

TENSEUP: Doubly-Efficient Lookup Arguments from Tensor Codes and Applications to Dynamic SNARKs

Christodoulos Pappas
Hong Kong University of
Science and Technology
cpappas@connect.ust.hk

Pariya Akhiani
University of Illinois
Urbana-Champaign
akhiani2@illinois.edu

Yupeng Zhang
University of Illinois
Urbana-Champaign
zhangyp@illinois.edu

Dimitrios Papadopoulos
Hong Kong University of
Science and Technology
dipapado@cse.ust.hk

Abstract

Doubly-Efficient Lookup Arguments are specialized argument systems that enable succinctly proving the correctness of a batch of k lookup accesses to a large table $\mathbf{x}$ in time proportional to k and *sublinear* in $|\mathbf{x}|$. All existing such schemes suffer from one of the following limitations. Either they rely on elliptic-curve-based assumptions for their security, hence they are not secure against quantum adversaries, or are rather impractical, as they rely on algorithms for fast polynomial evaluations (that are mostly of theoretical interest), or operate over fields of small characteristic—making them incompatible with most modern argument systems. In this work, we use *tensor codes* to construct transparent and post-quantum secure updatable doubly-efficient lookup arguments that asymptotically outperform all existing schemes with the same properties, in terms of prover time and/or preprocessing. By further optimizing this approach for a specific type of two-dimensional tensor codes, we build TENSEUP, a *concretely* efficient lookup argument that, as shown in our experiments, has comparable or even better performance than state-of-the-art non-post-quantum secure prior doubly-efficient lookup arguments. As an independent contribution, we propose a general-purpose compiler for building *dynamic SNARKs* from any “static” SNARK, via judicious use of updatable doubly efficient lookup arguments to store and access witness updates. Applying our compiler with different SNARKs and lookup arguments can lead to different novel results; e.g., combining a hash-based SNARK with our lookup arguments yields the *first transparent and post-quantum secure dynamic SNARK*.

1 Introduction

Several modern applications require verifying the correctness of stateful computations that entail accessing (via reads/writes) elements of potentially huge datasets. This abstraction broadly encompasses problems such as proving the correct execution of RAM programs accessing large memory space [61, 36, 49, 18], proving the validity of database query results [63, 83, 44, 57], or the correct inference of machine learning models [64, 24, 78]. This can be achieved via the use of *succinct non-interactive arguments of knowledge* (SNARKs), which enable a prover to generate a small and fast to verify proof about the validity of a statement for an NP relation. However, directly applying

a SNARK in these scenarios would require “loading” the entire dataset into the SNARK machinery (e.g., an arithmetic circuit or constraint system), making the prover complexity proportional to its size—and often prohibitively expensive for most real-world applications. While other alternative approaches exist in earlier literature [61, 68, 26]¹, the most promising solution is via the use of *lookup arguments* with sublinear prover times [5, 39], which in this work we refer to as *doubly-efficient*.²

At a high level, a lookup argument can be viewed as a specialized argument system that enables a prover to ensure the correctness of a batch of k accesses over a committed dataset represented as a table, with verification cost $o(k)$. The result of this is a new commitment on the accessed subset of the table that can be “readily” provided as input to a SNARK that can then argue about computations involving this accessed table.

Designing doubly-efficient lookup arguments has been a very active field of research, with the majority of existing schemes [38, 34, 82, 25, 36, 63, 76] relying (with some variations) on the classic elliptic-curve based KZG commitment scheme [51] and its multivariate extension PST [62]. This line of work has led to concretely very practical schemes; however, these suffer from some important drawbacks. First, relying on KZG/PST means that they require a one-time trusted setup to compute the public parameters, i.e., they are not *transparent*. Secondly, and most importantly, their security relies on elliptic-curve-based assumptions, making them completely insecure against adversaries with quantum capabilities [70]. This is further exacerbated by recent results (e.g., [21, 6]) suggesting that mandatory *post-quantum secure* cryptography may become reality earlier than expected.

Transparent and Post-Quantum Secure Doubly-Efficient Lookup Arguments. The only existing work that explicitly tries to build a post-quantum secure doubly-efficient lookup argument with transparent setup³ is by Campanelli et al. [23]. Their construction encodes the table $\mathbf{x}$ as a polynomial and relies on algorithms for fast polynomial evaluation [71, 14] that build a specialized data structure to speed up the evaluation at any point in time sub-linear in the polynomial degree. By deploying a Merkle tree over this data structure, the lookup argument of [23] achieves almost optimal lookup complexity. However, the very large hidden constants that stem from using existing fast polynomial evaluation algorithms make it rather impractical for use in practice (e.g., pre-processing a polynomial of 27 coefficients takes 30 hours and 60GB of RAM [28]).

Schemes from Error-Correction Codes. A different promising approach would be based on *error-correction codes*. Namely, if we consider a code $\mathcal{C}$ working over messages from $\mathbb{F}^n$, the prover can encode a lookup table $\mathbf{x} \in \mathbb{F}^n$ as a codeword $\mathbf{c}$. Then, given a lookup claim $(\mathcal{I} \in [n]^k, \mathbf{t} \in \mathbb{F}^k)$, where $\mathcal{I}$ contains the access indexes of and $\mathbf{t}$ the claimed lookup values from $\mathbf{x}$, it must convince a verifier (which only has oracle access to $\mathbf{c}$ and the encodings of $(\mathbf{x}, \mathcal{I})$) about relation:

$$\mathcal{R}_{\text{lkp}, \mathbf{c}, \delta} = \{([\mathbf{c}], [\mathcal{I}], [\mathbf{t}]); (\mathcal{I}, \mathbf{t}, \mathbf{x}) : \mathbf{t}[j] = \mathbf{x}[i_j] \text{ for all } i_j \in \mathcal{I}\}$$

where $\Delta(\mathbf{c}, \text{Enc}_{\mathcal{C}}(\mathbf{x})) < \delta$ with Δ defined as the relative Hamming distance.

For the resulting lookup argument to be doubly-efficient, the prover must be able to prove the above in time that is *sub-linear* in n , which affects the choice of $\mathcal{C}$. E.g., it is unclear how

¹Predominantly, using an *Authenticated Data Structure* (ADS) [56, 22, 29] to generate proofs showing the correctness of dataset accesses. While this can achieve prover overhead sublinear in the dataset size, it requires verifying ADS proofs inside the SNARK, which can impact performance significantly. Even worse, more practical instantiations rely on “SNARK-friendly” cryptographic primitives (e.g. [43, 3]) that are considerably less studied than conventional ones (e.g., BLAKE3 or SHA-3 [37]).

²Differentiating them from other widely used lookup arguments whose prover runs in time proportional to the entire underlying table size (e.g., [69, 39, 46]).

³To the best of our knowledge, no existing lattice-based lookup argument (e.g., [17]) satisfies these properties.

to achieve this for general codes (including Reed-Solomon [66]), since generically recovering an arbitrary entry of $\mathbf{x}$ from its codeword $\mathbf{c}$, one has to access $\mathcal{O}(n)$ elements of $\mathbf{c}$. An alternative would be to use (relaxed) Locally Decodable Codes (LDCs) or Correctable Codes (LCCs) [52, 54, 45, 12]. Unfortunately, this leads to verification complexity and proof size of $\Omega(k)$ since the verifier has to invoke the decoding algorithm for each single $(i_j, \mathbf{t}[j])$ [40, 16].

Using Projection Codes. A more refined approach is to use *projection codes*, proposed by Ishai et. al. [48], that allow the verifier to *batch* decode multiple elements at once. At a high level, with projection codes the prover can deterministically “project” $\mathcal{I}$ into a set of indexes $\mathcal{I}'$ over the codeword $\mathbf{c}$ such that $\mathbf{c}[\mathcal{I}']$ is an encoding of $\mathbf{t}$ with large distance. When coupled with a PCPP [11] (see [48], Section 5) they realise the only existing error-correction-code-based proving system for $\mathcal{R}_{\text{lkp}, \mathbf{c}}$ with $o(n)$ prover complexity and $o(k)$ proof size and verification.

While this seems very promising for our goal of building doubly-efficient lookup arguments, it does not solve our problem for two reasons. First, the authors of [48] only show how to construct projection codes from Reed-Muller codes that are only efficiently instantiated over binary or small-sized fields. This makes them incompatible with widely used SNARKs based on large-sized prime fields or field extensions that we aim for. Secondly, even if we could use it, the above construction is both concretely and theoretically impractical: Depending on the projection code used, it either incurs $\tilde{\mathcal{O}}(k^2)$ prover time with codeword size and encoding time of $\mathcal{O}(n^{1+a})$ for $a > 0$, or $\mathcal{O}(k^{1+a_c})$ prover time with $\mathcal{O}(n^c)$ (for $c \geq 2$) encoding time and codeword size.

1.1 Our Result: Doubly-Efficient Lookup Arguments from Tensor Codes

One main reason why the above approach suffers from these drawbacks is that the construction of [48] is based on a PCP, i.e., it is non-interactive. This necessitates the use of error-correction codes with strong “locality” properties, such as Reed-Muller, which leads to the above issues. In this work, we ask “*whether it is possible to achieve better results by allowing interaction between the parties*”.

Looking ahead, we answer this question positively by showing that when we allow for prover-verifier interaction, *tensor codes* support a proving system for $\mathcal{R}_{\text{lkp}}$ that is asymptotically and concretely more efficient than those from projection codes. Somewhat surprisingly, our lookup arguments achieve prover times comparable to (or even faster than) existing non-post-quantum-secure ones [63, 38]. Our main result shows that (here, informally):

Theorem 1 (Informal) *For a set of error-correction codes $\{\mathcal{C}_i\}_{i \in [d]}$, their tensor-product $\mathcal{C} = \mathcal{C}_1 \otimes \dots \otimes \mathcal{C}_d : \mathbb{F}^n \rightarrow \mathbb{F}^{n'}$, and security parameter λ , the following holds:*

- (1) *If d is constant and $\{\mathcal{C}_i\}_{i \in [d]}$ are linear-time encodable codes with constant distance and rate the IOP for $\mathcal{R}_{\text{lkp}}$ with $\mathcal{O}(\lambda k \sqrt[d]{n})$ prover time and $\mathcal{O}(n)$ codeword size and encoding time.*
- (2) *If $d = \frac{\log n}{a \log \log n}$, for a small constant $a \in \mathbb{N}$ and $\{\mathcal{C}_i\}_{i \in [d]}$ satisfying the singleton bound, then $\mathcal{C}$ has an IOP for $\mathcal{R}_{\text{lkp}}$ with $\mathcal{O}(\lambda k \log^a n)$ prover time and $\mathcal{O}(n^{1+1/a})$ codeword size and encoding time.*
- (3) *If $d = 2$ and $\mathcal{C}_2$ displays a proximity gap up to $\delta_{\mathcal{C}_2}/2$, then $\mathcal{C}$ has an IOP for $\mathcal{R}_{\text{lkp}}$ with $\mathcal{O}(\lambda \sqrt{nk})$ prover complexity and $\mathcal{O}(n \log n)$ codeword size and encoding complexity.*

Each of the above IOPs can be compiled into a lookup argument using the BCS transformation [10], resulting in constructions that are transparent and post-quantum secure. Our security analysis

Scheme	PreProc.	Prove	Verify & $ \pi $	Update	PQ-Sec.
CQ-based [38, 34, 82]	$\mathcal{O}(n \log n) \mathbb{G}$	$\mathcal{O}(k) \mathbb{G}$ $\mathcal{O}(k \log k) \mathbb{F}$	$\mathcal{O}(1) \mathbb{G}$	$\times$	$\times$
CFG [25]	$\mathcal{O}(n \log n) \mathbb{G}$	$\mathcal{O}(k) \mathbb{G}$	$\mathcal{O}(\log k) \mathbb{G}$	$\times$	$\times$
DGP ⁺ [36]	$\mathcal{O}(n \log n) \mathbb{G}$	$\tilde{\mathcal{O}}(\sqrt{kn}) \mathbb{G}$	$\mathcal{O}(1) \mathbb{G}$	$\tilde{\mathcal{O}}(\sqrt{kn}) \mathbb{G}$	$\times$
Rogue [63]	$\mathcal{O}(n \log n) \mathbb{G}$	$\mathcal{O}(k \log n) \mathbb{P}$	$\mathcal{O}(\log^2 n) \mathbb{P}$	$\mathcal{O}(k \log n) \mathbb{P}$	$\times$
CCC ⁺ [23]	$\tilde{\mathcal{O}}(n^{1+1/a}) \mathbb{F}$	$\mathcal{O}(k \cdot \text{polylog}(\mathbb{F} , n)) \mathbb{F}$	$\text{polylog}(k, n) \mathbb{F}$	$\times$	$\checkmark$
Proj. Codes [48, 49]	$\tilde{\mathcal{O}}(n^{1+1/a}) \mathbb{F}_{q(a)}$	$\tilde{\mathcal{O}}(k^2) \mathbb{F}_{q(a)}$	$\text{polylog}(k, n) \mathbb{F}_{q(a)}$	$\tilde{\mathcal{O}}(k(n^{1/a} + k)) \mathbb{F}_{q(a)}$	$\checkmark$
Sec 4.1 ($d \in \mathcal{O}(1)$)	$\mathcal{O}(\rho^{-d}n) \mathbb{F}$	$\tilde{\mathcal{O}}(\lambda k \sqrt[n]{n}) \mathbb{F}$	$\mathcal{O}(\log^3 n) \mathbb{F}$	$\mathcal{O}(k \log n) \mathbb{F}$	$\checkmark$
Sec 4.1 ($d = \frac{\log n}{a \log \log n}$)	$\tilde{\mathcal{O}}(n^{1+1/a}) \mathbb{F}$	$\tilde{\mathcal{O}}(\lambda k \log^{1+a} n) \mathbb{F}$	$\mathcal{O}(\log^4 n) \mathbb{F}$	$\tilde{\mathcal{O}}(k \sqrt[n]{n}) \mathbb{F}$	$\checkmark$
TENSEUP Sec 4.2	$\mathcal{O}(n \log^2 n) \mathbb{F}$	$\mathcal{O}(\lambda \sqrt{kn}) \mathbb{F}$	$\mathcal{O}(\log^3 n) \mathbb{F}$	$\mathcal{O}(k \log^3 n) \mathbb{F}$	$\checkmark$

Table 1: Performance comparison of existing doubly-efficient lookup arguments, for table size n and k lookups/updates. $\mathbb{F}, \mathbb{G}$ and $\mathbb{P}$ correspond to field, group and pairing operations. Projection codes only work for a field of size $q(a) \in \mathcal{O}(\log^{1+a} n)$. We denote with d the degree of the tensor code. For the construction of Section 4.1 when $d \in \mathcal{O}(1)$, $\rho \in \mathcal{O}(1)$ corresponds to the rate of the underlying codes used to form the tensor code.

only relies on the random oracle model. We provide additional details in our technical overview (Section 2).

Table 1 (bottom three rows correspond to the lookup arguments from (1)-(3) above) summarizes our constructions’ asymptotic performance and their comparison with prior solutions for doubly-efficient lookup arguments. For instance, compared to projection code solutions [48, 49], (2) achieves improved prover performance for large batch access sizes k and can be instantiated over any field. Compared to [23], we achieve slightly better asymptotics and with smaller hidden constants (as we rely on tensor codes and not fast polynomial evaluation algorithms). Another interesting observation is that (1) is the only existing scheme with *optimal* pre-processing time.

Finally, our third construction is a *concretely* practical post-quantum doubly-efficient lookup argument. We call this scheme TENSEUP, and we implemented and experimentally evaluated its performance in Section 7. For instance, when running on a single thread, for a table of size 2^{22} , TENSEUP takes 0.3-0.8sec for lookup sizes varying between 128-1024, whereas these times are 1.4-4sec for table size 2^{26} . For the same table sizes, preprocessing takes only $< 1\text{min}$ and 20min , respectively which can be further improved by parallelization. These numbers are not only very encouraging for real-world SNARK applications, but, somewhat surprisingly, for many instances TENSEUP is actually **faster** than an optimized implementation of CQ [75]—which is not post-quantum secure.

Efficiently Supporting Write Accesses. Many of our original motivating applications inherently rely on maintaining an updatable state, e.g., for RAM writes or relational database insertions. When using a lookup argument to prove dataset accesses, this translates into a need for efficient support, not only for lookups, but also for updates to the stored table. Most existing lookup arguments are *static*, i.e., a single change in the table necessitates the need to re-run the “heavy” pre-processing

phase (e.g., to recompute individual membership proofs in KZG-based schemes [38, 34]), which makes them inapplicable for these scenarios. The only prior works that introduce doubly-efficient *updatable* lookup arguments are [36, 63], neither of which is post-quantum secure, and the implicit construction of [49] from projection codes that, as we discussed above, operates in restricted choices of fields.

We propose a generic transformation that turns any static lookup argument into an updatable one, adapting amortization techniques from data structures [13], and apply it to our schemes (Table 1). Interestingly, even when applying it to lookup arguments based on univariate KZG, our transformation leads to an (amortized) lookup and update complexity of $\mathcal{O}(k \log^2 n)$, significantly improving upon the $\mathcal{O}(\sqrt{kn})$ complexity of [36]. Due to space limitations, we provide all details about our transformation in Section 5.

A General Compiler from Static to Dynamic SNARKs. In recent work [73], Wang et al., proposed the notion of *dynamic SNARKs*. Roughly speaking, a SNARK for relation $\mathcal{R}$ is dynamic if, having computed proof π for a pair $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$, it supports an *Update* functionality that can compute a new proof π' for an updated pair $(\mathbf{x}', \mathbf{w}') \in \mathcal{R}$, in time *proportional only to the difference* of $(\mathbf{x}, \mathbf{w})$ and $(\mathbf{x}', \mathbf{w}')$ (based on a distance metric, e.g., Hamming distance) and, hopefully, faster than computing a new proof “from scratch”. This feature may be useful in practical applications with frequent and somewhat “controlled” updates, such as zkRollups and verifiable ML inference or building primitives like sparse SNARKs [73], and has motivated subsequent works [81, 72, 7].

Existing works are restricted to specialized dynamic versions of PlonK [73, 7] or Groth16 [81, 72], which leaves open the question of building dynamic versions of other widely used SNARKs. In this work (Section 6), we propose a general compiler from any static SNARK to its dynamic version, relying on doubly-efficient updatable lookup arguments to store and efficiently access witness updates. The benefits of this are twofold. First, we reduce the hard problem of building dynamic SNARKs to the (seemingly) easier one of building doubly-efficient lookup arguments. Second, it allows for a multitude of potential dynamic SNARK schemes with different security and efficiency guarantees, including future improvements in the literature. As a concrete example, applying our compiler to a static SNARK like [74, 19] and combined with one of our doubly-efficient updatable lookup arguments, yields the *first construction of a transparent and post-quantum secure dynamic SNARK* (whose security, moreover, solely relies on the random oracle model). Additionally, combining any optimal time prover static SNARK with a Merkle Tree (viewed as a “trivial” updatable lookup argument), gives the *first dynamic SNARK with (amortized) $\mathcal{O}(k \log^2 n)$ updates* (where k is the distance between the instance/witness pairs and n is the witness size), improving upon the prior state of the art [73] by a $\log n$ factor.

2 Technical Overview

1) High-Level Idea. Recall that our initial goal is to identify an appropriate error-correction code that will allow us to prove $\mathcal{R}_{\text{lkp}}$ efficiently in an interactive manner. With that in mind, our starting point is any error-correction code $\mathcal{C}_2$, possibly without any locality property. Instead of using it to directly encode the table $\mathbf{x} \in \mathbb{F}^n$ (which would eventually make the prover complexity of $\mathcal{R}_{\text{lkp}}$ proportional to n), we first parse $\mathbf{x}$ as a matrix $\mathbf{X} \in \mathbb{F}^{\sqrt{n} \times \sqrt{n}}$ and then encode each row using $\mathcal{C}_2 : \mathbb{F}^{\sqrt{n}} \rightarrow \mathbb{F}^{n_2}$ thus computing the interleaved code $\mathbf{X}' \in \mathbb{F}^{\sqrt{n} \times n_2}$. Next, in order to prove $\mathcal{R}_{\text{lkp}, \mathbf{X}'}$, both parties will first partition $(\mathcal{I}, \mathbf{t})$ to k' pairs $\{(\mathcal{I}_i, \mathbf{t}_i)\}_{i \in [k']}$ so that all indexes of $\mathcal{I}_i$ lie on the same row of $\mathbf{X}$. Then, for each such partition $(\mathcal{I}_i, \mathbf{t}_i)$, the prover interacts with the verifier to show

its correctness. For given choices of $\mathcal{C}_2$, this can either be proven directly using specialized lookup protocols (with linear prover time with respect to the table size) such as Lasso or Plookup [69, 39] or, more generally (as we will show below), using code-switching [67].

Observe that if all indexes in $\mathcal{I}$ lie in the same row (i.e., when $k' = 1$), the above scheme achieves (almost) excellent asymptotics: It has $\mathcal{O}(\sqrt{n})$ prover complexity and $\mathcal{O}(\text{polylog}(k, n))$ proof size and verification time (assuming the existence of IOPPs with poly-logarithmic verifier complexity and omitting the verification costs from directly accessing $\mathbf{t}, \mathcal{I}$ for simplicity). Unfortunately, this is just a “best-case” result. In practice, in the worst case, each index of $\mathcal{I}$ can lie on a different row of $\mathbf{X}$ (i.e., $k' = k$). In that case, the prover time would be $\mathcal{O}(k\sqrt{n})$ but the proof size and verification time would blow up to $\mathcal{O}(k \cdot \text{polylog}(n))$.

The observation that the above simple approach gives a good solution when all indexes $\mathcal{I}$ happen to lie in the same row of $\mathbf{X}$ leads to the main idea behind our construction: Instead of working directly over interleaved codes and hoping for “good” queries, we modify the codes in such a way so that we will be able to reduce the claim about the correctness of $(\mathcal{I}, \mathbf{t})$ (over arbitrary indexes), into a claim about the correctness of another index/value pair $(\mathcal{I}', \mathbf{t}')$ where all indexes in $\mathcal{I}'$ lie on the same row of $\mathbf{X}'$ and thus their correctness can be proven efficiently using the above approach.

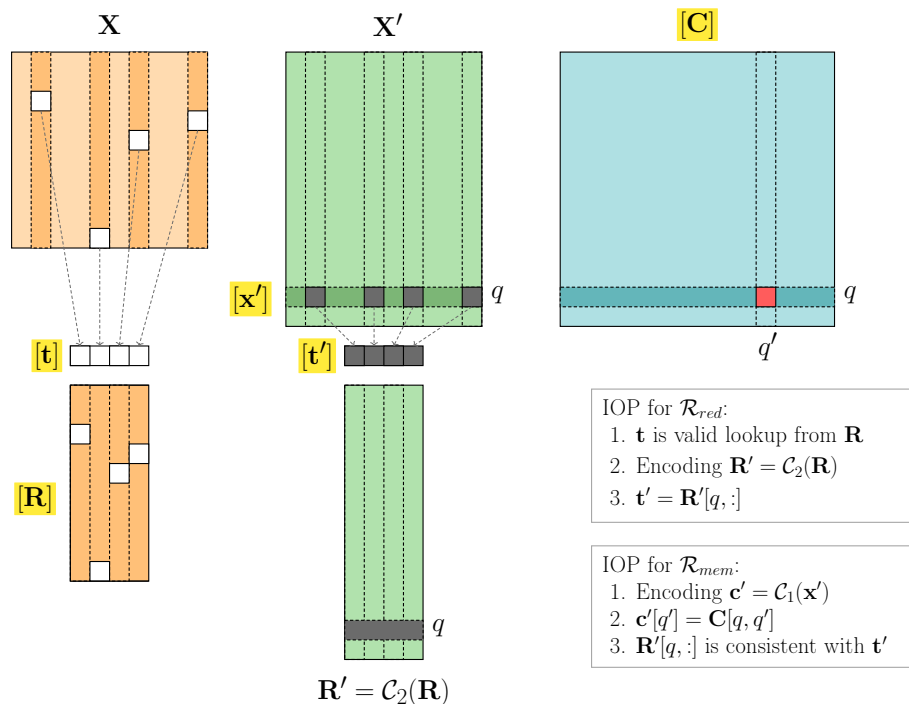


Figure 1: Overview of our IOP from tensor codes.

We achieve this by encoding $\mathbf{X}$ using a tensor code instead of just an interleaved code. Namely, we first encode each column of $\mathbf{X}$ using any error-correction code $\mathcal{C}_1 : \mathbb{F}^{\sqrt{n}} \rightarrow \mathbb{F}^{n_1}$ thus computing a different interleaved code $\mathbf{X}' \in \mathbb{F}^{n_1 \times \sqrt{n}}$, and then encode each row of $\mathbf{X}'$ using $\mathcal{C}_2 : \mathbb{F}^{\sqrt{n}} \rightarrow \mathbb{F}^{n_2}$ thus computing $\mathbf{C}$ (see Figure 1 above for a pictorial description of the original table $\mathbf{X}$, its interleaved encoding $\mathbf{X}'$ and the final codeword $\mathbf{C}$). Then, to prove $\mathcal{R}_{\text{lkp}, \mathbf{C}}$, we partition our protocol into two

phases: A *reduction phase*, where both parties interact to reduce a claim about the correctness of $(\mathcal{I}, \mathbf{t})$ to $(\mathcal{I}', \mathbf{t}')$ where all indexes of $\mathcal{I}' \in [\sqrt{n}]^{k'}$ now lie on the same row of the interleaved code $\mathbf{X}'$. Crucially, this should be done in a way such that if $(\mathcal{I}, \mathbf{t})$ is incorrect, then with overwhelming probability, $(\mathcal{I}', \mathbf{t}')$ will also be incorrect. At the end of this phase, the verifier will end up with the oracles of the encodings of the reduced indexes $\mathcal{I}'$ and lookup values $\mathbf{t}'$. In the second phase, which we call *lookup phase*, both parties interact in order to ensure the correctness of $(\mathcal{I}', \mathbf{t}')$. In more detail:

Reduction Phase. First, the prover decomposes each index $i_j \in \mathcal{I}$ into row and column indexes $(i_j^{(1)}, i_j^{(2)})$ of $\mathbf{X}$ and initializes sets $\mathcal{I}^{(1)}$ and $\mathcal{I}^{(2)}$ consisting of all (unique) accessed row and column indexes, respectively. It forms matrix $\mathbf{R} \in \mathbb{F}^{\sqrt{n} \times |\mathcal{I}^{(2)}|}$ where $\mathbf{R}[:, j]$ is claimed to be the i_j -th column of $\mathbf{X}$, for all $i_j \in \mathcal{I}^{(2)}$ (see bottom left table of Figure 1).

