Batching-Efficient RAM using Updatable Lookup Arguments

Moumita Dutta
Indian Institute of Science
Bangalore, India
moumitadutta@iisc.ac.in

Chaya Ganesh
Indian Institute of Science
Bangalore, India
chaya@iisc.ac.in

Sikhar Patranabis
IBM Research India
Bangalore, India
sikhar.patranabis@ibm.com

Shubh Prakash
Indian Institute of Science
Bangalore, India
shubhprakash@iisc.ac.in

Nitin Singh
IBM Research India
Bangalore, India
nitisin1@in.ibm.com

ABSTRACT

RAM (random access memory) is an important primitive in verifiable computation. In this paper, we focus on realizing RAM with efficient batching property, i.e, proving a batch of m updates on a RAM of size N while incurring a cost that is sublinear in N. Classical approaches based on Merkle-trees or address ordered transcripts to model RAM correctness are either concretely inefficient, or incur linear overhead in the size of the RAM. Recent works explore cryptographic accumulators based on unknown-order groups (RSA, class-groups) to model the RAM state. While recent RSA accumulator based approaches offer significant improvement over classical methods, they incur linear overhead in the size of the accumulated set to compute witnesses, as well as prohibitive constant overheads.

We realize a batching-efficient RAM with superior asymptotic and concrete costs as compared to existing approaches. Towards this: (i) we build on recent constructions of lookup arguments to allow efficient lookups even in presence of table *updates*, and (ii) we realize a variant of sub-vector relation addressed in prior works, which we call *committed index lookup*. We combine the two building blocks to realize batching-efficient RAM with sublinear dependence on size of the RAM. Our construction incurs an amortized proving cost of $\widetilde{O}(m \log m + \sqrt{mN})$ for a batch of m updates on a RAM of size N. Our results also benefit the recent arguments for sub-vector relation, by enabling them to be efficient in presence of updates to the table. We believe that this is a contribution of independent interest.

We implement our solution to evaluate its concrete efficiency. Our experiments show that it offers significant improvement over existing works on batching-efficient accumulators/RAMs, with a substantially reduced resource barrier.

CCS CONCEPTS

• **Security and privacy → Public key (asymmetric) techniques**.

KEYWORDS

Succinct Arguments, Efficient RAM, Indexed lookups, Lookup Arguments, Rollups

ACM Reference Format:
Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, Shubh Prakash, and Nitin Singh. 2024. Batching-Efficient RAM using Updatable Lookup Arguments. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670356

1 INTRODUCTION

General purpose *Succinct Non-interactive Arguments of Knowledge* (SNARKs) enable one to generate succinct proofs of membership of a statement in an NP relation expressed as an arithmetic circuit. These proofs are extremely cheap to verify, which makes them useful for *Verifiable Computation* (VC), where a resource-constrained client (e.g., a mobile phone), can outsource an expensive computation to an untrusted server, and later verify the correctness of the computation at a minimal cost.

Modeling RAM in Verifiable Computation. It turns out that arithmetic circuit-based representations are inefficient in expressing relations involving the result of a program execution on memory/state. Such relations frequently arise in the context of verifiable computation, in scenarios that require proving the correctness of query execution against a database, inference from a decision tree, or updates on a table of account balances (e.g., when a batch of transactions, such as account transfers, is applied to the table).

In the aforementioned examples, objects such as database tables, decision trees, and accounts tables can be naturally modelled as instances of addressable memory, or more generally, random access memory (RAM), where one needs to prove that the RAM has been accessed/updated in accordance with the correct execution of the computation. There exists a rich and expanding body of work on efficiently modeling abstractions of RAM in verifiable computation. While a complete treatment of this vast body of work is beyond the scope of this paper (a fairly recent survey in [36] is a good starting point), we mention two additional properties that are often demanded of the RAM primitive: *persistence* – the ability to persist the RAM state across several computations, and *batching* – where verifiable update of the RAM state is required for small batches of updates. These properties are also the focus of this work.

Application to Blockchain Rollups. Batching-efficient RAM is especially relevant in the context of blockchain *rollups* [3], an umbrella term for recent efforts to scale blockchains by moving expensive computation off the blockchain to the so-called *layer two* (or L2) chains. The blockchain only needs to verify succinct proofs attesting to the correctness of the off-chain computation. This approach is popularly called *rollup* as it allows verifying the result of several (rolled-up) transactions modifying the L2 state, as part of one transaction verified on the main chain. This simultaneously improves scalability and lowers the cost (e.g., gas fees) per transaction due to succinct verification. We consider improving efficiency of rollups an important motivation for our work, but avoid precise details of a smart-contract based instantiation of our solution.

