioned in Remark 7.4, this results in at most 3 backups stored at any time.

6.6. $\mathsf{BARG}.\pi := \mathsf{BARG.P}(\mathsf{BARG.crs}, C, \mathsf{stm}^{\mathbb{W}}, t)$.

7. Output $(out, \pi = (\mathsf{BARG}.\pi, \mathsf{rt_{st}}, \mathsf{rt}_h))$.

Complexity: The running time and space complexity are dominated by the initiation running time of the stream $\mathsf{stm}^{\mathbb{W}}$, and the complexity of the BARG prover relative to the stream of witnesses $\mathsf{stm}^{\mathbb{W}}$. Given the complexity of the BARG and $\mathsf{stm}^{\mathbb{W}}$, and considering that $|C| \leq \mathrm{poly}(\lambda, \mathsf{L_{st}}, \log t)$, we get that:

- The running time of the RAM prover is bounded by $|x^{\mathsf{imp}}| \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda, \mathsf{L_{st}}, \log t)$.
- The space used by the RAM prover is bounded by $|x^{\mathsf{imp}}| \cdot \mathrm{poly}(\lambda) + w \cdot \mathrm{poly}(\lambda, \mathsf{L_{st}}, \log t)$.

Acknowledgments

We thank Eylon Yogev and Gal Arnon for invaluable and useful discussions.

References

[ALMSS98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. “Proof Verification and the Hardness of Approximation Problems”. In: *J. ACM* 45.3 (1998), pp. 501–555.

[ARW17] Amir Abboud, Aviad Rubinstein, and R. Ryan Williams. “Distributed PCP Theorems for Hardness of Approximation in P”. In: *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*. Ed. by Chris Umans. IEEE Computer Society, 2017, pp. 25–36.

[BDFH09] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. “On problems without polynomial kernels”. In: *Journal of Computer and System Sciences* 75.8 (2009), pp. 423–434. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/j.jcss.2009.04.001`. URL: `https://www.sciencedirect.com/science/article/pii/S0022000009000282`.

[BGHSV05] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil P. Vadhan. “Short PCPs Verifiable in Polylogarithmic Time”. In: *20th Annual IEEE Conference on Computational Complexity (CCC 2005), 11-15 June 2005, San Jose, CA, USA*. IEEE Computer Society, 2005, pp. 120–134.

[BR22] Liron Bronfman and Ron D. Rothblum. “PCPs and Instance Compression from a Cryptographic Lens”. In: *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022, Berkeley, CA, USA*. Ed. by Mark Braverman. Vol. 215. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 30:1–30:19.

[CGJJZ23] Arka Rai Choudhuri, Sanjam Garg, Abhishek Jain, Zhengzhong Jin, and Jiaheng Zhang. “Correlation Intractability and SNARGs from Sub-exponential DDH”. In: *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14084. Lecture Notes in Computer Science. Springer, 2023, pp. 635–668.

[CGLRR19] Lijie Chen, Shafi Goldwasser, Kaifeng Lyu, Guy N. Rothblum, and Aviad Rubinstein. "Fine-grained Complexity Meets IP = PSPACE". In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*. Ed. by Timothy M. Chan. SIAM, 2019, pp. 1–20.

[CJJ21] Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. "SNARGs for $\mathcal{P}$ from LWE". In: *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. IEEE, 2021, pp. 68–79.

[DPP16] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. "Efficient Sparse Merkle Trees - Caching Strategies and Secure (Non-)Membership Proofs". In: *Secure IT Systems - 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings*. Ed. by Billy Bob Brumley and Juha Röning. Vol. 10014. Lecture Notes in Computer Science. 2016, pp. 199–215.

[Dru15] Andrew Drucker. "New Limits to Classical and Quantum Instance Compression". In: *SIAM J. Comput.* 44.5 (2015), pp. 1443–1479.

[FS08] Lance Fortnow and Rahul Santhanam. "Infeasibility of instance compression and succinct PCPs for NP". In: *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*. Ed. by Cynthia Dwork. ACM, 2008, pp. 133–142.

[HN10] Danny Harnik and Moni Naor. "On the Compressibility of NP Instances and Cryptographic Applications". In: *SIAM J. Comput.* 39.5 (2010), pp. 1667–1713.

[JKKZ21] Ruta Jawale, Yael Tauman Kalai, Dakshita Khurana, and Rachel Yun Zhang. "SNARGs for bounded depth computations and PPAD hardness from sub-exponential LWE". In: *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*. Ed. by Samir Khuller and Virginia Vassilevska Williams. ACM, 2021, pp. 708–721.

[KLVW22] Yael Tauman Kalai, Alex Lombardi, Vinod Vaikuntanathan, and Daniel Wichs. "Boosting Batch Arguments and RAM Delegation". In: *IACR Cryptol. ePrint Arch.* (2022), p. 1320.

[KPY19] Yael Tauman Kalai, Omer Paneth, and Lisa Yang. "How to delegate computations publicly". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. Ed. by Moses Charikar and Edith Cohen. ACM, 2019, pp. 1115–1124.

[KR09] Yael Tauman Kalai and Ran Raz. "Probabilistically Checkable Arguments". In: *Proceedings of the 29th Annual International Cryptology Conference*. CRYPTO '09. 2009, pp. 143–159.

[KVZ21] Yael Tauman Kalai, Vinod Vaikuntanathan, and Rachel Yun Zhang. "Somewhere Statistical Soundness, Post-Quantum Security, and SNARGs". In: *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part I*. Ed. by Kobbi Nissim and Brent Waters. Vol. 13042. Lecture Notes in Computer Science. Springer, 2021, pp. 330–368.

[Mer87] Ralph C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology - CRYPTO ’87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 369–378.

[OPWW15] Tatsuaki Okamoto, Krzysztof Pietrzak, Brent Waters, and Daniel Wichs. “New Realizations of Somewhere Statistically Binding Hashing and Positional Accumulators”. In: *Advances in Cryptology - ASIACRYPT 2015*. Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9452. Lecture Notes in Computer Science. Springer, 2015, pp. 121–145.

[Szy04] Michael Szydlo. “Merkle Tree Traversal in Log Space and Time”. In: *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. Lecture Notes in Computer Science. Springer, 2004, pp. 541–554.

[WW22] Brent Waters and David J. Wu. “Batch Arguments for NP and More from Standard Bilinear Group Assumptions”. In: *IACR Cryptol. ePrint Arch.* (2022), p. 336.

[Zim02] Marius Zimand. “Probabilistically Checkable Proofs the Easy Way”. In: *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002), August 25-30, 2002, Montreal, Quebec, Canada*. Ed. by Ricardo A. Baeza-Yates, Ugo Montanari, and Nicola Santoro. Vol. 223. IFIP Conference Proceedings. Kluwer, 2002, pp. 337–351.