At this point, observe that if the prover provided $\mathbf{R}$ and $\mathcal{I}$ to the verifier, the latter could directly perform a reduction as follows: First, it checks if every element in $\mathbf{t}$ is correctly included in $\mathbf{R}$. Then, it encodes each column of $\mathbf{R}$ using $\mathcal{C}_1$ (thus computing $\mathbf{R}' \in \mathbb{F}^{n_1 \times |\mathcal{I}^{(2)}|}$ corresponding to the bottom green table of Figure 1) and outputs the reduced claim $(\mathbf{t}', \mathcal{I}')$ where $\mathbf{t}'$ is a randomly sampled row of $\mathbf{R}'$ and $\mathcal{I}'$ corresponds to $\mathcal{I}^{(2)}$ —the unique column indexes of $\mathcal{I}$ (the dark green row of Figure 1). Note that if $(\mathcal{I}, \mathbf{t})$ and $\mathbf{R}$ are correct, then $(\mathcal{I}', \mathbf{t}')$ corresponds to the restriction of the q -th row of $\mathbf{X}'$ at $\mathcal{I}'$. In contrast, if $(\mathcal{I}, \mathbf{t})$ is incorrect, then the prover either can send the correct $\mathbf{R}$ (which would make the first check of the verifier fail), or send an incorrect $\mathbf{R}$. In the second case, there would exist a column $\mathbf{R}[:, j]$ inconsistent with the corresponding column in $\mathbf{X}[:, i_j]$ and thus their encodings $(\mathbf{R}'[:, j]$ and $\mathbf{X}'[:, i_j])$ will differ in at least δ_1 elements where δ_1 is the distance of $\mathcal{C}_1$. Consequently, the probability the verifier samples an incorrect $(\mathbf{t}', \mathcal{I}^{(2)})$ corresponds to the probability that $\mathbf{R}[q, j] \neq \mathbf{X}'[q, i_j]$ which is at least δ_1 .

Instead of relying on the verifier to perform these checks, to reduce verification complexity, the prover sends an oracle of the encodings of $\mathbf{R}$, $\mathcal{I}^{(1)}$, and $\mathcal{I}^{(2)}$ to the verifier, who then samples a random row index $q \in [n_1]$ of $\mathbf{X}'$ and delegates all of its work to the prover by running an IOP for the relation:

$$\begin{aligned} \mathcal{R}_{\text{Red}} = \{ & ([\mathbf{t}], [\mathbf{t}'], [\mathbf{R}], [\mathcal{I}], [\mathcal{I}^{(1)}], [\mathcal{I}^{(2)}], q); (\mathbf{t}, \mathbf{t}', \mathbf{R}, \mathcal{I}, \mathcal{I}^{(1)}, \mathcal{I}^{(2)}) : \\ & \mathbf{R}[i_j^{(1)}, i_j^{(2)}] = \mathbf{t}[j], \forall i_j \in \mathcal{I} \wedge \mathbf{t}' = \mathbf{R}'[q, :] \wedge \mathcal{I}^{(2)}, \mathcal{I}^{(1)} \text{ are honestly computed} \} \end{aligned}$$

Lookup Phase. In this phase, the prover shows that $\mathbf{X}'[q : i_j] = \mathbf{t}'[j]$ for all $i_j \in \mathcal{I}'$. For general codes $\mathcal{C}_2$, this can be achieved by utilizing code-switching techniques. Namely, the prover commits to $\mathbf{x}' = \mathbf{X}'[q, :]$ by sending to the verifier the oracle of its encoding (via another “IOP-friendly” error-correction code $\mathcal{C}'$) to the verifier. The verifier samples and sends a random index $q' \in [n_2]$ and the prover replies with $r \in \mathbb{F}$ claimed to be $\text{Enc}_{\mathcal{C}_2}(\mathbf{X}'[q, :])[q']$. Finally, the verifier queries the oracle of $\mathbf{C}$ to check if $\mathbf{C}[q, q'] = r$ and interacts with the prover following an IOP showing the satisfiability of:

$$\begin{aligned} \mathcal{R}_{\text{Mem}} = \{ & ([\mathbf{x}'], [\mathcal{I}^{(2)}], [\mathbf{t}'], q', r); (\mathbf{x}', \mathcal{I}^{(2)}, \mathbf{t}) : \\ & r = \text{Enc}_{\mathcal{C}_2}(\mathbf{x}')[q'] \wedge \mathbf{x}'[i_j] = \mathbf{t}[j], \forall i_j \in \mathcal{I}^{(2)} \} \end{aligned}$$

Assuming that the verifier has oracle access to an honestly computed codeword $\mathbf{C}$, then, it is not hard to see that the verifier rejects with probability $\approx \delta_1 \delta_2$ (where δ_2 is the distance of $\mathcal{C}_2$). That is, if $(\mathcal{I}, \mathbf{t})$ was incorrect, then $(\mathcal{I}', \mathbf{t}')$ would be incorrect with probability $\approx \delta_1$. Conditioned

on this, in this phase the prover must either commit to $\mathbf{X}'[q, :]$ which would not satisfy $\mathcal{R}_{\text{Mem}}$ and thus the verifier will reject with overwhelming probability, or commit to a wrong row and hope that the verification passes the point query test. The probability of this happening is only $1 - \delta_2$ (where δ_2 is the distance of $\mathcal{C}_2$). Hence, the overall rejection probability is $\delta_1 \delta_2$. In Section 4.1 we show how to appropriately apply repetition of the protocol l times, in order to obtain negligible round-by-round (knowledge) soundness and extend our construction to handle the case where the correctness of $\mathbf{C}$ can not be guaranteed.

2) Instantiating Our Lookup Argument. Observe that the above protocol achieves poly-logarithmic verification complexity and prover time of $\mathcal{O}(lk\sqrt{n}) = \mathcal{O}(\lambda k\sqrt{n})$ where l is the number of repetitions required to achieve negligible soundness error. Although this is already sublinear in n , we can further improve our result. We propose two different constructions that rely on the above building blocks, corresponding to the different complexities from Theorem 1.

For our first construction (Section 4.1), instead of encoding $\mathbf{x}$ using a tensor-product of two codes, we utilize a code $\mathcal{C}^{(d)}$ recursively defined as $\mathcal{C}^{(d)} : \mathcal{C}_d \otimes \mathcal{C}^{(d-1)}$, where $\mathcal{C}^{(i)} : \mathcal{C}_i \otimes \mathcal{C}^{(i-1)}$ and $\mathcal{C}^{(2)} : \mathcal{C}_2 \otimes \mathcal{C}_1$ for $\mathcal{C}_i : \mathbb{F}^{n_i} \times \mathbb{F}^{n'_i}$. Then, to prove $\mathcal{R}_{\text{lkp}, \mathbf{C}}$, we follow the reduction phase to reduce the lookup claim $(\mathcal{I}, \mathbf{t})$ over $\mathbf{X} \in \mathbb{F}^{n_1 \times (n_2 \cdots n_d)}$ into a lookup claim $(\mathcal{I}', \mathbf{t}')$ over a single row of the interleaved code $\mathbf{X}' \in \mathbb{F}^{n'_1 \times (n_2 \cdots n_d)}$. At this point, one could apply the lookup phase over that row, but this would result in $\mathcal{O}(n/n_1)$ prover time. Instead, we observe that this row is a lookup table encoded via $\mathcal{C}^{(d-1)}$ —which is a tensor product of $\mathcal{C}_{d-1}$ and $\mathcal{C}^{(d-2)}$. Hence, both parties can simply further reduce the lookup claim $(\mathcal{I}', \mathbf{t}')$ into another one via another invocation of the reduction phase. This continues until they end up with a claim over $\mathcal{C}^{(2)}$ which can be proven as explained above. For appropriately chosen parameters, this recursive construction achieves the first two claims of Theorem 1, depending on the chosen code.

In our second construction (see Section 4.2), we adapt our original approach to achieve a *concretely efficient* IOP for $\mathcal{R}_{\text{lkp}}$ with $\mathcal{O}(\sqrt{kn})$ prover time. Our starting point is the observation that if the number of lookups k was fixed, then one could obtain this complexity by simply setting the dimension of the matrix $\mathbf{X}$ to be $\sqrt{n/k} \times \sqrt{nk}$. Unfortunately, for most practical applications k would usually not be known and fixed ahead of time. To compensate for this, we can simply compute a different lookup table for each possible value of $k = 2^i$ for $i = 0, \dots, \log n - 1$ (assuming appropriate padding for $\mathcal{I}, \mathbf{t}$) leading to $\log n$ different codes. In this manner, we achieve $\mathcal{O}(\lambda\sqrt{kn})$ prover time for each k . Concretely, however, this approach is still slow since the prover has to produce IOP proofs of witness size proportional to $\lambda\sqrt{kn}$, mainly because the prover has to repeat the lookup phase for the l different reductions. We show how to reduce this complexity so that the prover needs to invoke the IOP only on a witness of size $\mathcal{O}(\sqrt{kn})$, while performing strictly $l\sqrt{kn}$ additional field operations outside the IOP, by aggregating the reduced claims and producing a single lookup proof over their aggregation.

Finally, we prove that both IOPs are round-by-round (knowledge) sound and instantiate corresponding lookup arguments via the BCS transformation [10].

3) Static-To-Updatable Lookup Arguments, Generically. Our transformation is inspired by the “classic” static-to-dynamic data structures compiler of [13], adapted to our setting. This technique has previously found numerous applications in designing efficiently updatable cryptographic primitives, e.g., Oblivious RAM [41], searchable encryption [35], and updatable LDCs and projection codes [30, 48, 49].

In particular, the prover actively maintains a data structure $\mathcal{D}_{\mathcal{P}}$ consisting of $\log n$ entries, where the i -th entry holds two lookup tables of size 2^i each, the first storing the updates and the second storing their corresponding indexes (sorted), except from the $\log n$ -th entry, which maintains

the original lookup table. Crucially, $\mathcal{D}_{\mathcal{P}}$ is sorted in chronological order, that is, for any $i_1 < i_2$, updates stored in $\mathcal{D}_{\mathcal{P}}[i_1]$ are “more recent” than those of $\mathcal{D}_{\mathcal{P}}[i_2]$. This enables the prover to “easily” prove the correctness of a lookup pair $(\mathcal{I}, \mathbf{t})$ by sequentially traversing all entries of $\mathcal{D}_{\mathcal{P}}$. For the i -th entry, it first removes all lookup pairs from $(\mathcal{I}, \mathbf{t})$ that have indexes in $\mathcal{D}_{\mathcal{P}}[i]$ and then shows that the remaining elements do not “belong” in $\mathcal{D}_{\mathcal{P}}[i]$. The latter can be efficiently proven since all values of $\mathcal{D}_{\mathcal{P}}[i]$ are sorted with respect to the indexes. As for the update algorithm, provided an update pair $(i \in [n], v \in \mathbb{F})$, our prover finds the first entry $j \in [\log n]$ such that $\mathcal{D}_{\mathcal{P}}[j] = \perp$. It then merges (i, v) and $\mathcal{D}_{\mathcal{P}}[1], \dots, \mathcal{D}_{\mathcal{P}}[j-1]$ into a new table and generates a SNARK proof showing that this new table was computed correctly. Finally, it stores it in $\mathcal{D}_{\mathcal{P}}[j]$ and clears all smaller tables. In Appendix 5 we give a detailed description of our transformation and also show how to support batch updates of a vector of update pairs $\{(i_j, v_j)\}_{j \in [k]}$ of arbitrary size k , without requiring the vector to be decomposed into individual updates. Looking ahead, the ability to handle arbitrary batches of updates is essential when presenting our dynamic SNARK compiler below.

4) From Doubly Efficient Updatable Lookup Arguments to Dynamic SNARKs. Armed with a doubly efficient updatable lookup argument, we now show how to use it to build a dynamic SNARK from any static one, instead of trying to adapt a specific proving system in order to make it dynamic. Assuming an underlying static SNARK, we make use of the following natural idea. First, we let the prover store the entire witness using an updatable lookup argument. Then, each time the instance and witness are updated, it simply generates a proof (possibly using a different SNARK) showing the correct computation of an *update* algorithm that not only updates the old witness to the new one (based on the update request) but also certifies its validity. If the updatable lookup argument used is doubly efficient, and there exists such an efficient update algorithm running in time sublinear in the witness for the given NP relation, the overall prover’s work would become smaller than computing a fresh proof from scratch.

To materialize the above idea, we choose the circuit satisfiability relation $\mathcal{R}_C = \{(\mathbf{x}; \mathbf{w}) : C(\mathbf{x}, \mathbf{w}) = 1\}$ for the underlying static SNARK, as it encompasses an update algorithm that satisfies the requirements described above. In particular, let $\mathbf{T}_C$ be the “extended” witness of $\mathbf{w}$, namely, a table consisting of $\mathbf{w}$ and all intermediate evaluations of $C(\mathbf{x}, \mathbf{w})$ as well as information about the circuit C . Then, there exists a “natural” algorithm (Lemma 3) that is given the pair $(\mathbf{x}, \mathbf{w})$, the table $\mathbf{T}_C$, as well as a list of changes over $(\mathbf{x}, \mathbf{w})$ and updates $\mathbf{T}_C$ to a new table $\mathbf{T}'_C$ that is consistent with the new instance/witness pair $(\mathbf{x}', \mathbf{w}')$ formed after applying the desired changes in time $\mathcal{O}(\mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C))$, where $\mathcal{D}$ is the distance metric defined as the number of wires the values of which have been changed. We then construct a SNARK that proves the correct computation of this update algorithm in time proportional to $\mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C)$ (i.e., sub-linear in $|C|$). This is achieved via standard techniques to emulate the algorithm as a RAM program [36, 49, 18], and proving its read/write accesses over $\mathbf{T}_C$ using the underlying doubly-efficient updatable lookup argument.

We now have everything we need to construct our dynamic SNARK for $\mathcal{R}_C$, based on our initial idea. Assuming an initial/witness pair $(\mathbf{x}, \mathbf{w})$, one way to achieve this would be to simply produce a proof for the update algorithm for each new update $(\mathbf{x}', \mathbf{w}')$. Unfortunately, by doing so, updates “accumulate” over time. Eventually $\mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C)$ becomes $|C|$, thus proving updates becomes as costly as producing a proof for the static SNARK. Alternatively, one could incrementally prove updates, storing all intermediate proofs, and proving each new update with respect to the most recent one. However, the following subtle issue arises: For the verifier to be convinced about the validity of a new $(\mathbf{x}^{(i+1)}, \mathbf{w}^{(i+1)})$, it must have validated each intermediate update by verifying its update proof, as well as verifying the original SNARK proof for $(\mathbf{x}, \mathbf{w})$. That is, in the worst case, for a verifier that has not “seen” prior updates the proof size and verification time are proportional

to the total number of updates. Instead, in our compiler (Section 6) we propose a way to *consolidate* intermediate update proofs by merging multiple update requests and recomputing a single proof for the update algorithm. Thus, at any given point, the prover stores only $\log |C|$ update proofs.

3 Preliminaries

Notation. Let $[n] = \{1, \dots, n\}$ and let $\mathbb{F}$ be a prime field. We represent vectors by bold lowercase letters and matrices by bold uppercase letters. For a vector $\mathbf{x}$, $\mathbf{x}[i]$ denotes its i -th entry; for a matrix $\mathbf{X}$, $\mathbf{X}[i, :]$ and $\mathbf{X}[:, j]$ denote its i -th row and j -th column, respectively. We denote the vector of pairs $((i_1, \mathbf{t}[1]), \dots, (i_k, \mathbf{t}[k]))$ as $(\mathcal{I} \in [n]^k, \mathbf{t} \in \mathbb{F}^k)$ where $\mathcal{I} = (i_1, \dots, i_k)$. We let λ be the security parameter and denote with $\text{negl}(\lambda)$ the function such that $\text{negl}(\lambda) < \frac{1}{\text{poly}(\lambda)}$ for every positive polynomial poly . Next, we present the main building blocks used to construct our schemes.

3.1 Error-Correction Codes

An error-correction code of codeword length n' and message length $n \leq n'$ over a field $\mathbb{F}$ is an n -dimensional subspace $\mathcal{C} \subseteq \mathbb{F}^{n'}$. The encoding of a message $\mathbf{x} \in \mathbb{F}^n$ is denoted by $\mathbf{c} \leftarrow \text{Enc}_{\mathcal{C}}(\mathbf{x})$. The *rate* of the code is $\rho = n/n'$. The *distance* of the code $\mathcal{C}$ is the minimum Hamming distance between any two codewords in $\mathcal{C}$. When $\mathcal{C}$ is a linear code, the minimum distance is equal to the minimum Hamming weight of any nonzero codeword $\min_{\mathbf{u} \in \mathcal{C}, \mathbf{u} \neq \mathbf{0}} \text{wt}(\mathbf{u})$. We let $\delta = \min_{\mathbf{u} \in \mathcal{C}, \mathbf{u} \neq \mathbf{0}} \text{wt}(\mathbf{u})/n'$ denote the relative distance of $\mathcal{C}$. We say that a family of code is asymptotically good if ρ and δ remain constants as $n \rightarrow \infty$.

Tensor Codes. A tensor code of any two codes $\mathcal{C}_1 : \mathbb{F}^{n_1} \rightarrow \mathbb{F}^{n'_1}, \mathcal{C}_2 : \mathbb{F}^{n_2} \rightarrow \mathbb{F}^{n'_2}$, is a code $\mathcal{C}_1 \otimes \mathcal{C}_2 : \mathbb{F}^{n_1 \times n_2} \rightarrow \mathbb{F}^{n'_1 \times n'_2}$ with an encoding algorithm that works as follows: for a message $\mathbf{x} \in \mathbb{F}^{n_1 \times n_2}$, $\mathbf{x}$ is first parsed as a matrix $\mathbf{X} \in \mathbb{F}^{n_1 \times n_2}$ and each column is encoded with $\mathcal{C}_1$ to obtain the intermediate matrix $\mathbf{X}' \in \mathbb{F}^{n'_1 \times n_2}$. Next, each row of $\mathbf{X}'$ is encoded with $\mathcal{C}_2$ thus computing the codeword $\mathbf{C} \in \mathbb{F}^{n'_1 \times n'_2}$. In a similar fashion, one can obtain a d -dimensional tensor-codeword using d error-correction codes $\mathcal{C}_1, \dots, \mathcal{C}_d$ over $\mathbb{F}$ by parsing the message $\mathbf{x}$ of length $n_1 \dots n_d$ accordingly. If $\mathcal{C}_i$ has relative distance δ_i , then $\mathcal{C}_1 \otimes \dots \otimes \mathcal{C}_d$ has relative distance $\prod_{i=1}^d \delta_i$. If each $\mathcal{C}_i$ is linear with rate ρ_i , then the tensor code is linear and has rate $\prod_{i=1}^d \rho_i$.

3.2 Interactive Oracle Proofs (Of Proximity)

In this section, we present our main building blocks used in order to construct our lookup argument.

Interactive Oracle Proofs. A public-coin *Interactive Oracle Proof* (IOP) [9, 10] is a k -round interactive protocol between a prover and verifier where the prover that holds an instance/witness pair $(\mathbf{x}, \mathbf{w})$ of an NP relation $\mathcal{R}$ tries to convince the verifier who only holds $\mathbf{x}$ that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$. This is done by letting the prover in each round $i \in [k]$ to send an oracle of a string $\pi^{(i)}$ to the verifier which responds with a fresh public random challenge r_i . After the interaction, the verifier makes at most q non-adaptive queries to the prover's oracle messages and decides whether to accept.

We say that an IOP is (perfectly) complete if and if for every $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$, the following holds:

$$\Pr[\langle \mathcal{P}(\mathbf{x}, \mathbf{w}), \mathcal{V}(\mathbf{x}) \rangle = 1] = 1,$$

where $\langle \mathcal{P}(x, w), \mathcal{V}(x) \rangle$ the output of the verifier after the completion of the protocol. Next, we say that an IOP is knowledge sound, if there exists a PPT extractor $\mathcal{E}$ such that, for any $\mathbf{x}$ and any

prover $\mathcal{P}$ for which $\mathcal{E}$ has rewinding access, the following holds:

$$\Pr[(\mathbf{x}, \mathbf{w} \leftarrow \mathcal{E}^{\mathcal{P}}(\mathbf{x})) \notin \mathcal{R} \wedge |\langle \mathcal{P}(\mathbf{x}, \mathbf{w}), \mathcal{V}(\mathbf{x}) \rangle = 1] < \text{negl}(\lambda).$$

Round-by-Round (Knowledge) Soundness. Round-by-round (knowledge) soundness [27, 32, 47] is a strengthening of standard (knowledge) soundness that enables us to prove (knowledge) soundness of an IOP when making it non-interactive using the BCS transformation [10]. In this work, we use the notions of round-by-round (knowledge) soundness proposed in [32, 47]. More formally:

Definition 1 (Round-by-Round Soundness) *We say that a k -round IOP(P) for an NP relation $\mathcal{R}$ is round-by-round sound with error ϵ_{rbr} if there exists a doomed set $\mathcal{D}$ of partial and complete transcripts (where a partial transcript is a suffix of $([\pi^{(1)}], r_1, \dots, [\pi^{(k)}], r_k)$ which is a complete transcript) such that:*

1. *If $\mathbf{x} \notin \mathcal{L}_{\mathcal{R}}$, then $(\mathbf{x}, \emptyset) \in \mathcal{D}$.*
2. *If $(\mathbf{x}, \mathbf{tr}) \in \mathcal{D}$ and the transcript $\mathbf{tr}$ is complete, the verifier rejects.*
3. *For every $i \in [k]$ and partial transcript $\mathbf{tr}_i$, such that $(\mathbf{x}, \mathbf{tr}_i) \in \mathcal{D}$, then it must hold that*

$$\Pr_{r_{i+1} \xleftarrow{\$} \mathbb{F}} [(\mathbf{tr}_i, [\pi^{(i+1)}], r_{i+1}) \notin \mathcal{D}] \leq \epsilon_{rbr}.$$

Definition 2 (Round-by-Round Knowledge Soundness) *We say that a k -round IOP(P) for an NP relation $\mathcal{R}$ is round-by-round knowledge sound with error κ_{rbr} if there exists a PPT extractor $\mathcal{E}$ and a doomed set $\mathcal{D}$ such that:*

1. *$(\mathbf{x}, \emptyset) \in \mathcal{D}$ for all possible inputs.*
2. *If for any input $\mathbf{x}$ and complete transcript $\mathbf{tr}$, holds that $(\mathbf{x}, \mathbf{t}) \in \mathcal{D}$ then, the verifier rejects.*
3. *For any $i \in [k]$ and partial transcript $\mathbf{tr}_i$, such that $(\mathbf{x}, \mathbf{t}_i) \in \mathcal{D}$, then, for any $\pi^{(i+1)}$ it holds that if:*

$$\Pr_{r_{i+1} \xleftarrow{\$} \mathbb{F}} [(\mathbf{t}_i, [\pi^{(i+1)}], r_{i+1}) \notin \mathcal{D}] > \kappa_{rbr}.$$
Then, the extractor outputs a witness $\mathbf{w}$ such that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$.

Provided an IOP with negligible round-by-round (knowledge) soundness errors, then someone can show that the derived non-interactive protocol has a negligible (knowledge) soundness (even against quantum adversaries) by utilizing the following theorem:

Theorem 2 (BCS Transformation [10, 32, 33]) *For any IOP with round-by-round (knowledge) soundness error of $\epsilon_{rbr}, \kappa_{rbr}$, any relation $\mathcal{R}$, random oracle $H : \{0, 1\}^* \rightarrow \{0, 1\}^k$ and query bound $Q \in \mathbb{F}$ the BCS transformation produces a non-interactive proof system for $\mathcal{R}$ with:*

- *Adaptive soundness error $\epsilon_{BCS} \leq Q\epsilon_{rbr} + 3(Q^2 + 1)/2^k$ and adaptive knowledge soundness error of $\kappa_{BCS} \leq Q\kappa_{rbr} + 3(Q^2 + 1)/2^k$.*
- *Adaptive soundness error $\epsilon_{qBCS} = \Theta(Q\epsilon_{BCS})$ and adaptive knowledge soundness error of $\kappa_{qBCS} = \Theta(Q\kappa_{BCS})$ for Q -query quantum adversaries.*

3.3 (Dynamic) SNARKs

A succinct non-interactive argument of knowledge (SNARK) [53] for an NP relation $\mathcal{R}$ is a cryptographic protocol that enables a prover who holds an instance/witness pair $(\mathbf{x}, \mathbf{w})$ to convince a verifier who only holds $\mathbf{x}$ that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$. A SNARK is non-interactive in the sense that there is no two-way interaction between the parties, and succinct if the proof size and verification times are quasi-linear in $|\mathbf{x}|$ and poly-logarithmic to $|\mathbf{w}|$. More formally, a SNARK is a tuple of three PPT algorithms defined as follows:

$\text{pk}, \text{vk} \leftarrow \text{Gen}(1^\lambda)$: It takes as input the security parameter λ and outputs the proving and verification public parameters.

$\pi \leftarrow \text{Prove}(\text{pk}, \mathbf{x}, \mathbf{w})$: Generates a SNARK proof π showing that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$.

$0, 1 \leftarrow \text{Verify}(\text{vk}, \mathbf{x}, \pi)$: Outputs 1 if π is valid and 0 otherwise.

A SNARK is (perfectly) complete if for any $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$ it holds that $\Pr[\text{Verify}(\text{vk}, \mathbf{x}, \pi) = 1 \text{ where } \pi \leftarrow \text{Prove}(\text{pk}, \mathbf{x}, \mathbf{w})] = 1$. Furthermore, a SNARK is knowledge sound if for any PPT $\mathcal{P}$ and honestly generated pk, vk there exists an extractor $\mathcal{E}$ such that:

$$\Pr[(\mathbf{x}, \pi), \mathbf{w} \leftarrow (\mathcal{P} \parallel \mathcal{E})(\text{pk}, \text{vk}) : (\mathbf{x}, \mathbf{w}) \notin \mathcal{R} \wedge \text{Verify}(\text{vk}, \mathbf{x}, \pi) = 1] < \text{negl}(\lambda)$$

Dynamic SNARKs [73, 81]. Finally, we say that a SNARK is dynamic if there exists a PPT algorithm that takes as input an old instance/witness and a new one and generates a SNARK proof for the latter in time that is proportional to $k = \mathcal{D}(\mathbf{w}, \mathbf{w}')$ (for a distance metric $\mathcal{D}$) and sublinear in $|\mathbf{w}|$. In more detail, this algorithm is defined as follows:

$\pi', \mathbf{aux}' \leftarrow \text{Update}(\text{pk}, (\mathbf{x}, \mathbf{w}), (\mathbf{x}', \mathbf{w}'), \pi, \mathbf{aux})$: Takes as input the old and new instance/witness pairs, the proof π as well as the auxiliary data (possibly produced by the prove algorithm) and outputs the updated auxiliary data $\mathbf{aux}'$ and a SNARK proof π' showing that $(\mathbf{x}', \mathbf{w}') \in \mathcal{R}$.

Remark 1 *Note that we slightly adapt the definition of updatability from [73, 81] by replacing the explicit use of Hamming distance with a more general metric $\mathcal{D}$ between the old and new witnesses. We believe this more general version of the definition can account for additional subtle overheads such as computing (and not just proving) $\mathbf{w}'$, which might not always be captured by Hamming distance as well as differences in the arithmetization techniques. For instance, in our dynamic SNARK for the circuit satisfiability relation $\mathcal{R}_C$, we define $\mathcal{D}$ as the number of wires of C whose values have been changed (which essentially corresponds to the time needed to update all intermediate gates of C when using the algorithm of Lemma 3). Interestingly, this is equivalent to the Hamming distance between two (extended) witnesses produced by the Plonk arithmetization, as these witnesses must include the value of each wire of C . In contrast, the Hamming distance between two extended witnesses as defined in our construction (see Section 6) corresponds only to the number of gate values that have been changed, which might not always be identical to $\mathcal{D}$ (e.g., there might exist some specific edge cases where a value of a gate changes but all the values of its parents remain unchanged). That said, our construction can also work for $\mathcal{D}$ being the Hamming distance as long as there exists an update algorithm that operates in that time.*

3.4 (Updatable) Lookup Arguments

At a very high level, lookup arguments enable a prover to commit to a large table $\mathbf{x}$ and then, provided a lookup pair $(\mathcal{I}, \mathbf{t})$, produce a SNARK proof showing that $\mathbf{t}[j] = \mathbf{x}[i_j]$ for all $i_j \in \mathcal{I}$. In this work, we define a lookup argument as a tuple of six PPT algorithms defined as follows:

$\mathbf{pk}, \mathbf{vk} \leftarrow \text{Gen}(1^\lambda, n)$: Takes as input the maximum table size and produces some proving and verification public parameters.