1.1 Our Contribution

We present batching-efficient RAM construction, which advances the efforts towards achieving *verifiable outsourcing of state update* such as in [12] and more recently in [16, 31]. The most popular approaches to succinctly represent state involve accumulators based on Merkle-trees [30], or ones based on groups of unknown order (e.g. RSA, class-groups) [7, 14, 16, 31]. The updates to the state are effected by insertions or deletions in the accumulated set. In this work, we model the state as an addressable memory (RAM) described by vector $\mathbf{T}$, which stores value v_i at address i. We denote this as $\mathbf{T}[\,i\,] = v_i$. The RAM supports two operations, viz, *loads* expressed as $v_i := \mathbf{T}[\,i\,]$, and *stores* expressed as $\mathbf{T}[\,i\,] = v_i$. We think of addresses $i \in [0, N]$ for some $N \in \mathbb{Z}$ while the values $v_i \in \mathbb{F}$ for some finite field $\mathbb{F}$. In our construction, we represent both the RAM and operations on it as polynomials, and use appropriate polynomial commitment schemes to obtain succinct commitments (digests) to them. We do not require commitments to be *hiding*, as our focus is on succinctness. We consider privacy as an orthogonal goal, one we believe is easily achievable via small adaptations to our construction. We summarize our contributions below.

- As our first contribution, we propose *update friendly lookup arguments*, which addresses the strict dependence of recent constructions on table-specific pre-processing parameters. Our innovation extends the utility of table-specific parameters to enable efficient lookups from tables, which are within certain Hamming distance of the pre-processed table.
- We construct *committed index lookup* arguments via black-box reduction to sub-vector arguments that use homomorphic commitments. A committed index lookup involves three committed vectors $\mathbf{t}$, $\mathbf{a}$ and $\mathbf{v}$ satisfying $v_i = t_{a_i}$ for all i. Similar definition is also used in recent multi-variate lookup arguments in [34], where a similar reduction to sub-vector arguments is obtained under a more restrictive assumption about the elements of the table.
- We crucially employ the above two contributions to construct a *batching-efficient* RAM, which can prove a batch of m updates with an amortized prover complexity of $O(m \log m + \sqrt{mN})$, with N being the size of the RAM. Our dependence on the RAM size is sublinear, in contrast to the linear complexity inherent in recent works on batching-efficient RAM using RSA accumulators [16, 31] or using generic memory checking techniques [4, 6, 35, 41]. All of our protocols are public-coin, and can be made non-interactive using standard techniques [22].
- We implement our scheme in Rust[1]. Experimentally, we show that our scheme performs significantly better than prior works, and is eminently deployable on a commodity hardware.

1.2 Techniques

We present a brief summary of our techniques below. A more detailed technical overview appears in Section 4.

Update-friendly Lookup Arguments. Our starting point is the recent line of works on lookup arguments which prove that a vector of size m appears as a sub-vector in a large fixed vector (table) of size N with succinct proof sizes and verification, but most notably ensuring that prover runs in time sublinear in the size of the table (N). The pioneering work [38] obtained prover complexity of $O(m^2 + m \log N)$, which was improved in subsequent works to $O(m^2)$ [33], $O(m \log^2 m)$ [39], and $O(m \log m)$ [15, 20]. However, the sublinear prover complexity requires table-dependent $O(N \log N)$ pre-processing and $O(N)$ storage. This table-dependent pre-processing implies that while the aforementioned lookup arguments can be used to obtain efficient ROM (read only memory) semantics they cannot be used as is for RAM (which supports update operations). Moreover, an update involving even a single index renders the entire $O(N)$ pre-processing unusable for further lookups, thus necessitating entire $O(N \log N)$ re-computation. This work is the first effort towards mitigating this rigid dependence, thereby increasing the applicability of the recent lookup arguments. An important contribution we make here is a new method for computing "encoded quotients" used in several recent lookup constructions such as [15, 20, 33, 38]. Our approach for computing these quotients from pre-computed parameters remains efficient even when the table is updated, and it directly applies to all the aforementioned constructions. For a table δ-hamming distance away from the pre-processed one, we incur $(m + \delta) \log^2(m + \delta)$ additional overhead for proving m lookups. To achieve such a quasi-linear overhead in both m and δ, we rely on novel algebraic algorithms described in Section 7. We informally summarize our contribution in this regard below, whereas Theorem 7.1 states the precise result.

Theorem 1.1 (Informal). *There exists a deterministic $O(N \log N)$ time algorithm $\mathsf{Preprocess}(\mathbf{T}) \to \mathsf{pp}_T$ which on input $\mathbf{T} \in \mathbb{F}^N$, outputs parameters pp_T of size $O(N)$ such that: Given pp_T, vectors $\mathbf{T}' \in \mathbb{F}^N$, $\mathbf{t} \in \mathbb{F}^m$ with $\mathbf{t}$ being a sub-vector of $\mathbf{T}'$ an argument of knowledge for the same can be computed in time $O((m+\delta)\log^2(m+\delta) + f(m))$ where $\delta = \Delta(\mathbf{T}, \mathbf{T}')$ is the Hamming distance between $\mathbf{T}$ and $\mathbf{T}'$ while $f(m)$ depends on the specific lookup protocol.*

For the constructions based on [33, 38], we set $f(m) = m^2$ in the above, while for [15, 20], we have $f(m) = m \log m$.

Committed Index Lookup: We augment the sub-vector relation in prior lookup arguments which considers whether each entry of a given vector appears in the target vector to one that also identifies the precise positions where the given vector appears in the target vector. When this relation is checked over commitments of the respective vectors; given vector, the target vector and the position vector, we call it *committed index lookup*. The relation we consider is similar to the one considered in [34]. For lookup

[1]https://anonymous.4open.science/r/updatableRAM/

arguments with homomorphic commitment schemes, we show that committed index lookup can be obtained using a sub-vector lookup argument (Lemma 6.1, Section 6.1). Such a construction was also considered in [34], but under a more restrictive assumption that the size of the elements in the table have to be within a certain bound. Lemma 6.1 yields a construction of committed index lookup that uses (a single instance of) the underlying sub-vector protocol in a black-box manner. This immediately implies efficient constructions of arguments for committed index lookups from [15, 20, 33, 38, 39]. In Appendix E of the full version of the paper [19], we also present an explicit (non-black-box) adaptation of [33] to obtain a committed index lookup, which again incurs costs comparable to a single instance of the underlying sub-vector protocol.

Batching-Efficient RAM from Lookup Arguments. Memory checking methods based on *address ordered transcripts* [4, 6, 35, 41], which are popularly used in efficient RAM abstractions, incur a cost linear in the size of the RAM. This is prohibitive for efficient batching. As a key idea in this work, we invoke committed index lookup on the large RAMs, to verifiably extract smaller sub-RAMs, which correspond to indices actually involved in the batch update. Then, we use the linear time memory-checking techniques to argue the consistency of these smaller sub-RAMs.

The idea needs to work through some more details, such as showing that the larger RAMs are identical on positions not referenced by the batch of updates (considered in Section 6.2). The overall idea is illustrated in Figure 1. We also note that the extracted sub-RAMs can have duplicate records, corresponding to multiple updates referencing the same RAM index; however, memory checking methods can be easily adapted to handle such cases. Finally, we would still hit the "rigidity" of lookup arguments in realizing this plan; once the table has changed, lookups are no longer efficient from it. To circumvent this, we use our first contribution on extending the utility of table-specific parameters to defer parameter re-computation optimally while still availing efficient lookups. More specifically, if we choose to re-compute the full table-specific parameters after k batches (of m updates each), the average cost per batch is $O(N \log N/k + mk \log^2(mk) + f(m))$. Here, $f(m)$ as earlier denotes complexity of the non-updatable base protocol. Setting $k \approx \sqrt{N/m}$ yields the average cost of m updates as $\widetilde{O}(f(m) + \sqrt{mN})$, which scales sublinearly with the size of the RAM. While the preceding analysis considers the worst case, in specific applications (such as account transactions, where few accounts contribute a large volume of transactions), it may be possible to further delay the computation of table-specific parameters. Thus we have:

Theorem 1.2 (Informal). *Given $m, N \in \mathbb{N}$, there exists an argument for verifiable RAM which proves updates of batch size m on RAM of size N with amortized prover complexity of $\widetilde{O}(f(m) + \sqrt{mN})$.*

Polynomial Protocol for RAM. There are several ways to implement the ordered transcript based memory consistency check on the smaller $O(m)$-sized RAMs, for example by expressing the same as an arithmetic circuit. However, for completeness, we also present an argument for RAM as an interactive *polynomial protocol* [26], which is then compiled into an argument of knowledge using the KZG [29] commitment scheme in the algebraic group model (AGM) [23]. This construction appears in Appendix B of full version of the paper [19].