$C, \mathbf{aux} \leftarrow \text{Commit}(\mathbf{pk}, \mathbf{x})$: Takes as input the table $\mathbf{x}$ and produces its commitment as well as some auxiliary data useful for producing lookup proofs.

$\pi_{\text{lkp}} \leftarrow \text{Lookup}(\mathbf{pk}, \mathbf{x}, \mathbf{aux}, \mathbf{t}, \mathcal{I})$: Takes as input $\mathbf{x}$, $\mathbf{aux}$ and the lookup claim $(\mathcal{I}, \mathbf{x})$ and produces a SNARK proof for the relation $\mathcal{R}_{\text{lkp}}$ defined as:

$$\mathcal{R}_{\text{lkp}} = \{(C, C_{\mathbf{t}}, C_{\mathcal{I}}); (\mathbf{x}, \mathbf{t}, \mathcal{I}) : \mathbf{t}[j] = \mathbf{x}[i_j] \text{ for all } i_j \in \mathcal{I}\}$$

$0, 1 \leftarrow \text{VerifyLookup}(\mathbf{vk}, C, \pi_{\text{lkp}}, C_{\mathbf{t}}, C_{\mathcal{I}})$: Takes as input the commitments of $\mathbf{x}, \mathbf{t}, \mathcal{I}$ and validates the correctness of π_{lkp} .

Note that if the time needed to produce π_{lkp} is sub-linear in n , then we say that the lookup argument is doubly efficient. Furthermore, we say that a lookup argument is updatable if the following functionalities work in (possibly amortized) sub-linear time to n :

$C', \mathbf{x}', \mathbf{aux}', \pi_{\text{upd}} \leftarrow \text{Update}(\mathbf{pk}, \mathbf{x}, \mathbf{aux}, \mathbf{t}, \mathcal{I})$: Takes as input $\mathbf{x}$, $\mathbf{aux}$ and the indexes $\mathcal{I}$ to be updated into the corresponding values of $\mathbf{t}$ and (1) updates the auxiliary information, table and commitment to $\mathbf{aux}'$, $\mathbf{x}'$ and C' and (2) produces a SNARK proof for the relation $\mathcal{R}_{\text{upd}}$ defined as:

$$\mathcal{R}_{\text{upd}}\{(C, C', C_{\mathbf{t}}, C_{\mathcal{I}}); (\mathbf{x}, \mathbf{x}', \mathbf{t}, \mathcal{I}) : \\ \mathbf{x}'[i_j] = \mathbf{t}[j] \text{ for all } i_j \in \mathcal{I} \wedge \mathbf{x}'[j] = \mathbf{x}[j] \forall j \in [n] \setminus \mathcal{I}\}$$

$0, 1 \leftarrow \text{VerifyUpdate}(\mathbf{vk}, C, C', \pi_{\text{upd}}, C_{\mathbf{t}}, C_{\mathcal{I}})$: Takes as input the commitments of $\mathbf{x}, \mathbf{x}', \mathbf{t}, \mathcal{I}$ and validates the correctness of π_{upd} .

4 Doubly Efficient Code-Based Lookup Arguments

In this section, we show how to construct our lookup arguments based on error-correction codes. Initially, in Sections 4.1 and 4.2 we present two different IOPs for proving $\mathcal{R}_{\text{lkp}}$. The first extends the ideas introduced in Section 2 to high-dimensional tensor codes. The second, focuses on two-dimensional tensor codes and incorporates additional techniques to further optimize prover time, both asymptotically and concretely. Finally, in Section 4.3 we show how to use the above constructions in order to build lookup arguments including TENSEUP.

4.1 Our IOP For $\mathcal{R}_{\text{lkp}}$ Using High-Dimensional Tensor Codes

In this section, we present our first construction which arises by recursively performing the reduction phase over a d -wise tensor code $\mathcal{C} : \mathcal{C}_1 \otimes \dots \otimes \mathcal{C}_d$ until the verifier ends up with a reduced lookup claim over a small table encoded by $\mathcal{C}_d$, which can be proven directly through the lookup phase. To simplify the protocol description, we will set the number of repetitions l to 1. Later, we will describe how to efficiently modify the following protocol for $l > 1$, which will allow us to construct an IOP for $\mathcal{R}_{\text{lkp}}$ with negligible round-by-round (knowledge) soundness error. Furthermore, throughout the description of the protocol and its analysis, we will assume that the verifier has access to an honestly computed codeword $\mathbf{C}$. At the end of this section, we will show how to remove this assumption.

Construction 1: Our IOP for High-Dimensional Tensor Codes

Encoding. Let $\mathbf{x} \in \mathbb{F}^n$ be the input message and $d \in \mathbb{N}$ the recursion depth. Furthermore, let $\mathcal{C}_1, \dots, \mathcal{C}_d$ be error-correction codes such that $\mathcal{C}_i : \mathbb{F}^{n_i} \rightarrow \mathbb{F}^{n'_i}$. We recursively define our code as $\mathcal{C} : \mathcal{C}_d \otimes \mathcal{C}^{(d-1)}$ where $\mathcal{C}^{(i)} = \mathcal{C}_i \otimes \mathcal{C}^{(i-1)}$. With that in mind, the encoding algorithm works as follows:

1. Parse $\mathbf{x}$ as a matrix $\mathbf{X}_d \in \mathbb{F}^{n_d \times n/n_d}$ and compute its interleaved code $\mathbf{X}'_d$ by encoding each column of $\mathbf{X}_d$ using $\mathcal{C}_d$.
2. Recursively encoding every row of $\mathbf{X}'_d$ using $\mathcal{C}^{d-1}$.

Proving Phase. Here, $\mathcal{P}$ holds the codeword $\mathbf{C}$, all intermediate interleaved codes produced from each recursion step of the encoding functionality as well as the lookup claim $(\mathcal{I} \in [n]^k, \mathbf{t} \in \mathbb{F}^k)$. $\mathcal{V}$ holds $[\mathbf{C}]$ and oracles to the encodings of $\mathcal{I}, \mathbf{t}$ denoted with $([\mathcal{I}], [\mathbf{t}])$. $\mathcal{P}$ interacts with $\mathcal{V}$ as follows:

Reduction Phase. In each recursion step $m \in [d]$, parties interact in order to reduce lookup claims of a table $\mathbf{X}_m \in \mathbb{F}^{n_m \times (n_{m-1} \cdots n_1)}$ (defined as $(\mathcal{I}^{(m)}, \mathbf{t}^{(m)})$) into claims of a single row of the interleaved code $\mathbf{X}'_m \in \mathbb{F}^{n'_m \times (n_{m-1} \cdots n_1)}$ (defined as $(\mathcal{I}^{(m-1)}, \mathbf{t}^{(m-1)})$), where $(\mathcal{I}^{(d)}, \mathbf{t}^{(d)}) = (\mathcal{I}, \mathbf{t})$. Initially, both parties set $m = d$ and work as follows:

1. $\mathcal{P}$: Decompose each $i_j \in \mathcal{I}^{(m)}$ as $(i_j^{(m,1)}, i_j^{(m,2)})$, corresponding to the row and column indices of $\mathbf{X}_m$, and set $\mathcal{I}^{(m,1)}, \mathcal{I}^{(m,2)}$ to be the sets of all row and column indices derived from $\mathcal{I}$. Send the oracles of the encodings of $\mathcal{I}^{(m,1)}, \mathcal{I}^{(m,2)}$ to $\mathcal{V}$.
2. $\mathcal{P}$: Form $\mathbf{R} \in \mathbb{F}^{n_m \times |\mathcal{I}^{(m,2)}|}$ such that $\mathbf{R}[:, j] = \mathbf{X}_m[:, i_j]$ for all $i_j \in \mathcal{I}^{(m,2)}$. Then send $[\mathbf{R}]$, the oracle (of an encoding) of $\mathbf{R}$, to the verifier.
3. $\mathcal{V}$: Sample a row index $q_m \in [n'_m]$ and send it to $\mathcal{P}$, who replies with an oracle of the encoding of $\mathbf{t}^{(m-1)} = \mathbf{R}'[q_m, :]$ where $\mathbf{R}'[:, i] = \text{Enc}_{\mathcal{C}_m}(\mathbf{R}[:, i])$ for all i .
4. $\mathcal{P}\text{-}\mathcal{V}$: Interact using a IOP for proving the following relation: $\mathcal{R}_{\text{Red}} = \{([\mathbf{t}^{(m-1)}], [\mathbf{R}], [\mathcal{I}^{(m)}], [\mathcal{I}^{(m,1)}], [\mathcal{I}^{(m,2)}], [\mathbf{t}^{(m)}], s); (\mathbf{R}, \mathbf{t}^{(m)}, \mathbf{t}^{(m-1)}) : \mathbf{R}[i_j^{(1)}, i_j^{(2)}] = \mathbf{t}^{(m)}[j], \forall i_j \in \mathcal{I}^{(m)} \wedge \mathbf{t}^{(m-1)} = \mathbf{R}'[q_m, :] \wedge \mathcal{I}^{(m,2)}, \mathcal{I}^{(m,1)} \text{ are valid}\}$
5. $\mathcal{V}$ updates its input to $([\mathcal{I}^{(m-1)}] \leftarrow [\mathcal{I}^{(m,2)}], [\mathbf{t}^{(m-1)}])$ and $\mathcal{P}$ forms $\mathbf{X}_{m-1} \in \mathbb{F}^{n_{m-1} \times (n_{m-2} \cdots n_1)}$, the parsed matrix of the q_m -th row of $\mathbf{X}'_m$. Both parties set $m \leftarrow m-1$. If $m = 1$, move to the lookup phase; otherwise, set $m \leftarrow m-1$ and repeat step 1 for the new lookup claim.

Lookup Phase. In this phase, $\mathcal{P}$ convinces the verifier about the correctness of the final reduced claim $(\mathcal{I}^{(1)}, \mathbf{t}^{(1)})$:

1. $\mathcal{P}$: Parse $\mathbf{X}_1 \in \mathbb{F}^{n_1 \times 1}$ as a vector $\mathbf{x}'$ and send $[\mathbf{x}']$, the oracle of its encoding, to $\mathcal{V}$.
2. $\mathcal{V}$: Sample a query $q_1 \in [n'_1]$ and send it to $\mathcal{P}$, who replies with $q = \text{Enc}_{\mathcal{C}_1}(\mathbf{x}')[q_1]$.
3. $\mathcal{V}$: Query $[\mathbf{C}]$ to check if $\mathbf{C}[q_d, \dots, q_1] = q$ and interact with $\mathcal{P}$ via an IOP for the relation $\mathcal{R}_{\text{Mem}} = \{([\mathbf{t}^{(1)}], [\mathbf{x}'], q, [\mathcal{I}^{(1)}]); (\mathbf{x}', \mathbf{t}^{(1)}) : \mathbf{t}'[j] = \mathbf{x}'[i_j] \forall i_j \in \mathcal{I}^{(1)} \wedge q = \text{Enc}_{\mathcal{C}_1}(\mathbf{x}')[q_1]\}$

Achieving Negligible Soundness Error. As we briefly discussed in Section 2, the above construction has a constant but large soundness error of $\approx 1 - \delta_C$. Consequently, we can not use it directly to construct our lookup argument which requires negligible soundness error. Naively, one could overcome this limitation by repeating the above construction l times, enough to derive a negligible decoding error. This, however, would blowup the proof size and verification time by a factor of l . Instead, in each recursion step we batch prove these repetitions. To achieve this, we apply the following changes: When $m = d$, the reduction phase remains almost the same, with the only exception being that in step 3, $\mathcal{V}$ samples l row indices $(q_1^{(1)}, q_1^{(2)}, \dots, q_1^{(l)})$ and the prover replies with the oracle of the concatenation of the queried rows, namely, $\mathbf{t}^{(d-1)} = (\mathbf{t}^{(d-1,1)}, \dots, \mathbf{t}^{(d-1,l)})$ where $\mathbf{t}^{(d-1,i)} = \mathbf{R}'[q_1^{(i)}, :]$. Likewise, $\mathcal{R}_{\text{Red}}$ is modified to work over all these rows. After the completion of this first phase, both parties end up with essentially l independent “lookup” claims $(\mathcal{I}^{(d-1)}, \mathbf{t}^{(d-1,i)})$ corresponding to $\mathbf{X}'_d$ —the $q_1^{(i)}$ -th row of $\mathbf{X}'_d$. Then, for each reduction step $m = d-1, \dots, 2$, both parties perform the reduction steps for each claim lookup claim $(\mathcal{I}^{(m)}, \mathbf{t}^{(m,i)})$ over the table $\mathbf{X}_m$ “in parallel”. This is first done by sending the oracle of $\mathbf{R} = (\mathbf{R}^{(1)}, \dots, \mathbf{R}^{(m)})$, where $\mathbf{R}^{(i)} \in \mathbb{F}^{n_m \times |\mathcal{I}^{(m,2)}|}$ corresponds to the columns of $\mathbf{X}_m$ at $\mathcal{I}^{(2)}$, to the verifier which replies with l random queries $(q_m^{(1)}, \dots, q_m^{(l)})$. Once again, the prover replies with the oracle of $\mathbf{t}^{(m-1)} = (\mathbf{t}^{(m-1,1)}, \dots, \mathbf{t}^{(m-1,l)})$, where now $\mathbf{t}^{(m-1,i)} = \mathbf{R}'^{(i)}[q_m^{(i)}, :]$. Finally, both parties interact over an IOP for the modified $\mathcal{R}_{\text{Red}}$ which repeats its checks l times for each instance. Similarly, we do the same for the lookup phase where $\mathcal{V}$ ends-up with l point queries over $[\mathbf{C}]$.

Based on the above, we state the following:

Theorem 3 *Let $\mathcal{C}_i : \mathbb{F}^{n_i} \rightarrow \mathbb{F}^{\rho_i^{-1} n_i}$ be an error-correction code and $\mathcal{C} = \mathcal{C}_d \otimes \dots \otimes \mathcal{C}_1$ be their tensor product. Then, $\mathcal{C}$ has a round-by-round (knowledge) sound IOP for the relation $\mathcal{R}_{\text{IOP}, \mathcal{C}, 1}$ with $\mathcal{O}(Pt_{\text{IOP}}(k \cdot n_d) + \sum_{i \in [d-1]} Pt_{\text{IOP}}(l \cdot k \cdot n_i))$ prover complexity, $\mathcal{O}(l + \sum_i Ps_{\text{IOP}}(l \cdot k \cdot n_i))$ proof size, $\mathcal{O}(l + \sum_i Vt_{\text{IOP}}(l \cdot k \cdot n_i))$ verification time where $l = \frac{-\lambda}{\log(1-\delta_C)}$ and $\delta_C = \prod \delta_{\mathcal{C}_i}$ and $\tilde{\mathcal{O}}(n \cdot \prod \rho_i^{-1})$, $\prod \rho_i$ encoding time and rate.*

Before proceeding to the proof of the above theorem, observe that different selections of d , $(n_1, \dots, n_d)$ and $(\rho_1, \dots, \rho_d)$ might lead to constructions with different performance characteristics. Here we consider the two extreme cases:

Constant Rate and Fast Encoding Times. From the above construction, one can derive a codeword $\bar{\mathcal{C}}$ with a constant rate and (quasi)-linear encoding time. One way to achieve this is by setting d and $(\rho_1, \dots, \rho_d)$ to be constants and $n_1 = \dots = n_d = \sqrt[d]{n}$. Assuming that the underlying IOP has quasi-linear prover time and poly-logarithmic proof size with respect to the witness and instance size, the above parameters result in a constant-rate codeword that has an IOP for $\mathcal{R}_{\text{IOP}, \mathcal{C}, 1}$ with $\tilde{\mathcal{O}}(\lambda k \sqrt[d]{n})$ prover time and $\mathcal{O}(\text{polylog}(k, n))$ proof size and verification time.

Almost Optimal Prover Time. The other “extreme” case is to significantly reduce the prover time by “sacrificing” the rate of the code. Our starting point, is to set $n_1 = \dots = n_d = \log^a n$ and thus $d = \frac{\log n}{a \log \log n}$ (derived by solving the equation $n = (\log n)^{ad}$). Then, we select $\{\mathcal{C}_i\}_{i \in [d]}$ codes satisfying the singleton bound (such as Reed-Solomon codes [66]) and set their rate to be $\rho_i = \frac{1}{\log n}$ and thus their relative minimum distance becomes $\delta_i = 1 - \frac{1}{\log n}$. Note that in this way, the codeword size becomes:

$$(\log n)^d \cdot n = (\log n)^{\frac{\log n}{a \log \log n}} \cdot n = n \cdot 2^{\frac{\log n \cdot \log \log n}{a \log \log n}} = n \cdot n^{1/a} = n^{1+1/a}$$

while $\delta_C = (1 - \frac{1}{\log n})^d > (1 - \frac{1}{\log n})^{\log n} > 1/3$ for any $\log n \geq 6$, as $(1 - 1/x)^x$ is increasing for $x > 1$. Consequently, l will be $O(\lambda)$, which eventually leads to a code that has $\mathcal{O}(n^{1+1/a})$ codeword size, $\tilde{\mathcal{O}}(n^{1+1/a})$ encoding time and an IOP for $\mathcal{R}_{\text{lkp}, \mathbf{C}, 1}$ with $\tilde{\mathcal{O}}(\lambda k \log^{a+1} n / (a \log \log n))$ prover time, and $\mathcal{O}(\text{polylog}(k, n))$ proof size and verification time (assuming that the underlying IOP has quasi-linear prover time and poly-logarithmic verification cost).

Removing the Well-Formedness Assumption of $\mathbf{C}$. Note that the above theorem holds if and only if the verifier has oracle access to an honestly computed codeword $\mathbf{C}$. This is a reasonable assumption in certain limited scenarios, e.g., when the table is publicly accessible and remains static. In the general case, however, the verifier must additionally ensure that each queried element $\mathbf{C}[i]$ is indeed equal to $\text{Enc}_C(\mathbf{x})[i]$. To address this issue, we can modify our construction in several ways. One approach is to employ the same technique used in lookup arguments with projection codes [48, 49], where the verifier checks the correctness of each queried value $\mathbf{C}[i]$ using (r)LCCs for tensor codes [45]. This approach, however, could blow-up the proof size and verification time by $\mathcal{O}(\max(\{n_i\}_{i \in [d]}))$.

Alternatively, during the encoding phase, the prover does not send the oracle of $\mathbf{C}$ but instead an oracle of an “IOP-friendly” encoding $\mathbf{x}$. Then, for each element $\mathbf{C}[i]$, it produces a proof π_i showing that $\mathbf{C}[i] = \text{Enc}_C(\mathbf{x})[i]$. This allows the verifier to replace its point query over $\mathbf{C}$ with a verification of the corresponding proof. Using the techniques of [65, 79] and the BCS transformation of [79], as we will do when compiling our IOPs into lookup arguments in Section 4.3, one can compute all these proofs in time $\tilde{\mathcal{O}}(|\mathbf{C}|)$, and these proofs add only a poly-logarithmic overhead (per point query) in terms of verification complexity.

4.1.1 Proof Of Theorem 3

Performance Analysis. In terms of prover time, observe that each reduction phase in **Construction 1** requires the prover to invoke the IOP prover of $\mathcal{R}_{\text{red}}$ for a witness of size $\mathcal{O}(k \cdot n_m)$. Likewise, in the lookup phase, the prover has to invoke an IOP prover for a witness of size that is proportional to n_1 . So, the total prover time will be $\mathcal{O}(\text{Pt}_{\text{IOP}}(k_1) + \sum_{i \in d, \dots, 2} \text{Pt}_{\text{IOP}}(kn_i))$. Note that when we apply the modifications in order to make **Construction 1** have a negligible round-by-round (knowledge) soundness error, then the above complexity becomes $\mathcal{O}(\text{Pt}_{\text{IOP}}(kn_d) + \text{Pt}_{\text{IOP}}(ln_1) + \sum_{i \in \{d-1, \dots, 2\}} \text{Pt}_{\text{IOP}}(lkn_i))$, where $l = \mathcal{O}(\lambda)$. Note that the prover complexity in the first reduction phase is unaffected by l , since the size of $\mathbf{R}$ will remain kn_d .

As for the proof size, observe that in every reduction phase the prover sends a constant number of oracles and produces an IOP proof. The same happens in the lookup phase, and additionally the verifier has to perform l queries over the oracle of $\mathbf{C}$. Consequently, the total proof size will be $\mathcal{O}(l + \sum \text{Ps}_{\text{IOP}}(lkn_i))$. Likewise, we argue about the verification time.

Completeness Proof. Here, we will show that **Construction 1** is an IOP with perfect completeness over $\mathcal{R}_{\text{lkp}, \mathbf{C}, 1}$. In particular, we will show that provided a valid lookup pair $(\mathcal{I}, \mathbf{t})$, the verifier will always accept. Since $(\mathcal{I}, \mathbf{t})$ is valid, it must hold that $\mathbf{R}[:, j] = \mathbf{X}_d[:, i_j]$ and thus $\mathbf{t}^{(d-1)} = \mathbf{R}'[q_m, :] = \mathbf{X}'_d[q_m, \mathcal{I}^{(2)}]$. So, the new claim $(\mathbf{t}^{(d-1)}, \mathcal{I}^{(2)})$ will be valid with probability 1 and $\mathcal{R}_{\text{Red}}$ will be satisfying meaning that the verifier will pass the IOP proof with probability 1. Since $(\mathcal{I}^{(d-1)}, \mathbf{t}^{(d-1)})$ is valid, we can repeat the same argument for each reduction phase until we end up in the lookup phase. Note that since $(\mathcal{I}^{(1)}, \mathbf{t}^{(1)})$ is valid, $\mathbf{x}'$ is also valid and $\mathbf{x}'[\mathcal{I}^{(1)}] = \mathbf{t}^{(1)}$. Consequently, $\mathcal{R}_{\text{Mem}}$ must be satisfied, which means that the verifier passes all IOP proofs with

probability 1. Finally, since $\mathbf{x}'$ is valid, it must hold that $\text{Enc}_{C_1}(\mathbf{x}')[q_1] = \mathbf{C}[q_1, \dots, q_d]$, so the verifier will also pass the point query test with probability 1.

Proof of Round-by-Round (Knowledge) Soundness Error. Here, we will prove that the modified version of **Construction 1** (by adding repetition) is an IOP with round-by-round (knowledge) soundness error of $\epsilon_{\text{rbr}} = \max(\epsilon_{\text{IOP}}, (1 - \delta_C)^l)$ (assuming that the verifier has oracle access to a well formed codeword $\mathbf{C} \in \mathcal{C}$). For this, we follow the same proof technique with that of [15].

Namely, assume that the initial transcript $\mathbf{tr}_0 = ([\mathcal{I}], [\mathbf{t}], [\mathbf{C}])$ belongs in a doomed state $\mathcal{D}$. This means that there exists at least one $i_j \in \mathcal{I}$ such that $\mathbf{t}[j] \neq \mathbf{x}[i_j]$. We will show that for each interaction step of **Construction 1**, the verifier leaves the doomed state with negligible probability.

First Step of Reduction Phase ($m = d$):

1. Conditioned that $\mathbf{tr}_0 \in \mathcal{D}$, we will show that $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] < (1 - \delta_d)^l$, where $\mathbf{tr}_1 = (\mathbf{tr}_0, [\mathbf{R}], \{q_d^{(1)}, \dots, q_d^{(l)}\}, [\mathbf{t}^{(d-1)}])$. Note that $\mathbf{tr}_1$ leaves the doomed state if and only if (1) $(\mathbf{R}, \mathbf{t}^{(d-1)}, \mathbf{t}^{(d)}, \mathcal{I})$ satisfy $\mathcal{R}_{\text{Red}}$ and (2) $\mathbf{R}'[q_d^{(i)}, :] = \mathbf{X}'_d[q_d^{(i)}, \mathcal{I}^{(2)}]$ for all $i \in [l]$. Note, however, that this can only happen with probability at most $(1 - \delta_d)^l$. Since $\mathbf{R}$ must satisfy $\mathcal{R}_{\text{Red}}$, then, there must exist $i_j \in \mathcal{I}$ such that $\mathbf{t}[j] = \mathbf{R}[j^{(1)}, j^{(2)}] \neq \mathbf{X}[i_j^{(1)}, i_j^{(2)}]$ and, subsequently, there exists a column i such that $\mathbf{R}[:, i] \neq \mathbf{X}[:, i_j]$. Consequently $\Delta(\mathbf{R}[:, i], \mathbf{X}'[:, i]) > \delta_d$. In other words, the probability that $\mathbf{R}'[q_d^{(i)}, :] = \mathbf{X}'_d[q_d^{(i)}, \mathcal{I}]$ holds for a single i is $1 - \delta_d$ and therefore the probability $\mathbf{tr}_1$ leaves the doomed state is $(1 - \delta_d)^l < \text{negl}(\lambda)$.
2. Conditioned that $\mathbf{tr}_1 \in \mathcal{D}$, we will show that $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] \leq \epsilon_{\text{IOP}}$, where $\mathbf{tr}_2 = (\mathbf{tr}_1, \mathbf{tr}_{\text{IOP,Red}})$ with $\mathbf{tr}_{\text{IOP,Red}}$ being the communication transcript produced from the IOPP of the reduction phase and ϵ_{IOP} its round-by-round soundness error. Note that since $\mathbf{tr}_1 \in \mathcal{D}$, it holds that either $\mathcal{R}_{\text{Red}}$ is not satisfied or there exists at least one $i \in [l]$ such that $\mathbf{R}'[q_d^{(i)}, :] \neq \mathbf{X}'_d[q_d^{(i)}, \mathcal{I}]$. In the first case, the prover leaves the doomed state with probability at most $\epsilon_{\text{IOP,Red}}$. Otherwise, it remains in a doomed state with probability 1.

Rest of Reduction Phase ($m < d$):

1. Conditioned that $\mathbf{tr}_{2(d-m)} \in \mathcal{D}$, we will show that $\Pr[\mathbf{tr}_{2(d-m)+1} \notin \mathcal{D}] < (1 - \delta_d \dots \delta_m)^l$ where $\mathbf{tr}_{2(d-m)+1} = (\mathbf{tr}_{2(d-m)}, [\mathbf{R} = (\mathbf{R}^{(1)} | \dots | \mathbf{R}^{(l)})], \{q_m^{(1)}, \dots, q_m^{(l)}\}, [\mathbf{t}^{(m-1)} = (\mathbf{t}^{(m-1,1)}, \dots, \mathbf{t}^{(m-1,l)})])$. Note that $\mathbf{tr}_{2(d-m)+1}$ leaves the doomed state if and only if (1) $(\mathbf{R}, \mathbf{t}^{(m-1)}, \mathbf{t}^{(m)}, \mathcal{I}^{(m)})$ satisfy $\mathcal{R}_{\text{Red}}$ and (2) $\mathbf{R}^{(i)'}[q_m^{(i)}, :] = \mathbf{X}_m^{(i)'}[q_m^{(i)}, \mathcal{I}^{(m,2)}]$ for all $i \in [l]$. As we will show next, this can only happen with negligible probability. First, note that since $\mathbf{tr}_{2(d-m)} \in \mathcal{D}$, it holds that there must exist at least one incorrect claim $(\mathcal{I}^{(m)}, \mathbf{t}^{(m,i)})$ and all previous relations $\mathcal{R}_{\text{Red}}$ must be satisfied (since otherwise the prover will remain in the doomed state throughout the entire protocol). Because of that, and because $\mathbf{R}$ must satisfy $\mathcal{R}_{\text{Red}}$, then, there must exist $i_j \in \mathcal{I}$ and $z \in [l]$ such that $\mathbf{R}^{(z)}[j^{(1)}, j^{(2)}] \neq \mathbf{X}_m^{(z)}[i_j^{(1)}, i_j^{(2)}]$ and, subsequently, $\mathbf{R}$ does not equal to $(\mathbf{X}_m^{(1)}[:, \mathcal{I}^{(m,2)}], \dots, \mathbf{X}_m^{(l)}[:, \mathcal{I}^{(m,2)}])$. Based on this, it remains to bound the probability that $(\mathbf{R}^{(1)'}[q_m^{(1)}, :], \dots, \mathbf{R}^{(l)'}[q_m^{(l)}, :]) = (\mathbf{X}_m^{(1)'}[q_m^{(1)}, \mathcal{I}^{(m,2)}], \dots, \mathbf{X}_m^{(l)'}[q_m^{(l)}, \mathcal{I}^{(m,2)}])$. For this, we will find the probability that $\mathbf{t}^{(m,i)} = \mathbf{R}^{(i)'}[q_m^{(i)}, :] \neq \mathbf{X}_m^{(i)'}[q_m^{(i)}, \mathcal{I}^{(m,2)}]$ for a fixed $i \in [l]$. Note that the latter inequation holds if and only if $(\mathcal{J}^{(m+1)}, \mathbf{t}^{(m+1,i)})$ is incorrect. Let δ'_{m+1} be the probability this happens. Conditioned that we follow the same argument with that of the first step (1), to show that the probability $\mathbf{t}^{(m,i)}$ is incorrect is δ_m . So, the overall

probability will be $p = \delta_m \delta'_{m+1}$. Recursively applying this argument for δ'_{m+1} , we can show that $\delta'_{m+1} = \delta_{m+1} \cdots \delta_d$, and thus $p = \delta_m \cdots \delta_d$.

Consequently, the probability that $\mathbf{R}^{(i)'}[q_m^{(i)}, :] = \mathbf{X}_m^{(i)'}[q_m^{(i)}, :]$ is $1 - \delta_d \cdots \delta_m$, and thus the probability that $\mathbf{R}^{(i)'}[q_m^{(i)}, :] = \mathbf{X}_m^{(i)'}[q_m^{(i)}, :]$ for all $i \in [l]$, is $(1 - \delta_d \cdots \delta_m)^l$ which is negligible. Hence, the probability $\mathbf{tr}_{2(d-m)+1}$ leaves the doomed state is $(1 - \delta_d \cdots \delta_m)^l < \text{negl}(\lambda)$.

2. Now, conditioned that $\mathbf{tr}_{2(d-m)+1} \in \mathcal{D}$, we show that $\Pr[\mathbf{tr}_{2(d-m+1)} \notin \mathcal{D}] < \epsilon_{\text{IOP}}$, where $\mathbf{tr}_{2(d-m+1)} = (\mathbf{tr}_{2(d-m)+1}, \mathbf{tr}_{\text{IOP,Red}})$ using the same argument as in the first step.

Lookup Phase:

1. Conditioned that $\mathbf{tr}_{2d} \in \mathcal{D}$, we will show that $\Pr[\mathbf{tr}_{2d+1} \notin \mathcal{D}] < (1 - \delta_c)^l$, where $\mathbf{tr}_{2d+1} = (\mathbf{tr}_{2d}, [\mathbf{x}' = (\mathbf{x}^{(1)'}, \dots, \mathbf{x}^{(l)'})], (q_1^{(1)}, \dots, q_1^{(l)}), \mathbf{q} = (q^{(1)}, \dots, q^{(l)})$). Observe that $\mathbf{tr}_{2d+1}$ leaves the doomed state if and only if (1) $(\mathbf{x}', \mathcal{I}^{(1)}, \mathbf{t}^{(1)}, \mathbf{q})$ satisfies $\mathcal{R}_{\text{Mem}}$ and (2) $\mathbf{q}[i] = \mathbf{C}[q_d^{(i)}, \dots, q_1^{(i)}]$ for all $i \in [l]$. Note that since $\mathbf{tr}_{2d} \in \mathcal{D}$, there should exist at least one $i \in [l]$ such that $\mathbf{t}^{(1,i)} \neq \mathbf{X}_1^{(i)'}[\mathcal{I}^{(1)}, :]$ and so, $\mathbf{X}_1^{(i)'} \neq \mathbf{x}^{(i)'}$, since otherwise $\mathcal{R}_{\text{Mem}}$ would not be satisfied. Consequently, finding the above probability translates to finding an upper bound for the event that $\mathbf{C}[q_d^{(i)}, \dots, q_1^{(i)}] = \mathbf{q}[i]$ for all $i \in [l]$. By following the same argument as in the reduction phase (1), it is not hard to see that the probability that $\mathbf{C}[q_d^{(i)}, \dots, q_1^{(i)}] \neq \mathbf{q}[i]$ is $\delta_c = \prod_i \delta_i$. As a result, the probability of leaving the doomed state is $(1 - \delta_c)^l < \text{negl}(\lambda)$.
2. Conditioned that $\mathbf{tr}_{2d+1} \in \mathcal{D}$ we finally show that $\Pr[\mathbf{tr}_{2d+2} \notin \mathcal{D}] < \epsilon_{\text{IOP}}$ following the same argument as in (1.a) where $\mathbf{tr}_{2d+2} = (\mathbf{tr}_{2d+1}, \pi_{\text{IOP,Mem}})$ with $\pi_{\text{IOP,Mem}}$ being the communication transcript produced by running the IOP for $\mathcal{R}_{\text{Mem}}$.

To conclude the proof, we need to find the round-by-round knowledge soundness error κ_{rbr} of our IOP. First, we assume that our extractor has access to the lookup table $\mathbf{X}$. Note that in practice, when our protocol will be used as a building block for constructing a lookup argument, our extractor will be able to retrieve $\mathbf{X}$ from the commitment phase. With that in mind, our extractor will invoke the extractors of the underlying IOP in order to retrieve all witness data such as all lookup claims and matrices $\mathbf{R}$. Next, in order to prove round-by-round knowledge soundness, we use the identical argument with that of [15]. Namely, for each $i \in [2d + 1]$ we need show that if $\Pr[\mathbf{tr}_{i+1} \notin \mathcal{D}] > \epsilon_{\text{rbr}}$ then, it must hold that $\mathbf{tr}_i \notin \mathcal{D}$ with overwhelming probability. This comes directly from our round-by-round soundness proof. Assuming that $\mathbf{tr}_i \in \mathcal{D}$, then it should hold that $\Pr[\mathbf{tr}_{i+1} \notin \mathcal{D}] < \epsilon_{\text{rbr}}$, thus reaching a contradiction. By repeating the same argument, we end up an initial state that is non-doomed and thus, the extracted $(\mathcal{I}, \mathbf{t})$ is a valid lookup pair with overwhelming probability.

4.2 Optimized IOP for Two-Dimensional tensor codes

In this section, we show how to construct a code with $\mathcal{O}(n \log n)$ codeword size that has an IOP for $\mathcal{R}_{\text{lkp}}$ with $\mathcal{O}(\sqrt{kn})$ prover time and poly-logarithmic proof size and verification time. Recall that the starting point of our construction is the construction described in the previous Section for $d = 2$, but encompassing the following key modifications: First, instead of encoding the lookup table $\mathbf{x}$ with a single tensor code, we encode $\mathbf{x}$ using $\log n$ codewords—one for each possible (padded) number of lookups k . We do this such that each m -th tensor code has the form $\mathbf{C}_m \in \mathbb{F}^{\sqrt{n/k} \times \sqrt{kn}}$,

where $k = 2^m$ is the number of lookups. Note that this is enough to achieve the desired asymptotic complexity. Our second modification significantly improves the concrete performance. In particular, in the lookup phase, instead of letting the prover run an IOP over a witness of size $\mathcal{O}(l\sqrt{kn})$, we adapt our protocol so that it only performs $l\sqrt{kn}$ field operations “outside” the IOP, and only run the latter for a witness of size $\mathcal{O}(\sqrt{kn})$. This is done by letting the prover aggregate the multiple lookup claims produced in the reduction phase into a single one. In more detail, our protocol works as follows:

Construction 2

Encoding. Let $\mathbf{x} \in \mathbb{F}^n$ be the input message. The encoding algorithm samples (constant rate and distance) error-correction codes $\{\mathcal{C}_1^{(i)} : \mathbb{F}^{\sqrt{n/2^i}} \rightarrow \mathbb{F}^{n'_{1,i}}, \mathcal{C}_2^{(i)} : \mathbb{F}^{\sqrt{n/2^i}} \rightarrow \mathbb{F}^{n'_{2,i}}\}_{i \in [\log n]}$. For $i \in \{1, \dots, \log n\}$, it works as follows:

1. Parse $\mathbf{x}$ as a matrix $\mathbf{X}_i \in \mathbb{F}^{\sqrt{n/2^i} \times \sqrt{n/2^i}}$ and compute $\mathbf{X}'_i$ by encoding each column of $\mathbf{X}_i$ using $\mathcal{C}_1^{(i)}$.
2. Compute the tensor code $\mathbf{C}_i$ by encoding each row of $\mathbf{X}'_i$ using $\mathcal{C}_2^{(i)}$.

Proving Phase. Here, $\mathcal{P}$ holds $\{\mathbf{C}_i, \mathbf{X}'_i\}_{i \in [\log n]}$, input vector $\mathbf{x} \in \mathbb{F}^n$ and lookup claim $(\mathcal{I} \in [n]^k, \mathbf{t} \in \mathbb{F}^k)$. $\mathcal{V}$ only holds $\{\mathbf{C}_i\}_{i \in [\log n]}$ and $([\mathcal{I}], [\mathbf{t}])$. Without loss of generality we assume $k = 2^m$. $\mathcal{P}$ and $\mathcal{V}$ interact as follows:

Reduction Phase. This phase is almost identical to the reduction phase of **Construction 1**, with the only difference that the verifier ends-up with an oracle of a linear combination of the l_1 reduced claims. In more detail:

1. $\mathcal{P}$: Decomposes each $i_j \in \mathcal{I}$ as $(i_j^{(1)}, i_j^{(2)})$ corresponding to the row and column indices of $\mathbf{X}_m$. Set $\mathcal{I}^{(1)}, \mathcal{I}^{(2)}$ to be the sets of all row and column indices derived from $\mathcal{I}$. Send the oracles of the encodings of $\mathcal{I}^{(1)}, \mathcal{I}^{(2)}$ to $\mathcal{V}$, denoted with $[\mathcal{I}^{(1)}], [\mathcal{I}^{(2)}]$.
2. $\mathcal{P}$: Form $\mathbf{R} \in \mathbb{F}^{\sqrt{n/2^i} \times |\mathcal{I}^{(2)}|}$ such that $\mathbf{R}[:, j] = \mathbf{X}_m[:, i_j]$ for all $i_j \in \mathcal{I}^{(2)}$. Finally, send $[\mathbf{R}]$, the oracle (of an encoding) of $\mathbf{R}$, to $\mathcal{V}$.
3. $\mathcal{V}$: Sample a set of rows $\mathcal{Q} = \{q_1, \dots, q_{l_1}\} \in [n'_{1,m}]^{l_1}$ as well as a random point $s \in \mathbb{F}$ and send them to $\mathcal{P}$, who replies with the oracle of the encoding of $\mathbf{t}' = \sum s^k \mathbf{R}'[q_k, :]$, where $\mathbf{R}'[:, i] = \text{Enc}_{\mathcal{C}_1^{(m)}}(\mathbf{R}[:, i])$ for all i .
4. $\mathcal{P}$ - $\mathcal{V}$: Interact using a IOP for proving the following relation:

$$\begin{aligned} \mathcal{R}'_{\text{Red}} = \{ & ([\mathbf{t}], [\mathbf{t}'], [\mathbf{R}], [\mathcal{I}], [\mathcal{I}^{(1)}], [\mathcal{I}^{(2)}], \mathcal{Q}, s); (\mathbf{t}, \mathbf{t}', \mathbf{R}, \mathcal{I}, \mathcal{I}^{(1)}, \mathcal{I}^{(2)}) : \\ & \mathbf{R}[i_j^{(1)}, i_j^{(2)}] = \mathbf{t}[j], \forall i_j \in \mathcal{I} \wedge \mathbf{t}' = \sum_{k \in [l_1]} s^k \mathbf{R}'[q_k, :] \wedge \mathcal{I}^{(1)}, \mathcal{I}^{(2)} \text{ are valid} \} \end{aligned}$$

Lookup Phase. Again, this phase is almost identical to that of our first construction, with the exception that now, the prover operates over the aggregated rows of $\mathbf{X}'_m$. In particular:

1. $\mathcal{P}$: Compute $\mathbf{x}' = \sum_{k \in [l_1]} s^k \mathbf{X}'_m[q_k, :]$ and send $[\mathbf{x}']$, the oracle of its encoding to $\mathcal{V}$.
2. $\mathcal{V}$: Samples a query set $\mathcal{Q}' = \{q'_1, \dots, q'_{l_2}\} \in [n_{2,m}]^{l_2}$ and send it to $\mathcal{P}$, who replies with $\mathbf{q}$ claimed to be $\text{Enc}_{\mathcal{C}_2^{(m)}}(\mathbf{x}')[\mathcal{Q}']$.
3. $\mathcal{V}$: Queries $[\mathbf{C}_m]$ on $\mathcal{Q} \times \mathcal{Q}'$ to check if $\mathbf{q}[j] = \sum_{k \in [l_1]} s^k \mathbf{C}_m[q_k, q']$ for all $q' \in \mathcal{Q}'$, and interacts with $\mathcal{P}$ in an IOPP for the following relation: $\mathcal{R}'_{\text{Mem}} = \{([\mathbf{t}'], [\mathbf{x}'], [\mathcal{I}^{(2)}], \mathbf{q}); (\mathbf{x}', \mathbf{t}', \mathcal{I}^{(2)}) : \mathbf{t}'[j] = \mathbf{x}'[i_j] \forall i_j \in \mathcal{I}^{(2)} \wedge \mathbf{q} = \text{Enc}_{\mathcal{C}_2^{(m)}}(\mathbf{x}')[\mathcal{Q}']\}$.

Finally, we claim the following, which we prove in Appendix A.1:

Theorem 4 *Let $\mathcal{C}$ be a code consisting of $\log n$ codes $\mathcal{C}^{(i)} : \mathcal{C}_1^{(i)} \otimes \mathcal{C}_2^{(i)}$ where $\mathcal{C}_1^{(i)} : \mathbb{F}^{\sqrt{n/2^i}} \rightarrow \mathbb{F}^{n'_{1,i}}$ and $\mathcal{C}_2^{(i)} : \mathbb{F}^{\sqrt{n/2^i}} \rightarrow \mathbb{F}^{n'_{2,i}}$ have constant rate and distance, and $\mathcal{C}_2^{(i)}$ displays a proximity gap approaching the unique decoding radius. Then, $\mathcal{C}$ has $1/\log n$ rate, $\tilde{O}(n \log n)$ encoding time, and has a round-by-round (knowledge) sound IOP for $\mathcal{R}_{\text{lkp}}$ with $\mathcal{O}(Pt_{\text{IOP}}(\sqrt{nk}) + l_1 \sqrt{nk})$ prover time, $\mathcal{O}(Vt_{\text{IOP}}(\sqrt{nk}) + l_1 l_2)$ verification time and $\mathcal{O}(Ps_{\text{IOP}}(\sqrt{kn}) + l_1 l_2)$ proof size, where $k = 2^i$ and $l_j = \frac{-\lambda}{\log(1 - \delta_{\mathcal{C}_j^{(i)}}/2)}$.*

4.3 Our Doubly-Efficient Code-Based Lookup Argument

In this section, we show how to construct our doubly efficient code-based lookup argument. Let $\mathcal{C}$ be any error-correction code from Theorems 3 and 4. Our lookup argument works as follows: initially, the **Gen** algorithm will sample a random oracle and generate the code $\mathcal{C}$, the description of which will be appended in $\mathbf{pk}$ along with its proximity parameter δ (e.g., $\delta = 1$ for our first construction and $\delta < \delta_{\mathcal{C}}/4$ for our second). Then, the **Commit** algorithm first encodes $\mathbf{x}$ to $\mathbf{C}$ via $\mathcal{C}$. If $\delta > 0$, it computes $C_{\mathbf{C}}$ —the commitment of $\mathbf{C}$ produced after applying BCS transformation [10]—and produces a SNARK proof π_{IOPP} showing that $\Delta(\mathbf{C}, \mathcal{C}) < \delta$. Otherwise, it sets $C_{\mathbf{C}}$ to be the commitment of an encoding of $\mathbf{x}$, and for each $i \in [|\mathbf{C}|]$, it generates a proof π_i showing that $\mathbf{C}[i] = \text{Enc}_{\mathcal{C}}(\mathbf{x})$ using the techniques of [79, 65], and sets $\pi_{\text{prox}} \leftarrow \emptyset$. Finally, it returns $C \leftarrow (C_{\mathbf{C}}, \pi_{\text{prox}})$. As for the **Lookup** algorithm, the prover invokes the non-interactive prover of the IOP for $\mathcal{R}_{\text{lkp}, \mathcal{C}, \delta}$, derived by applying the BCS transformation in order to produce π_{lkp} . Finally, the verifier not only validates π_{lkp} , but also π_{prox} (if any), which ensures that $\mathbf{C}$ is δ -close to $\mathcal{C}$. **Construction 3** gives a detailed description of our lookup argument.

Construction 3.

pk, vk $\leftarrow \text{Gen}_{\delta}(1^{\lambda}, n)$: Samples a random oracle $\mathcal{H} : \{0, 1\}^* \rightarrow \{0, 1\}^{\lambda}$ and appends it to **pk** and **vk**. Initialize the error-correction code $\mathcal{C}$ and append all of its parameters to **pk**.

$C, \mathbf{aux} \leftarrow \text{Commit}_{\delta}(\mathbf{pk}, \mathbf{x})$: Given as input the table $\mathbf{x} \in \mathbb{F}^n$, the prover invokes $\mathbf{c} \leftarrow \text{Enc}_{\mathcal{C}}(\mathbf{x})$. Then consider the following cases:

1. If $\delta = 0$, set $C_{\mathbf{x}}$ to be the commitment of an encoding of $\mathbf{x}$. Then, for each $i \in [|\mathbf{c}|]$ produce a proof π_i by applying the BCS transformation of [78] over an IOP for the relation

$\mathcal{R}_{C,i} = \{(C_{\mathbf{x}}, i, q); \mathbf{x} : q = \text{Enc}_{\mathcal{C}}(\mathbf{x})[i]\}$. Set $C \leftarrow C_{\mathbf{x}}$ and $\mathbf{aux} \leftarrow \{\mathbf{c}, \mathbf{x}', \{\pi_1, \dots, \pi_{|\mathbf{c}|}\}\}$ where $\mathbf{x}'$ are auxiliary data obtained by running $\text{Enc}_{\mathcal{C}}$ and they are used to efficiently compute lookup proofs (e.g., interleaved codes).

2. Computes the commitment of $C_{\mathbf{c}}$ by committing to $\mathbf{c}$ using the BCS transformation. Then, it runs the non-interactive prover resulted by applying the BCS over an IOPP prover for $\mathcal{R}_{\text{prox}, \mathcal{C}, \delta} = \{[\mathbf{c}]; \mathbf{c} : \Delta(\mathbf{c}, \mathcal{C}) < \delta\}$ and produces the proof π_{IOPP} . Set $C \leftarrow \{C_{\mathbf{c}}, \pi_{\text{IOPP}}\}$ and $\mathbf{aux} \leftarrow \{\mathbf{c}, \mathbf{x}'\}$.

$\pi_{\text{lkp}} \leftarrow \text{Lookup}(\mathbf{pk}, \mathbf{x}, \mathbf{aux}, \mathcal{I}, \mathbf{t})$: Return the lookup proof π_{lkp} produced by running the non-interactive prover resulted after applying the BCS over the IOP prover for $\mathcal{R}_{\text{lkp}, \mathbf{c}, \delta}$.

$0, 1 \leftarrow \text{VerifyLookup}(\mathbf{vk}, C, \pi_{\text{lkp}}, C_{\mathbf{t}}, C_{\mathcal{I}})$: Run the non-interactive verifier resulted after applying the BCS over the IOPP and IOP verifiers of $\mathcal{R}_{\text{IOPP}}$ and $\mathcal{R}_{\text{IOP}}$. Return 1 if all proofs are accepted.

Having established that, we can claim the following, which we prove in Appendix A.2:

Theorem 5 *Assuming that the IOP(P)s used have negligible round-by-round (knowledge) soundness error, then **Construction 3** is a lookup argument with $Pt_{\text{Com}}(n) \in \mathcal{O}(Pt_{\text{Enc}}(n) + Pt_{\text{IOPP}, \text{prox}}(n))$ commitment complexity, $Pt_{\text{lkp}}(k, n) \in \mathcal{O}(Pt_{\text{IOP}, \text{lkp}}(k, n))$ lookup complexity, $Ps(k, n) = \mathcal{O}(\log n \cdot (Ps_{\text{IOPP}, \text{prox}}(n) + Ps_{\text{IOP}, \text{lkp}}(k, n)))$ proof size and $Vt(k, n) \in \mathcal{O}(\log n \cdot (Vt_{\text{IOPP}, \text{prox}}(n) + Vt_{\text{IOP}, \text{lkp}}(k, n)))$ verification time. When we use the IOPs described above, one can obtain lookup arguments with the following performance characteristics:*

1. *If the IOP of Section 4.1 is used with the recursion depth d being a small constant, then we obtain a lookup argument with $\mathcal{O}(n)$ commitment time, $\mathcal{O}(\lambda k \sqrt[n]{n})$ prover time and $\mathcal{O}(\log^2 n)$ proof size and verification time.*
2. *If the IOP of Section 4.1 is used with the recursion depth d being $\frac{\log n}{a \log \log n}$ and a a small constant, then we obtain a lookup argument with $\tilde{\mathcal{O}}(n^{1+1/a})$ commitment time, $\mathcal{O}(\lambda k \frac{\log^a n}{a \log \log n})$ prover time and $\mathcal{O}(\log^3 n)$ proof size and verification time.*
3. *If the IOP of Section 4.2 is used, then we obtain TENSEUP: a lookup argument with $\mathcal{O}(n \log^2 n)$ commitment time, $\mathcal{O}(\lambda \sqrt{kn})$ prover time, and $\mathcal{O}(\log^2 n)$ proof size and verification time.*

5 Transformation From Static To Updatable Lookup Arguments

In this section, we present our transformation that turns any static doubly-efficient lookup argument to an updatable one which we describe in more detail in **Construction 4**. First, during the commitment algorithm, the prover initializes two data structures, $\mathcal{D}_{\mathcal{P}}, \mathcal{D}_{\mathcal{V}}$, defined as follows: $\mathcal{D}_{\mathcal{P}}$ has $\log n$ entries such that each entry $i \in [\log n - 1]$ stores two 2^i -sized lookup tables, along with their auxiliary data necessary for efficiently computing lookup proofs. The $\log n$ -th entry stores only the original lookup table. We define $\mathcal{D}_{\mathcal{V}}$ in a similar way, but instead of storing the tables, we only store their commitments and possibly some additional proofs. After the initialization phase is

complete, the prover invokes the commitment algorithm of the static lookup argument with input $\mathbf{x}$ and sets all $\mathcal{D}_{\mathcal{P}}[i] = \mathcal{D}_{\mathcal{V}}[i] = \perp$ for all $i \in [\log n - 1]$, $\mathcal{D}_{\mathcal{P}}[\log n] = (\mathbf{x}, \mathbf{aux})$ and $\mathcal{D}_{\mathcal{V}}[\log n] = C_{\mathbf{x}}$. Then, to update or lookup $\mathbf{x}$, it works as follows:

Update Algorithm. Provided an update pair $(\mathcal{I} \in [n]^k, \mathbf{t} \in \mathbb{F}^k)$, the prover tries to insert that update into the data structure $\mathcal{D}_{\mathcal{P}}$ in such a way that the updates are sorted in chronological order. Namely, for every i_1, i_2 with $i_1 < i_2$, it must hold that $\mathcal{D}_{\mathcal{P}}[i_1]$ contains a more recent update pair compared to $\mathcal{D}_{\mathcal{P}}[i_2]$. It is not hard to see that by doing so, one can easily have access to the updated table $\mathbf{x}$ by traversing all $\mathcal{D}_{\mathcal{P}}$ with increasing order of indices. What remains is to show how to achieve this in such a way that we get our desirable update cost. Assume for a moment that all update pairs have the same size k . In this case, upon receiving an update request, the prover would check if $\mathcal{D}_{\mathcal{P}}[k'] = \perp$, where $k' = \log k$. In this case, it simply invokes the (static) lookup commitment algorithm with inputs $\mathcal{I}$ and $\mathbf{t}'$ and sets $\mathcal{D}_{\mathcal{P}}[k'] \leftarrow (\mathbf{t}, \mathcal{I}, \mathbf{aux}_{\mathbf{t}}, \mathbf{aux}_{\mathcal{I}})$ and $\mathcal{D}_{\mathcal{V}}[k'] = (C_{\mathbf{t}}, C_{\mathcal{I}})$. Furthermore, it generates a SNARK proof showing that $\mathcal{I}$ is sorted—a crucial property necessary so that we can prove lookups. Now, in case $\mathcal{D}_{\mathcal{P}}[k'] \neq \perp$, the prover first merges both update pairs $(\mathcal{I}, \mathbf{t}), (\mathcal{I}', \mathbf{t}') \leftarrow \mathcal{D}[k']$ into a single table $(\mathcal{I}'', \mathbf{t}'')$ such that the latter consists of the most recent updates (e.g., if there is a common index between $\mathcal{I}$ and $\mathcal{I}'$ the prover keeps the update value of that index stored in $\mathbf{t}$). Finally, the prover generates a SNARK proof for correctly merging these tables, sets $\mathcal{D}[k'] = \perp$, and recursively invokes the update algorithm for the merged lookup pair.

Unfortunately, this approach does not work when the update pairs have different sizes. In particular, for a given k_1 -sized update pair, it may be the case that $\mathcal{D}_{\mathcal{P}}[\log k_1] = \perp$, while there exists some index $i < \log k_1$ such that $\mathcal{D}_{\mathcal{P}}[i] \neq \perp$. Following the algorithm above, the prover would simply insert the update pair at index $\log k_1$ and thus $\mathcal{D}_{\mathcal{P}}$ will no longer be sorted in chronological order. One way to overcome this limitation would be to decompose the update pair to at most $\log k$ update pairs and insert them in a manner that preserves their chronological order. This approach, however, requires the prover to decompose the update—a property, which as we will see later, becomes non-trivial when we utilize the same techniques to construct a dynamic SNARK. Instead, we propose a more generic approach that requires no decomposition and does not incur additional asymptotic overhead. In more detail, we adapt the above algorithm to ensure that for all $i < \log k_1$, $\mathcal{D}_{\mathcal{P}}[i] = \perp$. This is done by merging all update pairs with indices $i < \log k_1$ either with the new update pair, if $\mathcal{D}_{\mathcal{P}}[\log k_1] = \perp$, or with the one stored in $\mathcal{D}_{\mathcal{P}}[\log k_1]$.