2 RELATED WORK

Efficient modeling of the RAM primitive is a widely studied problem in verifiable computation (VC), due to its inherent usefulness in modeling several computations of interest. Encapsulating RAM semantics in VC circuits is also challenging; since (i) arithmetic/boolean circuits do not adequately model random access, and (ii) incorporating entire memory as gates in a circuit is prohibitive.

Several novel techniques have been proposed to work around the above limitations of circuit based representation of RAM. Among them, Merkle tree-based accumulators to model the RAM state are popular [5, 11, 12] as they can efficiently prove updates to the state, without modeling the entire memory in the arithmetic circuit. Other approaches based on *address ordered time-scripts* avoid the concrete costs of Merkle tree-based approaches by letting the prover provide inputs and outputs of RAM operations in a non-deterministic manner, which are then checked to satisfy consistency of *loads* and *stores*. Several works such as [4, 6, 35, 41] implement and improve variants of the aforementioned approach. Most transcript-based realizations of RAM only consider it to be transient, i.e, its state is useful only during the execution of a program, and do not consider *persistence* of the RAM state across several executions.

Another feature, which has only been considered in recent works [16, 31] is *batching*, where a verifiable update of RAM state is required for a batch of m updates, with m being much smaller than the RAM size. Both the Merkle tree-based approaches and the transcript-based approaches are inefficient with respect to batching. While constructions using Merkle tree-based accumulators (realized from collision-resistant hash functions) suffer from high concrete costs and poor ability to batch proofs, those based on checking consistency using transcripts incur a linear overhead in the RAM size.

Batching-Efficient RAM. There have been recent efforts [16, 31] on batching-efficient realization of the RAM primitive (see Table 1 for a summary of these schemes and the associated efficiency parameters). This is a natural setting in applications of verifiable computation, most notably in the context of blockchain *rollups*. Here, one is required to show that a batch of m transactions correctly updates the state of a table of account balances, which is maintained off-chain by the rollup provider. Here the batch-size m ranges from few hundreds to few thousands, whereas the table itself could contain several million accounts. Similar to prior work on batching-efficient RAMs [16, 31], our work is also motivated by the problem of enabling more efficient rollup for tables, which are naturally modeled as RAMs.

While the aforementioned works substantially mitigate disadvantages of both the Merkle-tree based approaches and transcript-based approaches by using RSA accumulators to model the state, they still incur large prover costs and memory requirements even for modest sized batches. The approaches in [16, 31] encode complex modular arithmetic over RSA groups and *hash to prime* functions as arithmetic circuits, which results in a fixed overhead of around 10 million R1CS constraints at batch sizes of $m = 1000$ (this overhead is significantly larger for [31]). This is already prohibitive on a modest hardware. In addition, the witness computation for each update incurs cost linear in the size of accumulated set. The prior works [16, 31] seek to mitigate this through pre-computation and parallel/distributed processing. However, the issue of maintaining pre-computed parameters in sync with dynamic accumulator state has not been adequately addressed in [16, 31]. For example, in RSA-based accumulators, generating $O(N)$ non-membership witnesses straightforwardly requires $O(N^2)$ time; however, these witnesses become stale once the accumulator state changes, and hence cannot be used as-is for subsequent update proofs.

Our approach considers both the cost of online proof generation as well as the offline cost of maintaining pre-computed parameters. In addition, our solution is almost "circuit-free". Our entire RAM operation is modeled as a polynomial protocol, which is readily transformed into an argument of knowledge using a polynomial commitment scheme. In an application of our primitive to rollups, the only part of the statement that would need to be expressed as a circuit is the verification of digital signatures on transactions (which is around 500 constraints per verification for EDDSA signatures). By suitably balancing the online and offline costs, we can prove a batch of 1000 updates on a RAM of size 1 million in an average of 90 seconds on a commodity laptop with a single-threaded implementation. Our performance can be substantially improved using a parallel implementation. See Sections 4 and 8 for more detailed discussions on the efficiency of our scheme.