Lookup Algorithm. As for our lookup algorithm, the prover takes as input the lookup pair $(\mathbf{t}, \mathcal{I})$, accesses the non-empty tables stored in $\mathcal{D}_{\mathcal{P}}$ from the smallest to largest index, and for each such table $(\mathcal{I}', \mathbf{x}') \leftarrow \mathcal{D}[z]$, it first partitions $(\mathcal{I}, \mathbf{t})$ in two lookup pairs $(\mathcal{I}_1, \mathbf{t}_1), (\mathcal{I}_2, \mathbf{t}_2)$. For the first one, there exists an index set $\mathcal{J}$ such that $\mathcal{I}_1[i] = \mathcal{I}'[j_i]$ for all $j_i \in \mathcal{J}$. As for the second one, it must hold that $\mathcal{I}' \cap \mathcal{I}_2 = \emptyset$. Then, (1) it proves the correctness of $(\mathcal{I}_1, \mathbf{t}_1)$ using $\mathcal{J}$ and invoking the lookup algorithm for the pairs $(\mathcal{I}_1, \mathcal{J})$ and $(\mathbf{t}_1, \mathcal{J})$, and (2) proves that there exists no pair in $(\mathcal{I}_2, \mathbf{t}_2)$ that is included in $(\mathcal{I}', \mathbf{x}')$ which can be easily proved using another invocation of the lookup algorithm and the fact that $\mathcal{I}'$ is sorted (which allows us to prove that no $i \in \mathcal{I}_2$ belongs to $\mathcal{I}'$). Finally, the prover shows that $(\mathcal{I}_1, \mathbf{t}_1) \cup (\mathcal{I}_2, \mathbf{t}_2) = (\mathcal{I}, \mathbf{t})$ and moves to the next entry of $\mathcal{D}_{\mathcal{P}}$ with input the remaining lookup pairs $(\mathcal{I}_2, \mathbf{t}_2)$. Note that if it ends up with the table $\mathcal{D}[\log n]$, then the prover simply invokes the lookup algorithm.

The verifier repeats the same process as the prover but instead of generating proofs, it verifies them.

Construction 4: Transformation that turns static to updatable lookup arguments.

$C, \mathbf{aux} \leftarrow \text{Commit}_{\text{ULA}}(\mathbf{x})$ Given as input the table $\mathbf{x} \in \mathbb{F}^n$, invokes the commit algorithm of the static lookup to obtain C' and $\mathbf{aux}'$. Initialize the functions $\mathcal{D}_{\mathcal{V}}, \mathcal{D}_{\mathcal{P}} : [\log n] \rightarrow \{0, 1\}^*$ by setting $\mathcal{D}_{\mathcal{V}}[i] = \mathcal{D}_{\mathcal{P}}[i] = \perp$ for all $i \in [\log n - 1]$, $\mathcal{D}_{\mathcal{V}}[\log n] = C'$ and $\mathcal{D}_{\mathcal{P}}[\log n] = (\mathbf{x}, \mathbf{aux}')$. Output $C \leftarrow \mathcal{D}_{\mathcal{V}}$ and $\mathbf{aux} = \{\mathcal{D}_{\mathcal{P}}, \mathcal{D}_{\mathcal{V}}\}$.

$C', \mathbf{x}', \mathbf{aux}', \pi_{\text{upd}} \leftarrow \text{Update}_{\text{ULA}}(\mathbf{x}, \mathbf{aux}, \mathcal{I}, \mathbf{t})$: Takes as input $\mathbf{x}, \mathbf{aux}$ as well as the update pair $(\mathcal{I} \in [n]^k, \mathbf{t} \in \mathbb{F}^k)$, sets $\pi_{\text{upd}} \leftarrow \mathcal{D}_{\mathcal{V}}$ and works as follows:

1. If $\mathcal{D}_{\mathcal{P}}[i] = \perp$, for all $i = 1, \dots, k'$ where $k' = \log k$, the prover computes $C_{\mathcal{I}}, \mathbf{aux}_{\mathcal{I}} \leftarrow \text{Commit}(\mathcal{I})$ and $C_{\mathbf{t}}, \mathbf{aux}_{\mathbf{t}} \leftarrow \text{Commit}(\mathbf{t})$ and generate a SNARK proof π_{sort} for the relation $\mathcal{R}_{\text{sort}} = \{C_{\mathcal{I}}; \mathcal{I} : \mathcal{I} \text{ is sorted}\}$. Finally, set $\mathcal{D}_{\mathcal{P}}[k'] \leftarrow (\mathcal{I}, \mathbf{aux}_{\mathcal{I}}, \mathbf{t}, \mathbf{aux}_{\mathbf{t}})$, $\mathcal{D}_{\mathcal{V}}[k'] \leftarrow (C_{\mathcal{I}}, C_{\mathbf{t}}, \pi_{\text{sort}})$ and return $\pi_{\text{upd}} \leftarrow \pi_{\text{upd}} \cup \{\pi_{\text{sort}}\}$, $C' \leftarrow \mathcal{D}_{\mathcal{V}}$ and (pointers) to $\mathbf{aux}' = \{\mathcal{D}_{\mathcal{P}}, \mathcal{D}_{\mathcal{V}}\}$ and $\mathbf{x}'$.
2. Else if $\mathcal{D}_{\mathcal{P}}[i] = \perp$ for all $i = 1, \dots, k' - 1$ but $\mathcal{D}_{\mathcal{P}}[k'] \neq \perp$ first retrieve $(\mathbf{t}', \mathcal{I}')$ from $\mathcal{D}_{\mathcal{P}}[k']$. Then, compute $\mathbf{t}'' = \mathbf{t} \cup \mathbf{t}'[\mathcal{J}]$ and $\mathcal{I}'' = \mathcal{I} \cup \mathcal{J}$ where $\mathcal{J} = \mathcal{I}' \setminus \mathcal{I}$ in such a way so that $(\mathbf{t}'', \mathcal{I}'')$ is sorted with respect to the indexes, properly pad them to the next power of two and generate a CP-SNARK proof π_{merge} for the relation:

$$\mathcal{R}_{\text{Merge}} = \{(C_{\mathbf{t}}, C_{\mathcal{I}}), (C_{\mathbf{t}'}, C_{\mathcal{I}'}), (C_{\mathbf{t}''}, C_{\mathcal{I}''}); (\mathbf{t}, \mathbf{t}', \mathbf{t}'', \mathcal{I}, \mathcal{I}', \mathcal{I}'') : \\ \mathbf{t}'' = \mathbf{t} \cup \mathbf{t}'[\mathcal{J}] \wedge \mathcal{I}'' = \mathcal{I} \cup \mathcal{J} \wedge \mathcal{J} = \mathcal{I}' \setminus \mathcal{I}\}$$

Finally, set $\pi_{\text{upd}} \leftarrow \pi_{\text{upd}} \cup \{\pi_{\text{merge}}\}$, $\mathcal{D}_{\mathcal{V}}[k'] = \mathcal{D}_{\mathcal{P}}[k'] \leftarrow \perp$, $(\mathbf{t}, \mathcal{I}) \leftarrow (\mathbf{t}'', \mathcal{I}'')$. If $\mathbf{t}$ has size k go to step 1 otherwise repeat step 2.

3. Else, consider the following cases:

- (a) If $\mathcal{D}_{\mathcal{P}}[k'] = \perp$ merge $(\mathcal{I}, \mathbf{t})$ with all pairs $(\mathcal{I}_i, \mathbf{t}_i)$ (stored in $\mathcal{D}_{\mathcal{P}}[i]$) for all $i \in [k' - 1]$ to $(\mathcal{I}', \mathbf{t}')$ (while keeping the most recent updates) and produce a proof π_{merge} for the (slightly modified) relation $\mathcal{R}_{\text{Merge}}$. Set $\mathcal{D}_{\mathcal{P}}[i] = \mathcal{D}_{\mathcal{V}}[i] \leftarrow \perp$ for all $i \in [k']$. Finally, invoke step 1 or 2 for $(\mathbf{t}', \mathcal{I}')$.
- (b) Else, find the smallest index z such that $z > k'$ and $\mathcal{D}_{\mathcal{P}}[z] = \perp$. Let $(j_1, \dots, j_l) \in \mathcal{J}$ be the index set containing all indexes such that $\mathcal{D}_{\mathcal{P}}[j_i] \neq \perp$, and $j_l < z$. Retrieve $(\mathbf{t}_i, \mathcal{I}_i) \leftarrow \mathcal{D}_{\mathcal{P}}[j_i]$ for all $j_i \in \mathcal{J}$, merge them into a single update pair $(\mathcal{I}', \mathbf{t}')$ (while keeping the most recent updates). Produce a proof π_{merge} for the (slightly modified) relation $\mathcal{R}_{\text{Merge}}$. Set $\mathcal{D}_{\mathcal{P}}[j_i] = \mathcal{D}_{\mathcal{V}}[j_i] \leftarrow \perp$ for all $j_i \in \mathcal{J}$. Finally, invoke step 1 first for $(\mathbf{t}', \mathcal{I}')$ (but without returning) and then for $(\mathbf{t}, \mathcal{I})$.

$0, 1 \leftarrow \text{VerifyUpdate}_{\text{ULA}}(C_{\text{old}}, C_{\text{new}}, \pi_{\text{upd}}, C_{\mathbf{t}}, C_{\mathcal{I}})$: Extract $\mathcal{D}_{\mathcal{V}, \text{old}}$ from π and check if all sorting proofs are correct and that $C_{\text{old}} = \mathcal{D}_{\mathcal{V}, \text{old}}$. Then, follow the same steps as with the update algorithm but instead of computing merge and sort proofs, validate their correctness and properly update $\mathcal{D}_{\mathcal{V}, \text{old}}$. Finally check if $\mathcal{D}_{\mathcal{V}, \text{old}} = C_{\text{new}}$. If all checks pass, return 1 and 0 otherwise.

$\pi_{\text{lkp}} \leftarrow \text{Lookup}_{\text{ULA}}(\mathbf{x}, \mathbf{aux}, \mathbf{t}, \mathcal{I})$ Provided $\mathbf{x}, \mathbf{aux}$ and a lookup pair $(\mathcal{I}, \mathbf{t})$, generate a lookup proof. Set $\pi_{\text{lkp}} \leftarrow \mathcal{D}_{\mathcal{V}}$, find the (sorted) set $\mathcal{Z} = (z_1, \dots)$ such that $\mathcal{D}_{\mathcal{P}}[z_i] \neq \perp$ and for each $z \in \mathcal{Z}$ retrieve the tables $(\mathbf{x}', \mathbf{aux}_{\mathbf{x}'}, \mathcal{I}', \mathbf{aux}_{\mathcal{I}'}) \leftarrow \mathcal{D}_{\mathcal{P}}[z]$ and work as follows:

1. If $z = \log n$, invoke the lookup algorithm to generate a proof π for $(\mathbf{t}, \mathcal{I})$. Return $\pi_{\text{lkp}} \leftarrow \pi_{\text{lkp}} \cup \pi$.
2. Otherwise, partition $(\mathbf{t}, \mathcal{I})$ to $(\mathbf{t}_1, \mathcal{I}_1), (\mathbf{t}_2, \mathcal{I}_2)$ such that there exists an index set $\mathcal{J}$ where $(\mathbf{t}_1[i], \mathcal{I}_1[i]) = (\mathbf{x}'[j_i], \mathcal{I}'[j_i])$ for all $j_i \in \mathcal{J}$ and $(\mathbf{t}, \mathcal{I}) \cap (\mathbf{t}_2, \mathcal{I}_2) = \emptyset$. Furthermore, find the set $\mathcal{J}'$ such that for each $i_j \in \mathcal{I}_1$, it holds that $\mathcal{I}'[j] < i_j < \mathcal{I}'[j+1]$ for all $(j, j+1) \in \mathcal{J}'$. Finally, compute the commitments of $\mathbf{t}_1, \mathbf{t}_2, \mathcal{I}_1, \mathcal{I}_2, \mathcal{J}, \mathcal{J}', \mathcal{I}''$ where $\mathcal{I}'' = \mathcal{I}[\mathcal{J}']$.
3. Generate three lookup proofs $\pi_{\text{lkp},1}, \pi_{\text{lkp},2}$ and $\pi_{\text{lkp},3}$ for the lookup claims $(\mathbf{t}_1, \mathcal{J})$ (over $\mathbf{x}'$), $(\mathcal{I}_1, \mathcal{J})$ and $(\mathcal{I}'', \mathcal{J})$ (over $\mathcal{I}'$).
4. Show that $\mathcal{I}_2 \cap \mathcal{I}' = \emptyset$ by generating a CP-SNARK proof π_1 for the relation: $\mathcal{R}_1 = \{(C_{\mathcal{I}'}, C_{\mathcal{I}_2}, C_{\mathcal{J}}); (\mathcal{I}'', \mathcal{I}_2, \mathcal{J}) : j_{2i+1} = j_{2i} + 1 \wedge \mathcal{I}''[2i] < \mathcal{I}_2[i] < \mathcal{I}''[2i+1], \forall (j_{2i+1}, j_{2i}) \in \mathcal{J}\}$
5. Show that $(\mathbf{t}_1, \mathbf{t}_2, \mathcal{I}_1, \mathcal{I}_2)$ are computed correctly by generating a CP-SNARK proof π_2 for the relation: $\mathcal{R}_2 = \{(C_{\mathbf{t}}, C_{\mathcal{I}}, \{C_{\mathbf{t}_i}, C_{\mathcal{I}_i}\}_{i \in [2]}); (\mathbf{t}, \mathcal{I}, \{\mathbf{t}_i, \mathcal{I}_i\}_{i \in [2]}) : \{(\mathbf{t}_1[i], \mathcal{I}_1[i])\}_i \cup \{(\mathbf{t}_2[i], \mathcal{I}_2[i])\}_i = \{(\mathbf{t}[i], \mathcal{I}[i])\}_i\}$
6. Set $\pi_{\text{lkp}} \leftarrow \pi_{\text{lkp}} \cup \{C_{\mathbf{t}_1}, C_{\mathbf{t}_2}, C_{\mathcal{I}_1}, C_{\mathcal{I}_2}, C_{\mathcal{J}}, C_{\mathcal{J}'}, C_{\mathcal{I}'}, \pi_1, \pi_2, \pi_{\text{lkp},1}, \pi_{\text{lkp},2}, \pi_{\text{lkp},3}\}$ and $(\mathbf{t}, \mathcal{I}) \leftarrow (\mathbf{t}_2, \mathcal{I}_2)$.

$0, 1 \leftarrow \text{VerifyLookup}_{\text{ULA}}(C, \pi_{\text{lkp}}, C_{\mathbf{t}}, C_{\mathcal{I}})$: First, extract $\mathcal{D}_{\mathcal{V}}$ from π_{lkp} , check if $\mathcal{D}_{\mathcal{V}} = C$ and validate all π_{sort} proofs that lie in all $\mathcal{D}_{\mathcal{V}}[i] \neq \perp$, for $i < \log n - 1$. Finally, iterate through all non-empty entries of $\mathcal{D}_{\mathcal{V}}$, extract from π_{lkp} the lookup and SNARK proofs and validate their correctness.

Finally, we introduce the following theorem, which we prove in Appendix B:

Theorem 6 *Provided a SNARK with $(Pt(\cdot), Vt(\cdot), Ps(\cdot))$ prover, verification time and proof size and a static lookup argument with $(Pt_{\text{com}}(\cdot), Pt_{\text{lkp}}(\cdot), Vt_{\text{lkp}}(\cdot), Ps_{\text{lkp}}(\cdot))$ commitment, prover, verifier times and proof size respectively, our transformation instantiates an Updatable lookup argument with the following performance characteristics:*

- Commitment time of $Pt'_{\text{com}}(n) = Pt_{\text{com}}(n)$.
- (Amortized) update prover time of $Pt'_{\text{upd}}(k) \in \mathcal{O}(k \log n \cdot \frac{Pt(n) + Pt_{\text{com}}(n)}{n})$ and $Vt'_{\text{upd}}(k) \in \mathcal{O}(Vt(n) \log n)$ and $Ps'_{\text{upd}}(k) \in \mathcal{O}(Ps(n) \log n)$ verification time and proof size.
- Lookup time of $Pt'_{\text{lkp}}(k) \in \mathcal{O}((Pt(k) + k \log n) \cdot \log n + \sum_{i \in [\log n]} Pt_{\text{lkp}}(k, n/2^i))$, verification time of $Vt'_{\text{lkp}}(k) \in \mathcal{O}((Vt(k) + Vt_{\text{lkp}}(k, n)) \cdot \log n)$ and proof size of $Ps'_{\text{lkp}}(k) \in \mathcal{O}((Ps(k) + Ps_{\text{lkp}}(k)) \cdot \log n)$.

Remark 2 *Note that if we use the lookup argument from the IOP of Section 4.1 (with $d = \frac{\log n}{a \log \log n}$), and a SNARK with optimal prover time and $\mathcal{O}(\log^2 n)$ verification complexity [74, 19], then we get a doubly efficient updatable lookup argument with $\mathcal{O}(n^{1+1/a})$ commitment time, $\mathcal{O}(kn^{1/a} \log n)$ update and $\mathcal{O}(\lambda k \log^{2+a} n)$ lookup time and $\mathcal{O}(\log^4 n)$ verification complexity. If we use the lookup argument using the IOP of Section 4.2, then, we get a doubly efficient updatable lookup argument with*

$\mathcal{O}(n \log n)$ commitment time, $\mathcal{O}(k \log^3 n)$ update and $\mathcal{O}(\lambda \sqrt{nk})$ lookup complexity, and $\mathcal{O}(\log^3 n)$ verification complexity.

Remark 3 (De-Amortization of the Update Algorithm) We note that if all update requests have the same size k , then the above construction can be de-amortized using the techniques of [60, 59]. Namely, for each level i , $\mathcal{D}_{\mathcal{P}}[i]$ maintains at most three pairs of lookup tables sorted with respect to a chronological order. On each update (or even lookup) request, the prover traverses $\mathcal{D}_{\mathcal{P}}$ from top to bottom and in each level i spends $\mathcal{O}(k \cdot (T_{\mathcal{P}}(2^i) + T_{\text{com}}(2^i))/2^i)$ time merging the two oldest tables in $i - 1$. As for the lookups, they remain almost the same with the difference that the prover repeats the same lookup process for all three table pairs of $\mathcal{D}_{\mathcal{P}}[i]$ from the newest to the oldest one. Finally, we believe that the above techniques can also be extended to support variable update sizes. In particular, if case 3.a holds, then we append the merged table to an appropriate pair of $\mathcal{D}_{\mathcal{P}}[k' + 1]$ (if the new table has size $2^{k'+1}$) which is guaranteed to be empty since $\mathcal{D}_{\mathcal{P}}[k']$ is empty or directly to $\mathcal{D}_{\mathcal{P}}[k']$ otherwise. As for 3.b, the prover first directly merges all tables in $\mathcal{D}_{\mathcal{P}}[i], i \in 1, \dots, k'$ forming at most 6 tables of size $2^{k'}$. Then, it sequentially inserts these tables and finally insets the original one. We leave a detailed construction and proof of correctness as future work.

6 Static-to-Dynamic SNARKs via Updatable Lookups

We start by introducing some notation. Let $C : \mathbb{F}^n \rightarrow \mathbb{F}^m$ be an arithmetic circuit and $T_C : \mathbb{F}^{|C|} \rightarrow \mathbb{F}^*$ be a function such that $T_C[i]$ returns the value, operation type and pointers to the children and parents of the gate of C with id i , where the ids $\{1, \dots, n\}$ and $\{|C| - m, \dots, |C|\}$ correspond to the input and output gates respectively. Let $\mathbf{T}_C$ (where $|\mathbf{T}_C| \in \mathcal{O}(|C|)$) be the function T_C parsed as a vector, $\mathbf{x} \in \mathbb{F}^n, \mathbf{y} \in \mathbb{F}^m$ be the input and output of C . We say that $\mathbf{T}_C$ is *valid* with respect to $\mathbf{x}, \mathbf{y}$, if and only if, the input and output values of $\mathbf{T}_C$ correspond to $\mathbf{x}$ and $\mathbf{y}$ respectively, $\mathbf{y} = C(\mathbf{x})$, and the values of $\mathbf{T}_C$ corresponding to intermediate gates of C are evaluated correctly.

Based on the above, we can show that there exist two RAM programs Π_C, Π_U that validate or update $\mathbf{T}_C$ in time $\mathcal{O}(|C|)$ and $\mathcal{O}(k = \mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C))$ respectively, where $\mathcal{D}$ corresponds to the number of gates that have descendant at least one updated input gate. At a very high level, Π_C accesses each gate of $\mathbf{T}_C$ in post-order traversal and ensures that it has been evaluated correctly. As for Π_U , takes a list of changes $(\mathcal{I}, \mathbf{t})$ (over the original input), inserts them in a queue and repeatedly de-queues a pair (i, v) , updates the gate with index i , and inserts all of its parents to the queue. Crucially, when $\mathbf{T}_C$ is committed via a doubly efficient updatable lookup argument, one can prove the correctness of Π_U in time proportional to $\mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C)$ and only sublinear in $|\mathbf{T}_C|$ (see Appendix C.2 for more details our RAM model, the programs and their proof generation).

Detailed Description of our Compiler. Now, we have everything we need to construct our compiler. To simplify discussion, we will focus on the case where for each update takes $\mathcal{O}(k)$ time. In **Construction 5**, which gives a detailed description of our compiler, we show how to remove this simplification by employing the same techniques as in Section 5.

Having established that, the prover initially obtains the first instance/witness pair $(\mathbf{x}, \mathbf{w})$, it produces a lookup table $\mathbf{T}_C$, computes its commitment $C_{\mathbf{T}_C}$ using a doubly efficient updatable lookup argument and generates a SNARK proof π_C for the program Π_C . It additionally initializes three data structures, $\mathcal{D}_{\mathcal{V}}, \mathcal{D}_{\mathcal{P}}, \mathcal{D}_T : [\log |C|] \rightarrow \{0, 1\}^*$, maintaining proofs, updates and lookup tables respectively. Initially, the prover sets $\mathcal{D}_{\mathcal{P}}[i] = \mathcal{D}_{\mathcal{V}}[i] = \mathcal{D}_T[i] = \perp$ for all $i \in \{1, \dots, \log |C| - 1\}$, $\mathcal{D}_{\mathcal{V}}[\log |C|] = (\mathbf{x}, C_{\mathbf{w}}, C_{\mathbf{T}_C}, \pi_C)$, $\mathcal{D}_{\mathcal{P}}[\log |C|] = (\{1, \dots, n\}, \mathbf{w})$, and $\mathcal{D}_T[\log |C|] = (\mathbf{T}_C, \mathbf{aux})$,

where **aux** denotes all auxiliary data necessary for efficiently performing lookups and updates over lookup table $\mathbf{T}_C$.

Next, upon receiving an update request consisting of the instance and witness changes, denoted with $(\mathcal{I}_x, \mathbf{x}_d)$ and $(\mathcal{I}, \mathbf{w}_d)$ respectively, the prover sets $\pi \leftarrow \mathcal{D}_V$ and operates based on the following two cases:

Case 1: If $\mathcal{D}_P[k'] = \perp$, (where $k' = \log k$) the prover generates a SNARK proof π_U for the correct computation of the Π_U program with inputs the most recent table $\mathbf{T}_C^{(k')}$, the new table $\mathbf{T}_C^{(k')}$ (produced after the update) and the update pairs. Note that the most recent table is stored in $\mathcal{D}_T[k']$, where $k'' > k'$ is the smallest index such that $\mathcal{D}_T[k''] \neq \perp$. Next, it updates $\mathcal{D}_V, \mathcal{D}_P$, sets $\mathcal{D}_T[k'] = (\mathbf{T}_C^{(k')}, \mathbf{aux}^{(k')})$ and returns $\pi \leftarrow \pi \cup (C_{\mathbf{T}_C^{(k')}}, \pi_U)$, and returns π .

Case 2: If $\mathcal{D}_P[k'] \neq \perp$, the prover retrieves $(\mathcal{I}^{(k')}, \mathbf{w}_d^{(k')}) \leftarrow \mathcal{D}_P[k']$, and merges it with $(\mathcal{I}, \mathbf{w}_d)$ (such that it keeps the most recent updates if duplicate indexes exist). In this way, it produces a new update request $(\mathcal{I}', \mathbf{w}_d')$, for which it generates a proof π_{Merge} that it has been computed honestly, and recursively invokes the update algorithm for $(\mathcal{I}', \mathbf{w}_d')$ after setting $\mathcal{D}_P[k'] = \mathcal{D}_V[k'] = \mathcal{D}_T[k'] = \emptyset$ and $\pi \leftarrow \pi \cup \pi_U$.

As for the verification algorithm, it takes as input π and the new instance $\mathbf{x}$, and first extracts the original verification state $\mathcal{D}_V$ (e.g., before applying the update) and checks the validity of each committed table $\mathbf{T}_C$ by verifying all updates proof as well as the proof of $\mathcal{D}_V[\log |C|]$. Next, it follows the same procedure as the prover but instead of computing merging proofs and the update proof, it merely validates their correctness.

Performance-wise, it is not hard to see that the proof size and verification times incur at most a logarithmic overhead with respect to a single SNARK and lookup proof. As for the prover complexity, while the per-update (amortized) for computing all proofs is sublinear in $|C|$, unfortunately, the total prover cost is $\mathcal{O}(|C|)$. That is because in **Case 1**, the prover updates $\mathcal{D}_T$ at k' by writing the entire lookup table $\mathbf{T}_C^{(k')}$ along with the necessary auxiliary data $\mathbf{aux}^{(k')}$. Fortunately, there is a generic way to overcome that issue: in the given level k' , the prover simply stores the changes between $\mathbf{T}_C^{(k')}, \mathbf{aux}^{(k')}$ and $\mathbf{T}_C^{(k')}, \mathbf{aux}^{(k')}$ (e.g., via a hash table) which by definition should be proportional to k . Next, in order to access $\mathbf{aux}^{(k')}[i]$ (and $\mathbf{T}_C^{(k')}[i]$ resp.), the prover first checks if i belongs to the list of changes and, if this is true, it retrieves the desired value. Otherwise, it moves to the next level and repeats the same procedure until it ends up querying **aux** directly. This only adds a $\mathcal{O}(\log |C|)$ additional overhead per prover query over $\mathcal{D}_T$. Interestingly, observe that when the updatable lookup argument is constructed our transformation (see Section 5), the above modification is not necessary since all updates of the original table as well as its auxiliary data will already be stored at the desirable level, thus avoiding that multiplicative logarithmic overhead.

Construction 5: Compiler from Updatable Lookups To Dynamic SNARKs

$\pi \leftarrow \text{Prove}(\mathbf{pp}, \mathbf{x}, \mathbf{w})$: Form the table $\mathbf{T}_C^{(\log |C|)}$, invoke $C^{(\log |C|)}, \mathbf{aux}^{(\log |C|)} \leftarrow \text{ULA.Commit}(\mathbf{T}_C^{(0)})$ and run a SNARK prover for RAM over the program Π_C (as described in Lemma 3) with input $(\mathbf{x}, \mathbf{w})$ and $\mathbf{T}_C^{(\log |C|)}$. Let π_C be the generated proof. Finally, set $\mathcal{D}_V[\log |C|] = (C^{(\log |C|)}, \mathbf{x}, C_{\mathbf{w}}, \pi_C)$, $\mathcal{D}_P[\log |C|] = (\mathcal{I}, \mathbf{w})$, $\mathcal{D}_T[\log |C|] = (\mathbf{T}_C^{(\log |C|)}, \mathbf{aux}^{(\log |C|)})$ and $\mathcal{D}_V[i] = \mathcal{D}_P[i] = \mathcal{D}_T[i] = \perp$ for all $i \in \{1, \dots, \log |C| - 1\}$. Output $\pi \leftarrow \mathcal{D}_V$.

$\pi \leftarrow \text{Update}(\mathbf{pp}, (\mathcal{I}_{\mathbf{x}}, \mathbf{x}_d), (\mathcal{I}, \mathbf{w}_d))$: Let $k' = \log k$ (with $k = \mathcal{D}(\mathbf{T}'_C, \mathbf{T}_C)$ where $\mathbf{T}_C$ the latest lookup table and $\mathbf{T}'_C$ the table after invoking Π_U). Set $\pi \leftarrow \mathcal{D}_{\mathcal{V}}$ and work as follows:

1. If $\mathcal{D}_{\mathcal{P}}[i] = \perp$ for all $i \in [k']$, the prover generates a SNARK proof π_U for proving the correct computation of Π_U on input $(\mathcal{I}, \mathbf{w}'_d), \mathbf{T}_C^{(k')}$ and $\mathbf{T}_C^{(k')}$ where k'' the smallest index such that $k'' < k'$ and $\mathcal{D}_{\mathcal{P}}[k''] \neq \perp$ and $\mathbf{T}_C^{(k')}$ the new lookup table produced after applying the changes. Finally, it sets $\mathcal{D}_{\mathcal{V}}[k'] = (\mathcal{I}_{\mathbf{x}}, \mathbf{x}_d, C_{\mathcal{I}}, C_{\mathbf{w}_d}, C_{\mathbf{T}_C^{(k')}}), \mathcal{D}_{\mathcal{P}}[k'] = (\mathcal{I}, \mathbf{w}_d), \mathcal{D}_T[k'] = (\mathbf{T}_C^{(k')}, \mathbf{aux}^{(k')})$. Return $\pi \leftarrow \pi \cup \{C_{\mathbf{T}_C^{(k')}}, \pi_U\}$.
2. Else if $\mathcal{D}_{\mathcal{P}}[k'] \neq \perp$ but $\mathcal{D}_{\mathcal{P}}[i] = \perp$ for all $i \in [k' - 1]$ the prover retrieves $\mathcal{I}^{(k')}, \mathbf{w}_d^{(k')}$ and $(\mathcal{I}_{\mathbf{x}}^{(k')}, \mathbf{x}_d^{(k')})$ from $\mathcal{D}_{\mathcal{P}}$ and $\mathcal{D}_{\mathcal{V}}$ respectively. Then, it merges $(\mathcal{I}^{(k')}, \mathbf{w}_d^{(k')})$ with $(\mathcal{I}, \mathbf{w}_d)$ forming $(\mathcal{I}', \mathbf{w}'_d)$ (by properly removing the duplicate indexes) and generates a SNARK proof π_{Merge} for the relation:

$$\begin{aligned} \mathcal{R}_{\text{Merge}} &= \{(C_{\mathcal{I}^{(k')}}, C_{\mathbf{w}_d^{(k')}}), (C_{\mathcal{I}'}, C_{\mathbf{w}'_d}), (C_{\mathcal{I}}, C_{\mathbf{w}_d}); (\mathcal{I}^{(k')}, \mathbf{w}_d^{(k')}, \mathcal{I}', \mathbf{w}'_d, \mathcal{I}, \mathbf{w}_d) : \\ &(\mathcal{I}', \mathbf{w}'_d) = (\mathcal{I}^{(k')}, \mathbf{w}_d^{(k')}) \cup \{(i_j, \mathbf{w}_d^{(k')}[j])\}_{i_j \in \mathcal{I}^{(k')} \setminus \mathcal{I}}\} \end{aligned}$$

Set $\pi \leftarrow \pi \cup (\pi_{\text{Merge}}, C_{\mathcal{I}'}, C_{\mathbf{w}'_d}), \mathcal{D}_{\mathcal{V}}[k'] = \mathcal{D}_{\mathcal{P}}[k'] = \mathcal{D}_T[k'] = \perp$ and recursively invoke the update algorithm with the new input $(\mathcal{I}', \mathbf{w}'_d)$ and $(\mathcal{I}'_{\mathbf{x}}, \mathbf{x}'_d)$ where the latter consists of the properly merged differences in the instance.

3. Else consider the following cases:

- (a) If $\mathcal{D}_{\mathcal{P}}[k'] = \perp$ but there exists $i \in [k' - 1]$ such that $\mathcal{D}_{\mathcal{P}}[i] \neq \perp$, then, retrieve all $(\mathcal{I}^{(i)}, \mathbf{w}_d^{(i)})$ and $(\mathcal{I}_{\mathbf{x}}^{(i)}, \mathbf{x}_d^{(i)})$ from the non-empty $\mathcal{D}_{\mathcal{P}}, \mathcal{D}_{\mathcal{V}}$ for all $i < k'$. Properly merge all updates with $(\mathcal{I}, \mathbf{w}_d)$ and $(\mathcal{I}_{\mathbf{x}}, \mathbf{x}_d)$ to $(\mathcal{I}', \mathbf{w}'_d)$ and $(\mathcal{I}'_{\mathbf{x}}, \mathbf{x}'_d)$ and generate a SNARK proof π_{Merge} showing the correct computation of $(\mathcal{I}', \mathbf{w}'_d)$. Finally, set $\pi \leftarrow \pi \cup \pi_{\text{merge}}$, clear all data in $\mathcal{D}_{\mathcal{V}}[i], \mathcal{D}_{\mathcal{P}}[i], \mathcal{D}_T[i]$ for all $i \in [k']$ and invoke the update algorithm with inputs $(\mathcal{I}', \mathbf{w}'_d)$ and $(\mathcal{I}'_{\mathbf{x}}, \mathbf{x}'_d)$.
- (b) Else, merge all instance and witness updates from all $\mathcal{D}_{\mathcal{V}}$ and $\mathcal{D}_{\mathcal{P}}$ respectively for $i \in [k']$. Invoke the SNARK prover to compute π_{merge} (set $\pi \leftarrow \pi \cup \pi_{\text{merge}}$) showing that the merged witness updates are computed correctly and run steps 1 or 2 with the merged inputs and witnesses but without returning. Finally, set $\mathcal{D}_{\mathcal{V}}[i] = \mathcal{D}_{\mathcal{P}}[i] = \mathcal{D}_T[i] = \perp$ for all $i \in [k']$ and invoke step 1 for $(\mathcal{I}, \mathbf{w}_d, \mathcal{I}_{\mathbf{x}}, \mathbf{x}_d)$.

$0, 1 \leftarrow \text{Verify}(\mathbf{pp}, \mathbf{x}, \pi)$: Extract from π the initial data-structure $\mathcal{D}_{\mathcal{V}}$. If the remaining proof is empty, then, parse $\mathcal{D}_{\mathcal{V}}[\log |C|]$ as $(C^{(\log |C|)}, \mathbf{x}^{(\log |C|)}, C_{\mathbf{w}}, \pi_C)$, check if $\mathbf{x}^{(\log |C|)} = \mathbf{x}$ and return the output of the SNARK verifier. Otherwise, for every $i \in \{\log |C|, \dots, 1\}$ validate the certification or update proof of each $\mathcal{D}_{\mathcal{V}}[i] \neq \perp$ (using the commitment of the previous lookup table). This ensures that the most recent table is valid. While doing this, also construct the latest instance $\mathbf{x}'$ and use $\mathbf{x}$ to compute its difference, namely, $\mathcal{I}_{\mathbf{x}}, \mathbf{x}_d$. Then, work as follows:

1. If $\mathcal{D}_{\mathcal{V}}[i] = \perp$ for all $i \in [k']$, extract $C_{\mathbf{T}_C^{(k')}}, \pi_U$ from π and return the output of the SNARK verifier for $\mathcal{P}_U$ on input $\mathcal{I}_{\mathbf{x}}, \mathbf{x}_d$.

2. Else if $\mathcal{D}_V[k'] \neq \perp$ but $\mathcal{D}_V[i] = \perp$ for all $i \in [k' - 1]$ compute the merged instance $(\mathcal{I}', \mathbf{x}'_d)$, invoke the SNARK verifier to validate the correctness of the pre-image of the merged witness updates $C_{\mathbf{w}'_d}, C_{\mathcal{I}'}$ and recursively invoke steps 1 or 2.
3. Else:
 - (a) If $\mathcal{D}_V[k'] = \perp$ but there exists $i \in [k' - 1]$ such that $\mathcal{D}_V[i] \neq \perp$ compute the merged updated instance $(\mathcal{I}'_{\mathbf{x}}, \mathbf{x}'_d)$, validate the correctness of the pre-image of $C_{\mathbf{w}'_d}, C_{\mathcal{I}'}$ by invoking the SNARK verifier for π_{merge} and go to steps 1,2 using $C_{\mathbf{w}'_d}, C_{\mathcal{I}'}, \mathcal{I}'_{\mathbf{x}}, \mathbf{x}'_d$.
 - (b) Else, follow the same process as in (a) but without using $(\mathcal{I}_{\mathbf{x}}, \mathbf{x}_d, C_{\mathbf{w}_d}, C_{\mathcal{I}})$. After validation of the former merged input is finished, validate the correctness of the latter following step 1.

We finally claim the following, which we prove in Appendix C.3:

Theorem 7 *Provided a doubly efficient lookup argument with a prover complexity of $Pt_{ULA}(k, n) = Pt_{ULA,lp}(k, n) + Pt_{ULA,upd}(k, n)$, verification time of $Vt_{ULA}(k, n) = Vt_{ULA,lp}(k, n) + Vt_{ULA,upd}(k, n)$ and proof size of $Ps_{ULA}(k, n) = Ps_{ULA,lp}(k, n) + Ps_{ULA,upd}(k, n)$ (where k the number of lookups/updates over a table of size n) and a SNARK with prover time of $Pt(k)$, $Vt(k)$, $Ps(k)$ prover time, verification time and proof size, then **Construction 5** is a dynamic SNARK with the following performance characteristics:*

1. *The prover algorithm has prover complexity of $Pt(|C|)$, verification time of $Vt(|C|)$ and proof size of $Ps(|C|)$.*
2. *The update algorithm has a per-update amortized prover complexity of:*

$$\mathcal{O}\left(\sum_{i \in [\log(|C|/k)]} \frac{1}{2^{i-1}} (Pt_{ULA}(2^{i-1}k, |C|) + 2^{i-1}k \log |C| + Pt(2^{i-1}k))\right)$$

and worst-case proof size and verification time of $\log |C| \cdot (Ps_{ULA}(|C|, |C|) + Ps(|C|))$ and $\log |C| \cdot (Vt_{ULA}(|C|, |C|) + Vt(|C|))$ respectively.

Finally, we note that when we use the doubly-efficient lookup arguments from Remark 2 and a SNARK with optimal prover complexity and poly-logarithmic proof size and verification time [74, 19], then, **Construction 5** leads to the first transparent and post-quantum secure dynamic SNARK that solely relies in the Random-Oracle Model with the following performance characteristics: poly-logarithmic proof size and verification time and either $\mathcal{O}(k \log |C| \cdot (|C|^{1/a} + \lambda \log^{(1+a)} |C|))$ (if we use the construction of Section 4.1 with $d = \log n / (a \log \log n)$) or $\mathcal{O}(\lambda \sqrt{k|C|} + k \log^3 |C|)$ (if we use the construction of Section 4.2), where $k = \mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C)$.

Remark 4 *If we use Merkle Trees to efficiently provide read/write access to the tables for the prover (following the ADS+SNARK paradigm mentioned in the Introduction), we obtain a transparent and post-quantum secure SNARK with poly-logarithmic verifier complexity and a prover complexity of $\mathcal{O}(k \log^2 |C|)$.*

7 Experiments

We fully implemented and experimentally evaluated TENSEUP and in this section we present our findings.

Implementation Details & Experimental Setup. We implemented TENSEUP in C++ with approximately 10,000 lines of code. To encode the table, we used tensor-products of Reed-Solomon codes with rate $1/2$. Furthermore, we implemented the IOP of Section 4.2 following the PIOP+PCS [20, 31] paradigm. We used the GKR protocol [42] as the underlying PIOP but further optimized it using specialized sumchecks for proving specific computations such as FFTs [55] and Basefold [77] as the underlying PCS. For field operations, we used the 61-bit Mersenne prime extension field of [80] and Blake3 [58] for hash operations. Our code is open-sourced and available at the anonymized repo [1].

All experiments were conducted on an AWS EC2 c6i.8xlarge instance running Ubuntu 24.04 LTS with 64 GB of memory. To provide a fair comparison between implementations, all reported benchmarks were executed single-threaded on the same machine unless otherwise stated. Lookup times were averaged across five repetitions. Numbers for setup and preprocessing were measured once for each table/decomposition configuration and reused across the five repetitions.

Benchmarking TENSEUP. Table 2 reports the lookup time, verification time, and proof size of our scheme when generating lookup proofs for tables of size $n = 2^{22}, 2^{24}, 2^{26}$ and $k = 128, 1024$ and 8192 lookups. Regarding the prover time, observe that for fixed k , the prover time becomes slower with the increased table size. E.g., for $k = 1024$, the prover time increases from 0.8 to 4 seconds as n grows from 2^{22} - 2^{26} . Additionally, observe that for fixed table sizes, TENSEUP scales sub-linearly with increasing k , as expected from our analysis.

As for the proof size, it ranges from 800 to 1,355 KBs where more than 80% of the proof size is derived from the IOP for $\mathcal{R}_{\text{Ikp}}$. We note that our implementation is focused on optimizing the prover’s performance and that in practice, one can significantly reduce the proof size with a small cost on prover time (e.g., by replacing BaseFold with PC schemes with better proof sizes such as [4] or utilizing batching techniques). Finally, our verification time is concretely fast, and remains smaller than 42ms.

Comparison with Other Lookup Arguments. Next, we compare our scheme against existing (practical) lookup arguments—which to date are those based on the KZG scheme—by measuring the preprocessing time (i.e., the time needed to compute the commitment of the table and its auxiliary

Lookup Size k	Table size n	Online Prover Time	Verifier Time	Proof Size (KiB)
128	2^{22}	305 ms	25.0 ms	800
	2^{24}	641 ms	27.3 ms	887
	2^{26}	1,454.0 ms	29.8 ms	985
1024	2^{22}	812.6 ms	27.3 ms	887
	2^{24}	1,732.1 ms	30.0 ms	974
	2^{26}	4,046.7 ms	33.7 ms	1,073
8192	2^{22}	2,152 ms	31.2 ms	971.5
	2^{24}	5,233 ms	36.2 ms	1,239
	2^{26}	12,168 ms	41.1 ms	1,355

Table 2: Summary of the performance of TENSEUP (single-thread).

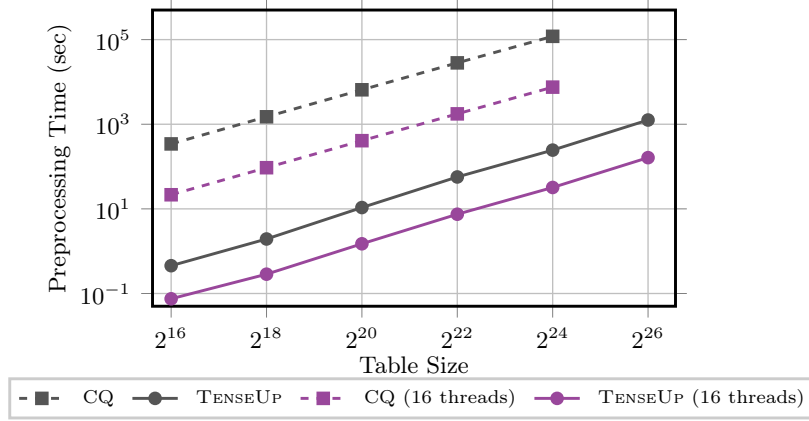


Figure 2: Preprocessing time of TENSEUP and CQ for $n = 2^{16}$ - 2^{26} .

data), lookup time, proof size and verification time. More specifically, we compare our scheme against CQ [38], a doubly-efficient lookup argument based on KZG, using the highly optimized library provided in [75, 2]. We did not compare our scheme with the only existing one based PST [63], since its lookup time is $\times 2.5$ slower compared to that of [38].

Preprocessing. Figure 2 shows the preprocessing time with respect to an increasing table size ranging from 2^{16} to 2^{26} . As shown in the figure, our preprocessing time vastly outperforms that of CQ by a factor of $600\times$. For instance, for a table of size $n = 2^{20}$, our preprocessing time is 10.7 seconds, compared to the 108 minutes for CQ. On top of that, our scheme requires far less space compared to CQ ranging from 4.2GB to 17.2GB for $n = 2^{24}$ - 2^{26} opposed to CQ which could only run up to a table size of 2^{24} . This is one of the major advantages of our scheme, as the slow preprocessing of CQ is a known drawback that limits the applicability of doubly-efficient lookup arguments in many prior works (e.g., [36, 50]). It can also lead to slow update times when using the transformation described in Appendix 5 to build updatable lookup arguments. We also include the preprocessing time using 16 threads (same as the number of physical cores on the AWS instance). The preprocessing of CQ is accelerated by almost $16\times$. Though our implementation is not specifically optimized for multi-threading, the running time is improved by 6.1 - $7.7\times$, which is still significantly faster than CQ.

Lookup Time. We then compare both schemes in terms of lookup time. Figure 3 (left) presents the time needed to generate lookup proofs for varying table sizes ranging between $n = 2^{16}$ - 2^{26} and $k \in \{16, 64, 256, 1024\}$ lookups while Figure 3 (right) shows the time lookup time of both schemes for tables of size $n \in \{2^{18}, 2^{24}\}$ and varying lookup sizes $k = 2^{16}$ - 2^{11} . For example, for a table size of $n = 2^{20}$ and lookup size $k = 1024$, the CQ prover takes 5.69 seconds while our scheme only takes 0.34 second, which is $16.7\times$ faster. This stems from the fact that while our prover has higher asymptotic complexity, it only relies on field and hash operations—making it significantly faster in practice than approaches bound by group operations. That said, for large enough table sizes, our prover time will eventually be slower than CQ. For example, the break-even point is around $n = 2^{24}$ for $k = 64$. Finally, observe that our scheme scales better with the increasing number of lookups. This agrees with our asymptotic analysis since, for large values of k , our $\mathcal{O}(\sqrt{kn})$ complexity surpasses the $\mathcal{O}(k \log k)$ of CQ.

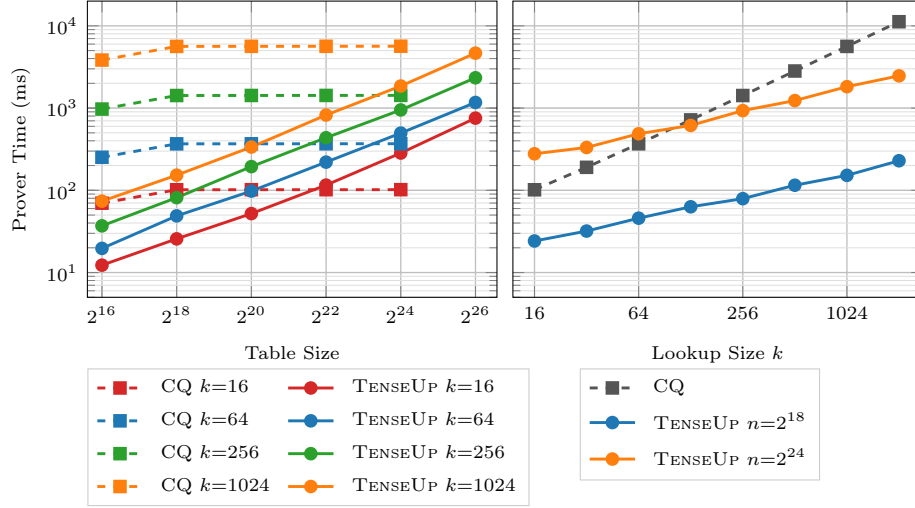


Figure 3: Online prover time for CQ and TENSEUP. Left: Increasing table size for $k \in \{16, 64, 256, 1024\}$ lookups. Right: Increasing number of lookups for tables of size $n \in \{2^{18}, 2^{24}\}$; CQ time remains almost identical for both values of n .

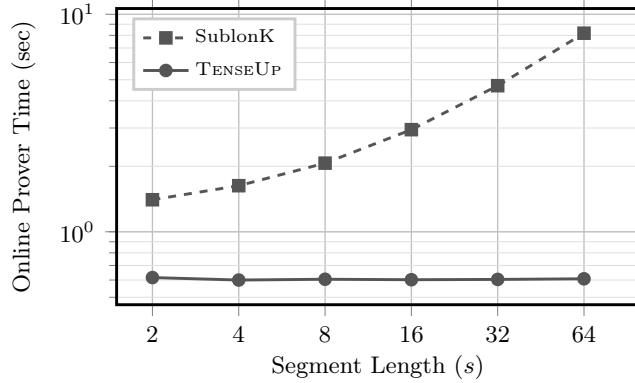


Figure 4: Online prover time for $n = 2^{24}$, $K = 128$, and s segments. The total number of looked-up elements is $K \times s = 128s$.

Proof Size and Verifier Time. As expected, the only drawback of our scheme is its much larger proof size ranging from 360-1073KB compared to that of CQ which remains constant and equal to 1.125KB. Finally, both schemes achieve very fast verification times smaller than 35ms.

Experimenting with Matrix Lookups. Doubly-efficient matrix lookup arguments [34, 24, 63] extend standard lookup arguments ones by enabling the prover to commit to a matrix and subsequently prove lookups for entire columns of the matrix in time sublinear to the matrix size. Recently, matrix lookups have been widely used to construct efficient verifiable indexes [63] and decision trees [64, 24], as well as to efficiently prove the correct computation of circuits with conditional

branches [34]. Interestingly TENSEUP is inherently a matrix lookup in the sense that performing various lookups over the same column of the matrix $\mathbf{X}$ *incurs negligible prover overhead*. At a very high-level, this stems from the fact that the prover already encodes $\mathbf{x}$ as a matrix $\mathbf{X}$ in order to compute its codeword, and during the reduction phase our construction in Section 4.2, it has to commit to the entire column of $\mathbf{X}$ even if a single element is accessed at that column.

Here we further test the performance of our scheme when proving matrix lookups and compare it with Sublonk [34], a specialized matrix lookup based on CQ. Figure 4 shows the time needed to generate matrix lookup proofs for $K = 128$ columns, total matrix size of $n = 2^{24}$ and column sizes ranging from $s = 2 \cdot 2^6$. As expected, increasing segment size has negligible impact to our prover time. By contrast, the prover time of SublonK increases with s . Note that SublonK is already optimized for segment lookups and the performance is better than running CQ on $k = K \cdot s$ elements. Our speedup grows from $2.3\times$ for $s = 4$ to $13.5\times$ for $s = 64$.

Acknowledgments

This work is partially supported by the National Science Foundation (NSF) under Grant No. 2401481. Dimitrios Papadopoulos was supported by the Hong Kong Research Grants Council under grant GRF-16213625. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these institutes.

References

- [1] <https://anonymous.4open.science/r/Code-Based-Lookup-Arguments-850D/README.md>
- [2] <https://github.com/txaty/ark-segmentlookup>
- [3] Albrecht, M., Grassi, L., Rechberger, C., Roy, A., Tiessen, T.: Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 191–219. Springer (2016)
- [4] Arnon, G., Chiesa, A., Fenzi, G., Yogev, E.: WHIR: reed-solomon proximity testing with super-fast verification. In: EUROCRYPT. pp. 214–243 (2025)
- [5] Arun, A., Setty, S.T.V., Thaler, J.: Jolt: Snarks for virtual machines via lookups. In: Joye, M., Leander, G. (eds.) Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part VI. Lecture Notes in Computer Science, vol. 14656, pp. 3–33. Springer (2024). https://doi.org/10.1007/978-3-031-58751-1_1, https://doi.org/10.1007/978-3-031-58751-1_1
- [6] Babbush, R., Zalcman, A., Gidney, C., Broughton, M., Khattar, T., Neven, H., Bergamaschi, T., Drake, J., Boneh, D.: Securing elliptic curve cryptocurrencies against quantum vulnerabilities: Resource estimates and mitigations (Mar 2026), <https://quantumai.google/static/site-assets/downloads/cryptocurrency-whitepaper.pdf>, google Quantum AI Whitepaper

- [7] Batista, M., Montes, Á., Paslis, N., Ràfols, C.: Dynasaurs: Efficient universal dynamic zk-snarks from sparse linear arguments. *Cryptology ePrint Archive* (2026)
- [8] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: Snarks for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*, Santa Barbara, CA, USA, August 18-22, 2013. *Proceedings, Part II. Lecture Notes in Computer Science*, vol. 8043, pp. 90–108. Springer (2013). https://doi.org/10.1007/978-3-642-40084-1_6, https://doi.org/10.1007/978-3-642-40084-1_6
- [9] Ben-Sasson, E., Chiesa, A., Riabzev, M., Spooner, N., Virza, M., Ward, N.P.: Aurora: Transparent succinct arguments for R1CS. In: *EUROCRYPT*. pp. 103–128 (2019)
- [10] Ben-Sasson, E., Chiesa, A., Spooner, N.: Interactive oracle proofs. In: *TCC*. pp. 31–60 (2016)
- [11] Ben-Sasson, E., Goldreich, O., Harsha, P., Sudan, M., Vadhan, S.P.: Short pcps verifiable in polylogarithmic time. In: *20th Annual IEEE Conference on Computational Complexity (CCC 2005)*, 11-15 June 2005, San Jose, CA, USA. pp. 120–134. IEEE Computer Society (2005). <https://doi.org/10.1109/CCC.2005.27>, <https://doi.org/10.1109/CCC.2005.27>
- [12] Ben-Sasson, E., Goldreich, O., Harsha, P., Sudan, M., Vadhan, S.P.: Robust pcps of proximity, shorter pcps, and applications to coding. *SIAM J. Comput.* **36**(4), 889–974 (2006). <https://doi.org/10.1137/S0097539705446810>, <https://doi.org/10.1137/S0097539705446810>
- [13] Bentley, J.L., Saxe, J.B.: Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms* **1**(4), 301–358 (1980). [https://doi.org/10.1016/0196-6774\(80\)90015-2](https://doi.org/10.1016/0196-6774(80)90015-2), [https://doi.org/10.1016/0196-6774\(80\)90015-2](https://doi.org/10.1016/0196-6774(80)90015-2)
- [14] Bhargava, V., Ghosh, S., Guo, Z., Kumar, M., Umans, C.: Fast multivariate multipoint evaluation over all finite fields. In: *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*. pp. 221–232. IEEE (2022). <https://doi.org/10.1109/FOCS54457.2022.00028>, <https://doi.org/10.1109/FOCS54457.2022.00028>
- [15] Block, A.R., Garreta, A., Katz, J., Thaler, J., Tiwari, P.R., Zajac, M.: Fiat-shamir security of FRI and related snarks. In: *ASIACRYPT*. pp. 3–40 (2023)
- [16] Blocki, J., Zhang, J.: Amortized locally decodable codes. In: *2025 IEEE International Symposium on Information Theory (ISIT)*. pp. 1–6. IEEE (2025)
- [17] Bootle, J., Guskind, J., Patrabis, S., Sotiraki, K.: Lookup arguments over rings and applications to batch-verification of RAM programs. *IACR Cryptol. ePrint Arch.* **2026**, 471 (2026), <https://eprint.iacr.org/2026/471>
- [18] Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. pp. 341–357. SOSP '13, ACM, New York, NY, USA (2013). <https://doi.org/10.1145/2517349.2522733>