Lookup Arguments. There are several recently proposed constructions of lookup arguments which enable proving that a vector of size m is a sub-vector of a larger (predetermined) vector of size N (see Table 1 for a summary of these schemes and the associated efficiency parameters). Of these, the very initial schemes [9, 10, 25] incur proving costs linear in N. Starting with Caulk [38], many lookup arguments were proposed with proving costs that are (largely) independent of N. Broadly, these schemes can be divided into categories based on how they achieve proving costs independent of N. The lookup arguments in the first category [15, 20, 24, 33, 38–40] use polynomial protocols in conjunction with the KZG polynomial commitment scheme, where the lookup efficiency relies crucially on *pre-computed* KZG opening proofs for the polynomial encoding the predetermined N-sized vector. We point out that naïvely adapting these lookup arguments for updatable tables/RAM is challenging since even small updates to the table require re-computing all of the opening proofs, which is prohibitively expensive (requires $\widetilde{O}(N)$ computation). To overcome the "rigidity", we propose a new method of constructing lookup arguments which allows re-using the pre-computed opening proofs across several batch updates, thus avoiding the need for re-computing after each batch (see Sections 4 and 7 for more details).

Lasso. The second category of lookup arguments is exemplified by the recently proposed Lasso scheme [2, 34], which enables efficient lookups for tables with a decomposable structure. Informally, a table $\mathbf{T} : \{0,1\}^n \to \{0,1\}^k$ is said to have a decomposable structure if there exists a decomposition of the table $\mathbf{T}$ into c sub-tables $\mathbf{T}_1, \ldots, \mathbf{T}_c : \{0,1\}^{n/c} \to \{0,1\}^\ell$ and a succinctly computable function f such that for $x = x_1 \| \ldots \| x_c$ where $x_i \in \{0,1\}^{n/c}$, we have

$$\mathbf{T}[x] = f(\mathbf{T}_1(x_1), \ldots, \mathbf{T}_c(x_c))$$

A simple example of such a function is a bit-wise AND (we refer to [2, 34] for a more detailed exposition). Lasso crucially leverages this decomposability of the table to reduce a lookup into a table of size $N = 2^n$ into c lookups, each into a table of size $N^{1/c}$. While this strategy works elegantly for tables with special structure, it is not compatible with arbitrary tables/updates, which is the focus of our work (in particular, in applications such as rollups, we need the ability to handle updates to arbitrary tables).

To summarize, existing lookup arguments achieve efficiency either by leveraging table-specific pre-processing or exploiting special structure, both of which do not naïvely extend to arbitrary dynamic tables. We focus on handling batch updates for arbitrary tables, and our techniques can be viewed as enabling the utility of table-specific pre-processing even *across batch updates.*

3 PRELIMINARIES

This section presents notations and preliminary background material used in the rest of the paper.

Notation. Throughout the paper, we assume a bilinear group generator BG which on input λ outputs parameters for the protocols. Specifically $\mathsf{BG}(1^\lambda)$ outputs $(\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, g_t)$ where:

- $\mathbb{F} = \mathbb{F}_p$ is a prime field of super-polynomial size in λ, with $p = \lambda^{\omega(1)}$.
- $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ are groups of order p, and e is an efficiently computable non-degenerate bilinear pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$.
- Generators g_1, g_2 are uniformly chosen from $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively and $g_t = e(g_1, g_2)$.

We write groups $\mathbb{G}_1$ and $\mathbb{G}_2$ additively, and use the shorthand notation $[x]_1$ and $[x]_2$ to denote group elements $x \cdot g_1$ and $x \cdot g_2$ respectively for $x \in \mathbb{F}$. We implicitly assume that all the setup algorithms for the protocols invoke BG to generate descriptions of groups and fields over which the protocol is instantiated. We use $[n]$ to denote the set of integers $\{1, \ldots, n\}$.

Lagrange Polynomials. We denote the Nth root of unity by ξ and define the subgroup $\mathbb{H}$ as $\mathbb{H} = \{\xi, \ldots, \xi^N\}$. Let $\{\mu_i(X)\}_{i=1}^N$ be the associated Lagrange basis polynomials over the set $\mathbb{H}$; that is, $\mu_i(X) = \prod_{j \neq i} \frac{X - \xi^j}{\xi^i - \xi^j}$. We denote by $Z_{\mathbb{H}}$ the vanishing polynomial of $\mathbb{H}$; $Z_{\mathbb{H}}(X) = X^N - 1$.