- [19] Bünz, B., Fenzi, G., Rothblum, R.D., Wang, W.: Tensorswitch: Nearly optimal polynomial commitments from tensor codes. In: Heninger, N., Rosulek, M. (eds.) *Advances in Cryptology - CRYPTO 2026 - 46th Annual International Cryptology Conference*, Santa Barbara, CA, USA, August 17-20, 2026, *Proceedings, Part IX*. *Lecture Notes in Computer Science*, vol. 16808, pp. 297–329. Springer (2026). https://doi.org/10.1007/978-3-032-35424-2_11, https://doi.org/10.1007/978-3-032-35424-2_11
- [20] Bünz, B., Fisch, B., Szeponiec, A.: Transparent snarks from dark compilers. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. pp. 677–706. Springer (2020)
- [21] Cain, M., Xu, Q., King, R., Picard, L.R., Levine, H., Endres, M., Preskill, J., Huang, H.Y., Bluvstein, D.: Shor’s algorithm is possible with as few as 10,000 reconfigurable atomic qubits. *arXiv preprint arXiv:2603.28627* (2026)
- [22] Camenisch, J., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: *Annual international cryptology conference*. pp. 61–76. Springer (2002)
- [23] Campanelli, M., Carrillo, M., Cascudo, I., Fiore, D., Francati, D., Gennaro, R.: On the power of polynomial preprocessing: Proving computations in sublinear time, and more. *Cryptology ePrint Archive* (2025)
- [24] Campanelli, M., Faonio, A., Fiore, D., Li, T., Lipmaa, H.: Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. In: *PKC*. pp. 337–369 (2024)
- [25] Campanelli, M., Fiore, D., Gennaro, R.: Natively compatible super-efficient lookup arguments and how to apply them. *J. Cryptol.* p. 14 (2025)
- [26] Campanelli, M., Fiore, D., Han, S., Kim, J., Kolonelos, D., Oh, H.: Succinct zero-knowledge batch proofs for set accumulators. In: *ACM CCS*. pp. 455–469 (2022)
- [27] Canetti, R., Chen, Y., Holmgren, J., Lombardi, A., Rothblum, G.N., Rothblum, R.D., Wichs, D.: Fiat-shamir: from practice to theory. In: *ACM STOC*. pp. 1082–1090 (2019)
- [28] Carrillo Redondo, M.: Efficient polynomial evaluations with preprocessing and applications to cryptography (2024)
- [29] Catalano, D., Fiore, D.: Vector commitments and their applications. In: *International Workshop on Public Key Cryptography*. pp. 55–72. Springer (2013)
- [30] Chandran, N., Kanukurthi, B., Ostrovsky, R.: Locally updatable and locally decodable codes. In: *Theory of Cryptography Conference*. pp. 489–514. Springer (2014)
- [31] Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, P., Ward, N.P.: Marlin: Preprocessing zk-snarks with universal and updatable SRS. In: *EUROCRYPT*. pp. 738–768 (2020)
- [32] Chiesa, A., Manohar, P., Spooner, N.: Succinct arguments in the quantum random oracle model. In: *TCC*. pp. 1–29 (2019)
- [33] Chiesa, A., Ojha, D., Spooner, N.: Fractal: Post-quantum and transparent recursive proofs from holography. In: *EUROCRYPT*. pp. 769–793 (2020)

- [34] Choudhuri, A.R., Garg, S., Goel, A., Sekar, S., Sinha, R.: Sublonk: Sublinear prover plonk. PETs pp. 314–335 (2024)
- [35] Demertzis, I., Chamani, J.G., Papadopoulos, D., Papamanthou, C.: Dynamic searchable encryption with small client storage. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020. The Internet Society (2020), <https://www.ndss-symposium.org/ndss-paper/dynamic-searchable-encryption-with-small-client-storage/>
- [36] Dutta, M., Ganesh, C., Patranabis, S., Prakash, S., Singh, N.: Batching-efficient RAM using updatable lookup arguments. In: CCS. pp. 4077–4091. ACM (2024)
- [37] Dworkin, M.J.: Sha-3 standard: Permutation-based hash and extendable-output functions (2015)
- [38] Eagen, L., Fiore, D., Gabizon, A.: cq: Cached quotients for fast lookups. IACR Cryptol. ePrint Arch. p. 1763 (2022)
- [39] Gabizon, A., Williamson, Z.J.: plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive (2020)
- [40] Goldreich, O.: On the lower bound on the length of relaxed locally decodable codes. Electron. Colloquium Comput. Complex. **TR23**, TR23-064 (2023), <https://eccc.weizmann.ac.il/report/2023/064>
- [41] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) **43**(3), 431–473 (1996)
- [42] Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. In: STOC. pp. 113–122. ACM (2008)
- [43] Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schafnegg, M.: Poseidon: A new hash function for {Zero-Knowledge} proof systems. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 519–535 (2021)
- [44] Gu, B., Fang, J., Nawab, F.: Poneglyphdb: Efficient non-interactive zero-knowledge proofs for arbitrary sql-query verification. Proc. ACM Manag. Data pp. 63:1–63:27 (2025)
- [45] Gur, T., Ramnarayan, G., Rothblum, R.: Relaxed locally correctable codes. Theory Comput. **16**, 1–68 (2020). <https://doi.org/10.4086/TOC.2020.V016A018>, <https://doi.org/10.4086/toc.2020.v016a018>
- [46] Haböck, U.: Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Paper 2022/1530 (2022), <https://iacr.org>
- [47] Holmgren, J.: On round-by-round soundness and state restoration attacks. IACR Cryptol. ePrint Arch. (2019)
- [48] Ishai, Y., Ostrovsky, R., Shah, A.: Succinct arguments for RAM programs via projection codes. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20–24, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 14082, pp. 159–192. Springer (2023). https://doi.org/10.1007/978-3-031-38545-2_6, https://doi.org/10.1007/978-3-031-38545-2_6

- [49] Ishai, Y., Ostrovsky, R., Shah, A.: Zero-knowledge RAM: doubly efficient and black-box. In: Fehr, S., Fouque, P. (eds.) *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 15604, pp. 363–395. Springer (2025). https://doi.org/10.1007/978-3-031-91134-7_13, https://doi.org/10.1007/978-3-031-91134-7_13
- [50] Jiang, Y., Yang, X., Liu, X., Guo, Y., Zhang, J.: zkrag: Efficiently proving rag retrieval in zero knowledge. *Cryptology ePrint Archive* (2026)
- [51] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: *ASIACRYPT*. pp. 177–194 (2010)
- [52] Katz, J., Trevisan, L.: On the efficiency of local decoding procedures for error-correcting codes. In: Yao, F.F., Luks, E.M. (eds.) *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, May 21-23, 2000, Portland, OR, USA. pp. 80–86. ACM (2000). <https://doi.org/10.1145/335305.335315>, <https://doi.org/10.1145/335305.335315>
- [53] Kilian, J.: A note on efficient zero-knowledge proofs and arguments. In: *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. pp. 723–732 (1992)
- [54] Kopparty, S., Meir, O., Ron-Zewi, N., Saraf, S.: High-rate locally correctable and locally testable codes with sub-polynomial query complexity. *J. ACM* **64**(2), 11:1–11:42 (2017). <https://doi.org/10.1145/3051093>, <https://doi.org/10.1145/3051093>
- [55] Liu, T., Xie, X., Zhang, Y.: zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In: *CCS*. pp. 2968–2985 (2021)
- [56] Merkle, R.C.: A digital signature based on a conventional encryption function. In: *Conference on the theory and application of cryptographic techniques*. pp. 369–378. Springer (1987)
- [57] Namboothiry, B., Shirzad, A., Solit, S., Marcus, R., Mishra, P.: Truthtable: A verifiable query engine. *IACR Cryptol. ePrint Arch.* **2026**, 1260 (2026), <https://eprint.iacr.org/2026/1260>
- [58] O’Connor, J., Aumasson, J.P., Neves, S., Wilcox-O’Hearn: BLAKE3: one function, fast everywhere. <https://github.com/BLAKE3-team/BLAKE3> (2020)
- [59] Overmars, M.H.: *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, vol. 156. Springer (1983). <https://doi.org/10.1007/BFb0014927>, <https://doi.org/10.1007/BFb0014927>
- [60] Overmars, M.H., van Leeuwen, J.: Worst-case optimal insertion and deletion methods for decomposable searching problems. *Inf. Process. Lett.* **12**(4), 168–173 (1981). [https://doi.org/10.1016/0020-0190\(81\)90093-4](https://doi.org/10.1016/0020-0190(81)90093-4), [https://doi.org/10.1016/0020-0190\(81\)90093-4](https://doi.org/10.1016/0020-0190(81)90093-4)
- [61] Ozdemir, A., Wahby, R., Whitehat, B., Boneh, D.: Scaling verifiable computation using efficient set accumulators. In: *29th USENIX Security Symposium (USENIX Security 20)*. pp. 2075–2092 (2020)

- [62] Papamanthou, C., Shi, E., Tamassia, R.: Signatures of correct computation. In: TCC. pp. 222–242 (2013)
- [63] Pappas, C., Cai, Z., Papadopoulos, D.: Rogue: Updatable matrix lookup arguments and applications to verifiable databases. Cryptology ePrint Archive (2026)
- [64] Pappas, C., Papadopoulos, D.: Sparrow: Space-efficient zkSnark for data-parallel circuits and applications to zero-knowledge decision trees. In: ACM CCS. pp. 3110–3124 (2024)
- [65] Pappas, C., Papadopoulos, D., Papamanthou, C.: Hydraproofs: Optimally computing all proofs in a vector commitment (with applications to efficient zkSnarks over data from multiple users). In: Blanton, M., Enck, W., Nita-Rotaru, C. (eds.) IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12–15, 2025. pp. 3421–3439. IEEE (2025). <https://doi.org/10.1109/SP61157.2025.00204>, <https://doi.org/10.1109/SP61157.2025.00204>
- [66] Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the society for industrial and applied mathematics pp. 300–304 (1960)
- [67] Ron-Zewi, N., Rothblum, R.: Local proofs approaching the witness length. Journal of the ACM **71**(3), 1–42 (2024)
- [68] Scroll architecture. <https://docs.scroll.io/en/technology/> (2023)
- [69] Setty, S., Thaler, J., Wahby, R.: Unlocking the lookup singularity with lasso. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 180–209. Springer (2024)
- [70] Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM review **41**(2), 303–332 (1999)
- [71] Umans, C.: Fast polynomial factorization and modular composition in small characteristic. In: Dwork, C. (ed.) Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17–20, 2008. pp. 481–490. ACM (2008). <https://doi.org/10.1145/1374376.1374445>, <https://doi.org/10.1145/1374376.1374445>
- [72] Wang, W., Papamanthou, C., Srinivasan, S., Papadopoulos, D.: Designated-verifier dynamic zk-snarks with applications to dynamic proofs of index. Cryptology ePrint Archive (2026)
- [73] Wang, W., Papamanthou, C., Srinivasan, S., Papadopoulos, D.: Dynamic zk-snarks (with applications to sparse zk-snarks and ivc). In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 329–358. Springer (2026)
- [74] Xie, T., Zhang, Y., Song, D.: Orion: Zero knowledge proof with linear prover time. In: Dodis, Y., Shrimpton, T. (eds.) Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 13510, pp. 299–328. Springer (2022). https://doi.org/10.1007/978-3-031-15985-5_11, https://doi.org/10.1007/978-3-031-15985-5_11

- [75] Xin, J., Tin, S.C.O., Pappas, C., Huang, Y., Papadopoulos, D.: Gryphes: Hybrid proofs for modular snarks with applications to zkrollups. *Proc. Priv. Enhancing Technol.* **2026**(2), 198–214 (2026). <https://doi.org/10.56553/POPETS-2026-0044>, <https://doi.org/10.56553/popets-2026-0044>
- [76] Zapico, A., Buterin, V., Khovratovich, D., Maller, M., Nitulescu, A., Simkin, M.: Caulk: Lookup arguments in sublinear time. In: *CCS*. pp. 3121–3134 (2022)
- [77] Zeilberger, H., Chen, B., Fisch, B.: Basefold: efficient field-agnostic polynomial commitment schemes from foldable codes. In: *Annual International Cryptology Conference*. pp. 138–169. Springer (2024)
- [78] Zhang, J., Fang, Z., Zhang, Y., Song, D.: Zero knowledge proofs for decision tree predictions and accuracy. In: *ACM CCS*. pp. 2039–2053 (2020)
- [79] Zhang, J., Xie, T., Hoang, T., Shi, E., Zhang, Y.: Polynomial commitment with a one-to-many prover and applications. In: Butler, K.R.B., Thomas, K. (eds.) *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. pp. 2965–2982. USENIX Association (2022), <https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-jiaheng>
- [80] Zhang, J., Xie, T., Zhang, Y., Song, D.: Transparent polynomial delegation and its applications to zero knowledge proof. In: *IEEE SP*. pp. 859–876 (2020)
- [81] Zhang, T., Ouyang, Y., Zhang, Y.: Dynark: Making groth16 dynamic. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. pp. 178–208. Springer (2026)
- [82] Zhang, Y., Sun, S., Gu, D.: Efficient kzg-based univariate sum-check and lookup argument. In: *PKC* (2). pp. 400–425. Springer (2024)
- [83] Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: A zero-knowledge version of vsql. *IACR Cryptol. ePrint Arch.* (2017)
- [84] Zhang, Y., Genkin, D., Katz, J., Papadopoulos, D., Papamanthou, C.: vram: Faster verifiable RAM with program-independent preprocessing. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. pp. 908–925. IEEE Computer Society (2018). <https://doi.org/10.1109/SP.2018.00013>, <https://doi.org/10.1109/SP.2018.00013>

Appendix

A Additional Details on Our Code-Based Lookup Arguments

A.1 Proof Of Theorem 4

Performance Analysis. Regarding the prover time, observe that step 1 of the reduction phase has $\mathcal{O}(k)$ complexity, steps 2 and 3 have $\tilde{\mathcal{O}}(k\sqrt{n/k}) = \tilde{\mathcal{O}}(\sqrt{kn})$ and step 4 has a complexity of

$\mathcal{O}(\text{Pt}_{\text{IOP}}(\sqrt{kn}))$. Next, the step 1 of the lookup phase requires $l_1\sqrt{kn}$ field operations while steps 2 and 3 have a proving complexity of $\tilde{\mathcal{O}}(\sqrt{kn})$ and $\mathcal{O}(\text{Pt}_{\text{IOP}}(\sqrt{kn}))$. So the total prover time will be $\mathcal{O}(\text{Pt}_{\text{IOP}}(\sqrt{kn}) + \lambda\sqrt{kn})$.

As for the proof size, observe that in both phases the prover only sends to the verifier a constant number of oracles and produces two IOP proofs. In addition, the verifier does l_1l_2 queries over $\mathbf{C}$. Hence, the total proof size will be $\mathcal{O}(\text{Ps}_{\text{IOP}}(\sqrt{kn}) + l_1l_2)$. Likewise we argue about the verification time.

Completeness Proof. Here, we will show that **Construction 2** is an IOP with perfect completeness (over $\mathcal{R}_{\text{IOP}, \mathbf{C}, 1}$). In particular, we will show that provided a valid lookup pair $(\mathcal{I}, \mathbf{t})$, the verifier will always accept. Since $(\mathcal{I}, \mathbf{t})$ is valid, it must hold that $\mathbf{R}[:, j] = \mathbf{X}_m[:, i_j]$ and thus $\mathbf{t}' = \sum s^k \mathbf{R}'[q_k, :] = \sum s^k \mathbf{X}'_m[q_k, \mathcal{I}^{(2)}] = \mathbf{x}'[\mathcal{I}^{(2)}]$. So both $\mathcal{R}'_{\text{Red}}$ and $\mathcal{R}'_{\text{Mem}}$ must be satisfied and thus the verifier will pass the IOP proofs with probability 1. Finally, since $\mathbf{x}'$ is honestly computed, it must hold that $\text{Enc}(\mathbf{x}')[q'] = \text{Enc}(\sum_k s^k \mathbf{X}'_m[q_k, :])[q'] = \sum_k s^k \text{Enc}(\mathbf{X}'_m[q_k, :])[q'] = \sum_k s^k \mathbf{C}[q_k, q']$ and thus the point query tests pass with probability 1.

Round-by-Round (Knowledge) Soundness Error. Similarly to the proof of Theorem 3, we will prove that **Construction 2** is an IOP with round-by-round (knowledge) soundness error of $\epsilon_{\text{rbr}} = \kappa_{\text{rbr}} = \max(\epsilon_{\text{IOP}}, (1 - \delta_{\mathcal{C}_1^{(m)}}/2)^{l_1}, (1 - \delta_{\mathcal{C}_2^{(m)}}/2)^{l_2}, \epsilon_{\text{prox}})$. By our initial assumption, it holds that $\Delta(\mathbf{C}^{(i)}, \mathcal{C}^{(i)}) < \delta_{\mathcal{C}_1^{(i)}}\delta_{\mathcal{C}_2^{(i)}}/4$ and we denote with $\mathbf{C}^{(i)*} \in \mathcal{C}^{(i)}$, the unique codeword such that $\Delta(\mathbf{C}^{(i)}, \mathbf{C}^{(i)*}) < \delta_{\mathcal{C}_1^{(i)}}\delta_{\mathcal{C}_2^{(i)}}/4$. Before diving into the details of our proof, we first present the following Lemma:

Lemma 1 *The maximum fraction of rows of $\mathbf{C}^{(i)}$ satisfying $\Delta(\mathbf{C}^{(i)}, \mathbf{C}^{(i)*}) \geq \delta_{\mathcal{C}_2^{(i)}}/2$ is $\delta_{\mathcal{C}_1^{(i)}}/2$.*

Assume that more than $\delta_{\mathcal{C}_1^{(i)}}/2$ rows are $\delta_{\mathcal{C}_2^{(i)}}/2$ -far from the corresponding rows of $\mathbf{C}^*$. Then the number of errors of $\mathbf{C}$ would be more than $\delta_{\mathcal{C}}/4$ which contradicts our assumption. We denote with $\mathcal{J}$ the set of indexes of the “good” rows of $\mathbf{C}$, namely those that are $\delta_{\mathcal{C}_2^{(i)}}/2$ -close to those of $\mathbf{C}^*$.

Having established that, assume that the initial transcript $\mathbf{tr}_0 = ([\mathcal{I}], [\mathbf{t}], [\mathbf{C}^{(i)}])$ is in a doomed state $\mathcal{D}$. We will show that for each round between $\mathcal{P}$ and $\mathcal{V}$, our protocol leaves the doomed state with negligible probability.

Reduction Phase:

1. Conditioned that $\mathbf{tr}_0 \in \mathcal{D}$, we will show that $\Pr[\mathbf{tr}_1 \notin \mathcal{D}] < (1 - \delta_{\mathcal{C}_1^{(m)}}/2)^{l_1} + l_1/|\mathbb{F}|$ where $\mathbf{tr}_1 = (\mathbf{tr}_0, [\mathbf{R}], (\mathcal{Q}, s), [\mathbf{t}'])$. Note that $\mathbf{tr}_1$ leaves the doomed state if and only if (1) $(\mathbf{t}', \mathbf{R}, \mathcal{I}, \mathcal{Q}, \mathbf{t}, s)$ satisfy $\mathcal{R}'_{\text{Red}}$ and (2) either (i) $\mathbf{t}' = \sum_{k \in [l_1]} s^k \mathbf{X}'_m[q_k, \mathcal{I}^{(2)}]$ or (ii) $\mathbf{R}'[q_k, :] = \mathbf{X}'_m[q_k, \mathcal{I}^{(2)}]$ for all $q_k \in \mathcal{J}$. Note that since $\mathbf{tr}_0 \in \mathcal{D}$, and because $\mathcal{R}'_{\text{Red}}$ must be satisfied, it must hold that $\mathbf{R} \neq \mathbf{X}[:, \mathcal{I}^{(2)}]$. We will use this, to bound the probability that condition (2) is satisfied by first focusing on the event (2.ii). Note that the probability of (2.ii) happening is at most $(1 - p)^{l_1}$ where p the probability that $\mathbf{R}'[q_k, :] \neq \mathbf{X}'[q_k, \mathcal{I}^{(2)}]$ and $q_k \in \mathcal{J}$. So what remains, is to find p . For this, observe that since $\mathbf{R} \neq \mathbf{X}[:, \mathcal{I}^{(2)}]$, there must exist a column $c_j \in \mathcal{I}^{(2)}$ such that $\mathbf{R}[:, j] \neq \mathbf{X}[:, c_j]$ and consequently $\Delta(\mathbf{R}'[:, j], \mathbf{X}'[:, c_j]) > \delta_{\mathcal{C}_1^{(m)}}$. Let $\mathcal{J}'$ be the set of indexes where $\mathbf{X}'[:, c_j]$ differs to $\mathbf{R}'[:, j]$, then the probability a random query q_k satisfies the above condition would be $p = \frac{|\mathcal{J}' \setminus \mathcal{J}|}{n_1} > \frac{|\mathcal{J}'| - |\mathcal{J}|}{n_1} > \delta_{\mathcal{C}_1^{(m)}} - \delta_{\mathcal{C}_1^{(m)}}/2 = \delta_{\mathcal{C}_1^{(m)}}/2$. So condition

(2.ii) happens with probability at most $(1 - \delta_{\mathcal{C}_1^{(m)}}/2)^{l_1} < \text{negl}(\lambda)$. Finally, note that if (2.ii) does not hold, then, (2.i) happens with probability $\frac{l_1}{|\mathbb{F}|}$ due to the Schwartz-Zippel Lemma.

2. Conditioned that $\mathbf{tr}_1 \in \mathcal{D}$, we will show that $\Pr[\mathbf{tr}_2 \notin \mathcal{D}] < \epsilon_{\text{IOP}}$ where $(\mathbf{tr}_1, \mathbf{tr}_{\text{IOP,Red}'})$ and $\mathbf{tr}_{\text{IOP,Red}'}$ the communication transcript of the IOP proof for $\mathcal{R}'_{\text{Red}}$. Note that since $\mathbf{tr}_1 \in \mathcal{D}$, then either (1) or (2) do not hold. If (1) does not hold then $\mathcal{P}$ leaves the doomed state with probability ϵ_{IOP} due to the round-by-round soundness of the underlying IOP. Otherwise, $\mathcal{P}$ stays in a doomed state with probability 1.

Lookup Phase:

1. Conditioned that $\mathbf{tr}_2 \in \mathcal{D}$, we will show that $\Pr[\mathbf{tr}_3 \notin \mathcal{D}] < (1 - \delta_{\mathcal{C}_2^{(i)}}/2)^{l_2} + \epsilon_{\text{prox}}$, where $\mathbf{tr}_3 = (\mathbf{tr}_2, ([\mathbf{x}', \mathcal{Q}', \mathbf{q}])$. Note that $\mathbf{tr}_3$ leaves the doomed state if (1) $\mathcal{R}'_{\text{Mem}}$ is satisfied and (2) $\mathbf{q}[j] = \sum_{k \in [l_1]} s^k \mathbf{C}^{(m)}[q_k, q'_j]$ for all $q'_j \in \mathcal{Q}'$. Assuming that (1) is satisfied, we will bound the probability of (2) happening. For this, let $\mathbf{c}' = \sum_{k \in [l_1]} s^k \mathbf{C}[q_k, :]$. If $\mathbf{c}'$ is $\delta_{\mathcal{C}_2^{(m)}}/2$ -far from any code in $\mathcal{C}^{(m)}$, then, the probability that (2) holds is at most $(1 - \delta_{\mathcal{C}_2^{(m)}}/2)^{l_2}$. So, we focus on the case that $\mathbf{c}'$ is $\delta_{\mathcal{C}_2^{(m)}}/2$ -close to a codeword in $\mathcal{C}_2^m$. In this case, we will prove that $\mathbf{c}'$ is at least $\delta_{\mathcal{C}_2^{(i)}}/2$ -far from $\text{Enc}_{\mathcal{C}_2^{(i)}}(\mathbf{x}')$ which ensures that (2) happens with probability $(1 - \delta_{\mathcal{C}_2^{(i)}}/2)^{l_2}$. In particular, because of the proximity-gap property of $\mathcal{C}^{(2)}$ it must hold that all $\mathbf{C}[q_k, :]$ are $\delta_{\mathcal{C}_2^{(i)}}/2$ -close with probability ϵ_{prox} . Let $\mathbf{Z} \in \mathbb{F}^{l_1 \times \sqrt{n^{2i}}}$ be the matrix such that $\mathbf{Z}[k] = \text{Dec}_{\mathcal{C}_2^{(i)}}(\mathbf{C}[q_k, :])$ and let $\mathbf{c}^* = \text{Enc}_{\mathcal{C}_2^{(i)}}(\mathbf{z}')$ where $\mathbf{z}' = \sum_{k \in [l_1]} s^k \mathbf{Z}[k, :]$. Since $\Delta(\mathbf{c}', \mathbf{c}^*) < \delta_{\mathcal{C}_2^{(i)}}/2$, we simply need to show that $\mathbf{c}^* \neq \text{Enc}_{\mathcal{C}_2^{(i)}}(\mathbf{x}')$. This comes directly from the fact that $\mathbf{tr}_2 \in \mathcal{D}$ and so there exists at least one k where $\mathbf{X}'[q_k, \mathcal{I}] = \mathbf{Z}[k, \mathcal{I}^{(2)}] \neq \mathbf{R}[q_k, :]$ and so $\sum_{i \in [l_1]} s^i \mathbf{Z}[i, :] \neq \mathbf{t}'$ which implies that $\mathbf{z}' \neq \mathbf{x}'$.
2. Finally, conditioned that $\mathbf{tr}_3 \in \mathcal{D}$, we show that $\Pr[\mathbf{tr}_4 \notin \mathcal{D}] < \epsilon_{\text{IOP}}$ where $\mathbf{tr}_4 = (\mathbf{tr}_3, \mathbf{tr}_{\text{IOP,Mem}})$. Note that since $\mathbf{tr}_3 \in \mathcal{D}$ it implies that $\mathcal{R}_{\text{Mem}}$ is not satisfied, so, due to the round-by-round soundness of the IOP $\mathcal{P}$ will leave the doomed state only with probability ϵ_{IOP} .

A.2 Proof of Theorem 5

Performance Analysis. Regarding the prover time, observe that in the commitment algorithm the prover only invokes the encoding algorithm of $\mathcal{C}$ and the IOPP prover for $\mathcal{R}_{\text{prox}}$ thus leading to $\text{Pt}_{\text{Com}}(n) \in \mathcal{O}(\text{Pt}_{\text{Enc}}(n) + \text{Pt}_{\text{IOPP, lkp}}(n))$. As for the lookup time, the prover simply invokes the IOP prover for $\mathcal{R}_{\text{lkp}}$. Finally, the proof size corresponds to the size of the commitment and the size of the lookup proof. The first one includes the commitment of $\mathbf{C}$ (which has a constant size) and the proximity proof obtained by the BCS transformation hence having size of $\mathcal{O}(\log n \cdot \text{Ps}_{\text{IOPP, lkp}}(n))$ while the latter will have $\mathcal{O}(\log n \cdot \text{Ps}_{\text{IOP, lkp}}(k, n))$ after applying the BCS transformation. Likewise, we argue about the verification time.

Completeness. Completeness of **Construction 3** comes directly from the completeness of the non-interactive variants of the IOPP for $\mathcal{R}_{\text{prox}}$ and the IOP for $\mathcal{R}_{\text{lkp}}$.

Knowledge Soundness. To prove knowledge soundness, we first construct an extractor that is given $C, \pi_{\text{lkp}}, C_{\mathcal{I}}, C_{\mathbf{t}}$ (containing accepting proofs) and internal access to $\mathcal{P}$ and obtains the table

$\mathbf{x}$ as well as $\mathcal{I}, \mathbf{t}$ such that $\mathcal{R}_{\text{lkp}}$ is satisfied with probability at least $1 - \text{negl}(\lambda)$. If $\delta = 0$, our extractor will run the extractor of the non-interactive prover derived from the IOP prover for $\mathcal{R}_{\mathbf{C},i}$ (for each queried i) in order to obtain $\mathbf{x}$. Otherwise, our extractor, will first run the extractor of the non-interactive prover derived from the IOPP prover for $\mathcal{R}_{\text{prox}}$ in order to obtain $\mathbf{x}$ and then it invokes the extractor for the non-interactive prover derived from the IOP prover for $\mathcal{R}_{\text{prox}}$ to obtain $\mathbf{t}$ and $\mathcal{I}$. By the round-by-round knowledge soundness of the underlying IOPs, Lemma 2 and from the fact that $\delta < \delta_{\mathbf{C}}/2$, it must hold that $\Delta(\mathbf{C}, \text{Enc}_{\mathbf{C}}(\mathbf{x})) < \delta$ with probability of at least $1 - \text{negl}(\lambda)$. Likewise, we argue in order to show that $\mathbf{t}, \mathcal{I}$ and $\mathbf{x}$ must satisfy $\mathcal{R}_{\text{lkp}}$ with probability $> 1 - \text{negl}(\lambda)$ which concludes our proof.

B Proof of Theorem 6

Performance Analysis. First, we focus on measuring the amortized prover cost of the update algorithm and later move on analyzing the lookup prover cost. Finally, measure proof size and verification time.

Update Complexity. First, we focus on update prover time. To simplify the analysis, we consider the worst case scenario in which all updates have different indexes. In this way, the merging operations done at the i -th level of $\mathcal{D}_{\mathcal{P}}$ will always increase the size of the new update requiring it to move to the $(i+1)$ -th level. With that in mind, we first analyze the prover time assuming that all updates have the same size k (e.g., the algorithm only touches steps 1 or 2). For this, we simply have to measure the total (end-to-end) prover time needed for n/k update invocations. First observe that the prover has to run the lookup commitment algorithm for a $2^i k$ -sized table $n/(2^i k)$ times. Likewise, a similar observation holds for the SNARK prover for $\mathcal{R}_{\text{merge}}$ and $\mathcal{R}_{\text{sort}}$. Since we assume that the indexes are always sorted, both merge and sort operations have $\mathcal{O}(k2^i)$ complexity. Consequently, the total prover time over n/k invocations is $\mathcal{O}(\sum_{i \in [\log(n/k)]} \frac{n}{k2^{i-1}} (\text{Pt}_{\text{com}}(k2^{i-1}) + \text{Pt}(k2^{i-1})))$ and the amortized cost per update will be $\mathcal{O}((\sum_{i \in [\log(n/k)]} \frac{n}{k2^{i-1}} (\text{Pt}_{\text{com}}(k2^{i-1}) + \text{Pt}(k2^{i-1}))) / n)^4$.

Now, assume that the update sizes are not always equal. To prove the amortized cost in this case, we have to find (1) the maximum accumulated prover time needed until recomputing the original table from scratch and (2) the minimum number of someone needs to insert in order to cause table recomputation. We will prove that (1) equals to $\mathcal{O}((\sum_{i \in [\log(n/k)]} \frac{n}{k2^{i-1}} (\text{Pt}_{\text{com}}(k2^{i-1}) + \text{Pt}(k2^{i-1}))))$ and (2) to $n/2$.

To prove (1), let $\mathcal{S} = \{(\mathcal{I}_1, \mathbf{t}_1), \dots, (\mathcal{I}_l, \mathbf{t}_l)\}$ be a sequence update pairs throughout the different invocations where $(\mathcal{I}_1, \mathbf{t}_1)$ is the first one and $(\mathcal{I}_l, \mathbf{t}_l)$ the final one, namely, the one causing the prover to recompute the table. Let $\mathcal{S}'$ be another sequence of updates such that $\mathcal{S} \subseteq \mathcal{S}'$ and when running $\mathcal{S}'$, the algorithm only uses steps 1,2. That new sequence is formed in the following way: Take any $(\mathcal{I}_i \in [n]^{k_i}, \mathbf{t}_i \in \mathbb{F}^{k_i})$ that uses step 3. If it lies in step 3.1 then compute new update pairs $\{(\mathcal{I}_i^{(k)}, \mathbf{t}_i^{(k)})\}_k$ populating them with pairs of $(\mathcal{I}_i, \mathbf{t}_i)$ such that when running the above algorithm over this sequence, the algorithm uses only steps 1 and 2 and we end up with $\mathcal{D}_{\mathcal{P}}[\log k_i] \neq \perp$ and $\mathcal{D}_{\mathcal{P}}[i] = \perp$ for all $i \in [\log(k_i) - 1]$. Insert those pairs before $(\mathcal{I}_i, \mathbf{t}_i)$. If it lies in step 3.2 then we do the same but using elements of $\mathcal{D}_{\mathcal{P}}[j_l]$. In other words, we “emulate” step 3 by adding new pairs that only use steps 1 and 2. Note that running the proving algorithm for $\mathcal{S}'$ takes at least the same time with that of $\mathcal{S}$ since the prover time incurred by running step 3 on pairs of $\mathcal{S}$ corresponds to

⁴Note that if the n added elements had duplicate indexes, then, the total prover time for n/k invocations ranges from $\mathcal{O}(\frac{n}{k} T_{\text{com}}(k))$ (e.g., each invocation updates the same set of indexes) and can be no slower than the worst case.

the time needed to insert the new pairs added in $\mathcal{S}'$. Furthermore, from the above analysis, since $\mathcal{S}'$ only uses steps 1 and 2 and because the total number of pairs of $\mathcal{S}'$ is at most twice that of $\mathcal{S}$, its total complexity will be $\mathcal{O}(2 \cdot (\sum_{i \in [\log(n/k)]} \frac{n}{k2^{i-1}} (\text{Pt}_{\text{com}}(k2^{i-1}) + \text{Pt}(k2^{i-1})))$.

What remains, is to show that $|\mathcal{S}| > n/2$. First, note by the way the merge operation works, $|\mathcal{S}|$ must contain at least $n/2 + 1$ unique indexes since otherwise the algorithm would “stuck” in $\mathcal{D}_{\mathcal{P}}[\log n - 1]$. In the worst case, $|\mathcal{S}|$ can be $n/2 + 1$, which can happen when the first update $(\mathcal{I}_1, \mathbf{t}_1)$ has size $n/2$ and the second one has size 1 and an index unique to any index of $\mathcal{I}_1$. Consequently, the worst case amortized cost will be $\mathcal{O}(4 \cdot (\sum_{i \in [\log(n/k)]} \frac{n}{k2^{i-1}} (\text{Pt}_{\text{com}}(k2^{i-1}) + \text{Pt}(k2^{i-1}))/n)$.

As for the proof size, In the worst case, all levels of $\mathcal{D}_{\mathcal{V}}$ changed. Since we have $\log n$ levels, the total proof size will be $\mathcal{O}(\text{Ps}(k) \log n)$. Likewise, we argue about the verification time.

Lookup Complexity. As for the lookup time, observe that in the worst case, all levels $\mathcal{D}_{\mathcal{P}}$ are populated with tables and all indexes of the lookup lie on the original table. In each iteration, the prover has to invoke a constant number of lookups of size k , two SNARK provers for a relation of total witness size of $\mathcal{O}(k)$ and perform at most k binary searches over the index vector. Consequently, the total prover time will be $\mathcal{O}((\text{Pt}(k) + k \log n) \cdot \log n + \sum_{i \in [\log n]} \text{Pt}_{\text{lkp}}(k, n/2^i))$. Likewise, we argue about the proof size and verification time.

Completeness. Completeness of the update algorithm derives from the completeness of the underlying SNARK while completeness of the lookup algorithm derives from the completeness of the underlying SNARK and static lookup argument.

Knowledge Soundness. First, we will show that the above transformation actually instantiates an updatable lookup argument—namely, the update and lookup algorithms are SNARKs for the relations $\mathcal{R}_{\text{upd}}$ and $\mathcal{R}_{\text{lkp}}$.

Update Algorithm. First, note that by the knowledge soundness of the lookup argument and the underlying SNARK, we can construct an extractor such that for any valid update proof π_{upd} , it extracts $\mathbf{x}$, $\mathbf{x}'$ and $(\mathcal{I}, \mathbf{t})$. This is done by invoking the extractor of the underlying SNARK. What remains, is to show that $(\mathbf{x}, \mathbf{x}', \mathcal{I}, \mathbf{t})$ satisfy the relation $\mathcal{R}_{\text{upd}}$.

First, observe that given access to $\mathcal{D}_{\mathcal{P}}$, one can construct the original table $\mathbf{x}$ by simply merging all updates $\mathcal{D}_{\mathcal{P}}[i]$, for $i \in [\log n - 1]$ and omitting older duplicate indexes (e.g., those stored in larger indexes). Then, apply all these changes to the original table of $\mathcal{D}_{\mathcal{P}}[\log n]$ and return $\mathbf{x}$.

Now, given the update pair $(\mathcal{I}, \mathbf{t})$ we need to show that the algorithm updates $\mathcal{D}_{\mathcal{P}}$ in such a way so that when we follow the above procedure we end up with the correct $\mathbf{x}'$. In more detail, if $(\mathcal{I}, \mathbf{t})$ satisfies case (1), then by default the above pair will be the newest entry in $\mathcal{D}_{\mathcal{P}}$ and thus $(\mathcal{I}, \mathbf{t})$ is merged with all other updates “with the highest priority”. Now, if the update pair lies on step (2) then, by the knowledge soundness of the SNARK, $\mathcal{R}_{\text{Merge}}$ must be satisfied which guarantees that for every possible recursion step $d \in [\log n]$, the merged result $(\mathcal{I}'', \mathbf{t}'')$ will contain all latest updates stored throughout $\mathcal{D}[i]$, $i \in 1, \dots, d$. Eventually, the prover either ends up with k' that satisfies case (1) which concludes our proof, or ends up with $k' = \log n$ where, again, $\mathcal{R}_{\text{Merge}}$ guarantees that all newest updates have been included in the original table. Similarly, we argue when the update lands in case 3.a. Finally, if it lands in 3.b, observe that in the first merge, the algorithm will merge all old updates and insert them in a single level of $\mathcal{D}_{\mathcal{P}}$. Because of the knowledge soundness of the underlying SNARK, the updated data structure $\mathcal{D}_{\mathcal{P}}$ will still store $\mathbf{x}$ since existing updates only got merged. Finally, we follow the same argument as in step 1 for proving the correctness of $\mathcal{D}_{\mathcal{P}}$ after the insertion of the update pair.

Lookup Algorithm. As for the lookup argument, its knowledge soundness comes directly from the knowledge soundness of the underlying SNARK and lookup argument. In more detail, assume that

the lookup pair contains a pair $i_j, \mathbf{t}[j]$ such that $\mathbf{x}[i_j] \neq \mathbf{t}[j]$. We will show that the probability the prover can generate an accepting proof is negligible. Let z be the smallest index where the index set of $\mathcal{D}_{\mathcal{P}}[z]$ contains i_j . Then, the following can happen (1) either the prover has to convince the verifier that i_j is not included in that index set or (2) the prover convinces that $\mathbf{t}[j]$ is the correct value at i_j or (3) the partitions $(\mathbf{t}_1, \mathcal{I}_1)$ or $(\mathbf{t}_2, \mathcal{I}_2)$ are computed incorrectly. Because of the knowledge soundness of the SNARK and lookup argument (1)-(3) can happen only with negligible probability.

C Details of our Static-to-Dynamic SNARK Compiler

In this section, we provide additional details about our compiler.

C.1 Our RAM model of Computation

In order to instantiate our compiler from updatable lookups to dynamic SNARKs, we need to be able to prove the correctness of a RAM program that reads and writes data from a huge dataset $\mathbf{T}$ in time that is proportional to the number of computing steps of the program and sublinear in $|\mathbf{T}|$. In this section, we formally define the RAM model that describes our programs and provide a brief overview of how their correctness can be efficiently proven.

In this work, we adopt the definition of the RAM computation model proposed in [84, 8, 5], but adapt it in order to better fit our application scenarios (in a similar way as [49] did for [41]). More specifically, our computation model consists of a CPU which executes instructions specified by an instruction set architecture (ISA). We note that our model puts no restriction on the choice of ISA, as long as the correctness of its instructions can be efficiently proven by a SNARK, as is the case for TinyRAM [8] and RISC-V [5]. Furthermore, our model consists of one large memory of size n (corresponding to the dataset $\mathbf{x}$), and a “small” memory serving as the working space for the underlying program. Likewise, there are two types of read and write operations. The standard ones that operate over the working space of the program and ones that read (write resp.) data from $\mathbf{x}$ (working space resp.) and write them to the working space ($\mathbf{x}$ resp.). Finally, our model also contains two tapes, one for a “private” input $\mathbf{w}$ and one for a “public” input $\mathbf{x}$.

Proving the Correctness of RAM Computations. Now, we have everything we need in order to introduce the following lemma which at a high level states that when give an updatable lookup argument and a SNARK then, one can use the techniques of [63, 36, 49, 18, 61] in order to efficiently prove the correctness of programs in the above RAM model. In more detail:

Lemma 2 (Informal) *Let Π be a RAM program as defined above, which, on input $(\mathbf{x}, \mathbf{w})$ reads/writes k_1/k_2 elements from/to a dataset $\mathbf{T}$ and takes $k \geq |\mathbf{w}|$ computing steps. Provided a SNARK with (Pt, Vt, Ps) and an updatable lookup argument with $(Pt_{ULA}(\cdot, \cdot) = (Pt_{ULA, \text{lkp}}(\cdot, \cdot) + Pt_{ULA, \text{upd}}(\cdot, \cdot)), Vt(\cdot), Ps(\cdot))$ prover/verification time and proof size, then, one can construct a SNARK for proving the correct computation of the above program with $\mathcal{O}(Pt_{ULA}(k_1 + k_2, n) + Pt(k))$ prover complexity, $\mathcal{O}(|\mathbf{x}| + Vt_{ULA}(k_1 + k_2, n) + Vt(\max(k)))$ verification complexity and $\mathcal{O}(Ps_{ULA}(k_1 + k_2, n) + Ps(k))$ proof size. After the completion of the proving and verification algorithms, the prover ends up with the updated dataset $\mathbf{T}'$ and auxiliary information $\mathbf{aux}'$ needed for efficiently generating the next proofs and the verifier ends up with the commitment for $\mathbf{T}'$.*

C.2 Certification and Update Algorithms

Lemma 3 *For any circuit C , there exists a RAM program Π_C (as defined in Section C.1) that takes as input $\mathbf{x}, \mathbf{y}$, has access to $\mathbf{T}_C$ checks if it is valid in time $\mathcal{O}(|C|)$. Furthermore, there exists a RAM program Π_U that takes as input a list of input updates $(\mathcal{I}, \mathbf{t})$, has access to a valid $\mathbf{T}_C$ (with respect to the original input/output pairs $(\mathbf{x}, \mathbf{y})$), and updates $\mathbf{T}_C$ to $\mathbf{T}'_C$ that is valid with respect to $(\mathbf{x}', C(\mathbf{x}'))$ where $\mathbf{x}'[[n] \setminus \mathcal{I}] = \mathbf{x}[[n] \setminus \mathcal{I}]$ and $\mathbf{x}'[\mathcal{I}] = \mathbf{t}$. Let $k = \mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C)$, where $\mathcal{D}$ corresponds to the number of wires the values of which have been changed. Π_U takes $\mathcal{O}(k)$ time.*

What Π_C simply does is to first check if the input and output are assigned correctly by checking if $\mathbf{T}_C[i].\text{value} = \mathbf{x}[i]$, for all $i \in [n]$ and $\mathbf{T}_C[|C| - m + i].\text{value} = \mathbf{y}[i]$ for $i \in [m]$. Next, it performs a post-order traversal over the graph representing the circuit (by accessing $\mathbf{T}_C$), and for each gate, it checks if its value is consistent with respect to the children values and operation type.

As for Π_U , it maintains a queue and initially inserts all pairs of $(\mathcal{I}, \mathbf{t})$. Then, it dequeues the next pair (i, v) from the queue. If i is an input index, then set $\mathbf{T}_C[i].\text{value} \leftarrow v$, otherwise, retrieve the values of the children of i and update $\mathbf{T}_C[i].\text{value}$ based on the given operation. If i is not an output gate, and its value has been changed compared to the previous one, insert all parents of that gate to the queue as pairs $(id, \perp)$. It is not hard to see that the total number of gates Π_U puts into the queue is proportional to the number of changed wires in the circuit, since it only inserts gates in the queue that are connected with at least one gate the value of which has been changed. For each such gate, Π_U performs a constant number of operations and only accesses the children of that gate (the number of which we assume constant). Hence, the complexity of Π_U is $\mathcal{O}(k)$.

Remark 5 *Note that when we use Π_U in the context of our dynamic SNARK, we slightly modify it to also check whether the output gate corresponds to 1.*

C.3 Proof of Theorem 7

We start with the following lemma, which will be useful in our proof:

Lemma 4 *Let $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}_C$ and $\mathbf{T}_C$ being a valid table with respect to $\mathbf{x}$ and $\mathbf{w}$. Furthermore, consider the following update pairs $(\mathcal{I}_x^{(1)}, \mathbf{x}_d^{(1)}, \mathcal{I}^{(1)}, \mathbf{w}_d^{(1)})$ and $(\mathcal{I}^{(2)}, \mathbf{x}_d^{(2)}, \mathcal{I}^{(2)}, \mathbf{w}_d^{(2)})$ and $\mathbf{T}_C^{(1)}$ and $\mathbf{T}_C^{(2)}$ the corresponding tables produced by sequentially running Π_U . Finally, let $(\mathcal{I}'_x, \mathbf{x}'_d, \mathcal{I}', \mathbf{w}'_d)$ be the merged update pair. Then, when running $\mathcal{P}_U$ with input $\mathbf{T}_C$ and the merged updates, we end up with $\mathbf{T}_C^{(2)}$. Furthermore, the running time of Π_U is at most $\mathcal{O}(k_1 + k_2)$ where $k_1 = \mathcal{D}(\mathbf{T}_C, \mathbf{T}_C^{(1)})$ and $k_2 = \mathcal{D}(\mathbf{T}_C^{(1)}, \mathbf{T}_C^{(2)})$.*

To prove the first part of our lemma, let $\mathbf{x}', \mathbf{w}'$ be the new instance/witness pair produced by performing all updates of $(\mathcal{I}'_x, \mathbf{x}'_d, \mathcal{I}', \mathbf{w}'_d)$. Note that $(\mathbf{x}^{(2)}, \mathbf{w}^{(2)}) = (\mathbf{x}', \mathbf{w}')$. That is because $(\mathbf{x}^{(2)}, \mathbf{w}^{(2)})$ encompasses all updates of $(\mathcal{I}_x^{(2)}, \mathbf{x}_d^{(2)}, \mathcal{I}^{(2)}, \mathbf{w}_d^{(2)})$ as well as those of the initial sequence, excluding the ones with common indexes which is identical to the merged update. Hence, $\mathbf{T}'_C = \mathbf{T}_C^{(2)}$ since the inputs are identical. As for the second part, first, observe that $\mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C) \geq \max(k_1, k_2)$ and that $\mathcal{D}(\mathbf{T}_C, \mathbf{T}'_C) \leq \mathcal{D}(\mathbf{T}_C, \mathbf{T}_C^{(1)}) + \mathcal{D}(\mathbf{T}_C^{(1)}, \mathbf{T}'_C) = k_1 + k_2$. Hence, as per Lemma 3, the running time of $\mathcal{P}_U$ over the merged updates will be $\mathcal{O}(k_1 + k_2)$.

Performance Analysis. First, we analyze prover complexity and then focus on proof size and verification time.

Prover Complexity. We start with the proving algorithm that has a prover complexity of $O(\text{Pt}(|C|))$ since Π_C requires $\mathcal{O}(|C|)$ computing steps. As for the update algorithm, we will analyze its amortized cost assuming that all update requests require $\mathcal{O}(k)$ time, i.e., the prover never lands on step 3 of **Construction 5**, since, as shown in our analysis of Section B for the updatable lookup argument, this step does not put any additional asymptotic overhead to the prover time. Furthermore, we consider the worst-possible scenario where all updates are done on different indexes. Having established that, observe that for a sequence of $|C|/k$ updates, the prover will invoke steps 1 or 2 for instances of size $2^{i-1}k, \frac{|C|}{2^{i-1}k}$ times. Hence, the total time needed to insert $|C|/k$ updates will be $\mathcal{O}(|C| \log^2 |C| + \sum_{i \in [\log(|C|/k)]} \frac{|C|}{2^{i-1}k} (\text{Pt}_{\text{ULA}}(2^{i-1}k, |C|) + \text{Pt}(2^{i-1}k)))$, where the additional $|C| \log^2 |C|$ overhead comes from the accesses made by the prover to the tables and their auxiliary information during proof generation of the lookup proof (see our protocol description). Consequently, the amortized cost per-update is $\mathcal{O}(k \log^2 |C| + \sum_{i \in [\log(|C|/k)]} \frac{1}{2^{i-1}} (\text{Pt}_{\text{ULA}}(2^{i-1}k, |C|) + \text{Pt}(2^{i-1}k)))$.

Verification Time & Proof Size. Regarding proof size, observe that in the worst case, the extracted $\mathcal{D}_V$ will have no non-empty entry and thus its size would be $\mathcal{O}(\log |C| \cdot (|\pi_U| + |\mathbf{x}|)) = \mathcal{O}(\log |C| \cdot (\text{Ps}_{\text{ULA}}(|C|, |C|) + \text{Ps}(|C|)))$. In this case, the proof π will also consist of $\log |C|$ merge proofs and a final π_C proof leading to a total proof size of $\mathcal{O}(\log |C| \cdot (\text{Ps}_{\text{ULA}}(|C|, |C|) + \text{Ps}(|C|)))$. Likewise, we argue about verification time.

Completeness. Here we will show that if $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}_C$ and all $(\mathbf{x}^{(z_i)}, \mathbf{w}^{(z_i)}) \in \mathcal{R}_C$ where $(\mathbf{x}^{(z_i)}, \mathbf{w}^{(z_i)})$ the instance/witness corresponding to the $\mathcal{D}_V[z_i]$ (and $\mathcal{Z} = (z_1, \dots)$ the vector containing the indexes of all non-empty entries of $\mathcal{D}_V$), then the verifier accepts the update proof with probability 1. In case $(\mathbf{x}, \mathbf{w})$ is the first pair to be proven, then the verifier will accept with probability 1 because of the completeness of the underlying SNARK. Otherwise, we first show that the verifier accepts all proofs in $\mathcal{D}_V$ with probability 1. Since $(\mathbf{x}^{(z_1=\log |C|)}, \mathbf{w}^{(z_1=\log |C|)}) \in \mathcal{R}_C$, then the verifier accepts the proof in $\mathcal{D}_V[\log |C|]$ with probability 1. Next, since $(\mathbf{x}^{(z_2)}, \mathbf{w}^{(z_2)}) \in \mathcal{R}_C$, it must hold that $\mathbf{T}_C^{(z_2)}$ must have been updated correctly from $\mathbf{T}^{(z_1)}$ and thus because of Lemma 2, the verifier will accept π_U with probability 1. Likewise, we argue for all other z_i .

Now, assume that π contains no merge proofs. Since $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}_C$ then, $\mathbf{T}_C$ must be a correct update from $\mathbf{T}^{(z_1)}$ and thus the verifier will accept π_U with probability 1. Otherwise, due to the completeness of the underlying SNARK, all merging proofs will be accepted with probability 1, and using Lemma 4 we argue that the final π_U will be accepted with probability 1.

Knowledge Soundness. To prove knowledge soundness, we consider the following cases:

Case 1: When $\mathcal{D}_V$ is empty everywhere except from $\mathcal{D}_V[\log |C|]$. Then, we can construct an extractor that outputs a witness $\mathbf{w}$ such that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}_C$. This extractor simply invokes the extractor of the underlying SNARK. Because of its knowledge soundness, the extracted witness will be valid with overwhelming probability.

Case 2: $\mathcal{D}_V$ is not empty everywhere. We will show how to extract a witness $\mathbf{w}$ for $\mathbf{x}$ that is valid with overwhelming probability. As a first step, our extractor obtains the original instance/witness pair $(\mathbf{x}', \mathbf{w}')$ lying in $\mathcal{D}_V$. We will first show that with overwhelming probability $(\mathbf{x}', \mathbf{w}') \in \mathcal{R}_C$. This is done by invoking the SNARK and Lookup extractor for each proof in $\mathcal{D}_V$ and obtain $\mathbf{w}^{(\log |C|)}, \mathbf{T}_C^{(\log |C|)}$ as well as $\{(\mathcal{I}_{\mathbf{x}}^{(i_j)}, \mathbf{x}_d^{(i_j)}, \mathcal{I}_{\mathbf{w}}^{(i_j)}, \mathbf{w}_d^{(i_j)}, \mathbf{T}_C^{(i_j)})\}_{i_j \in \mathcal{J}}$ where $\mathcal{J}$ is the index set containing all non-empty entries of $\mathcal{D}_V$ such that $i_j > i_{j+1}$. Let $\mathbf{w}'$ be $\mathbf{w}^{(\log |C|)}$ updated by the merge of all witness updates (and likewise we define $\mathbf{x}'$). Because of the knowledge soundness of the SNARK and lookup argument, it must hold that $\mathbf{T}_C^{(\log |C|)}$ is valid with overwhelming probability. In addition, $\mathbf{T}_C^{(i_1)}$ is valid since it was computed by $\mathcal{P}_U$ with input the valid table $\mathbf{T}_C^{(\log |C|)}$. We repeat the

same argument to conclude that $\mathbf{T}_C^{(i_{|\mathcal{I}|})}$ is valid and thus by Lemma 4, $(\mathbf{x}', \mathbf{w}')$ will be valid with overwhelming probability.

It remains to extract $\mathbf{w}$ and show that it is valid with overwhelming probability. For this, assume that the update proof has no merge proof. In this case, $\mathcal{D}_V[i] = \perp$ for all $i \in [k']$ and our extractor uses the SNARK extractor to obtain $(\mathcal{I}, \mathbf{w}_d)$ and $\mathbf{T}_C^{(k')}$. From the knowledge soundness property of the SNARK and Lemma 4, $\mathbf{T}_C^{(k')}$ must be valid based on $(\mathbf{x}, \mathbf{w})$, and $\mathbf{w}$ is the updated witness from $\mathbf{w}'$ with overwhelming probability. So, $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}_C$ with overwhelming probability. Next, assume that the proof contains merging proofs, but the verifier never lands on (3.b) of **Construction 5**. Our extractor uses the extractor of the SNARK protocol for $\mathcal{R}_{\text{merge}}$ to obtain the final merged update $(\mathcal{I}', \mathbf{w}'_d)$ (i.e., the one that ends up in step 1 of **Construction 5**) as well as all intermediate updates, including the original one $(\mathcal{I}, \mathbf{w}_d)$. By the knowledge soundness of the SNARK protocol, $(\mathcal{I}', \mathbf{w}'_d)$ must be valid with overwhelming probability. Finally, we follow a similar argument with that of the case where there are no merge proofs in order to show that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}_C$ with overwhelming probability. In a similar fashion, we treat the case where the verification algorithm lands on case (3.b) of **Construction 5**.