Formal Derivatives of Polynomials. For a polynomial $f(X) = \sum_{i=0}^d a_i X^i \in \mathbb{F}[X]$, we define its formal derivative to be the polynomial $f'(X) = \sum_{i=1}^d i a_i X^{i-1}$.

3.1 Succinct Arguments of Knowledge

Let $\mathcal{R}$ be a NP-relation and Ł be the corresponding NP-language, where Ł $= \{x : \exists\ w$ such that $(x, w) \in \mathcal{R}\}$. A *succinct argument of knowledge* consists of a pair of PPT algorithms $(\mathcal{P}, \mathcal{V})$. Given a public instance x, the prover $\mathcal{P}$, convinces the verifier $\mathcal{V}$, that $x \in$ Ł, where the prover additionally has as a witness w. We use the notation $b \leftarrow \langle \mathcal{P}(w), \mathcal{V} \rangle(x)$ to denote $\mathcal{V}$'s output in the interactive protocol involving $\mathcal{P}$ and $\mathcal{V}$ with w as $\mathcal{P}$'s input and x as the common input. The *knowledge-soundness* property says that if the verifier is convinced, then an efficient extractor algorithm given oracle access to the prover outputs a witness w such that $(x, w) \in \mathcal{R}$. An argument system is *succinct* if the communication complexity

Scheme	Setup	Proof Size	Prover Work	Verifier Work
Lookup Arguments for Static Tables				
Plookup [25]	Updatable	$5\mathbb{G}_1,\ 9\mathbb{F}$	$O(N)\ \mathbb{G}_1, O(N\log N)\ \mathbb{F}$	$2P$
LogUp [28, 32]		$O(1)\ \mathbb{G}_1,\ O(1)\ \mathbb{F}$	$O(N)\ \mathbb{G}_1, O(N\log N)\ \mathbb{F}$	$O(1)\ P$
Halo2 [9, 10]		$6\mathbb{G}_1,\ 5\mathbb{F}$	$O(N)\ \mathbb{G}_1, O(N\log N)\ \mathbb{F}$	$2P$
Caulk [38]		$14\mathbb{G}_1,\ 1\mathbb{G}_2,\ 4\mathbb{F}$	$O(m^2)\ \mathbb{G}_1, O(m^2+m\log N)\ \mathbb{F}$	$4P$
Caulk+ [33]		$7\mathbb{G}_1,\ 1\mathbb{G}_2,\ 2\mathbb{F}$	$O(m^2)\ \mathbb{G}_1, O(m^2)\ \mathbb{F}$	$3P$
Flookup [24]		$7\mathbb{G}_1,\ 1\mathbb{G}_2,\ 2\mathbb{F}$	$O(m)\ \mathbb{G}_1, O(m\log^2 m)\ \mathbb{F}$	$3P$
Baloo [39]		$12\mathbb{G}_1,\ 1\mathbb{G}_2,\ 4\mathbb{F}$	$14m\ \mathbb{G}_1, O(m\log^2 m)\ \mathbb{F}$	$5P$
CQ [20]		$8\mathbb{G}_1,\ 3\mathbb{F}$	$8m\ \mathbb{G}_1, O(m\log m)\ \mathbb{F}$	$5P$
CQ+ [15]		$7\mathbb{G}_1,\ 1\mathbb{F}$	$8m\ \mathbb{G}_1, O(m\log m)\ \mathbb{F}$	$5P$
Locq [40]	Trusted	$4\mathbb{G}_1,\ 1\mathbb{G}_2$	$6m\ \mathbb{G}_1, m\ \mathbb{G}_2, O(m\log m)\ \mathbb{F}$	$4P$
Lasso [34]	Transparent	$O(1)\ \mathbb{G},\ O(\log m)\ \mathbb{F}$	$o(cm+cN^{1/c})\ \mathbb{G}_1, O(cm)\mathbb{F}$	$O(\log m)\ \mathbb{F}, O(1)\ \mathbb{G}$
Lookup Arguments for Updatable Tables/Batching-Efficient RAM				
OWWB20 [31]	Trusted	$O(1)\ \mathbb{G}$	$\widetilde{O}(m)\ \mathbb{F}, \mathbb{G}, \widetilde{O}(N)\ \mathbb{G}'$	$1P$
B-INS-ARISA [16]		$O(1)\ \mathbb{G}$	$\widetilde{O}(m)\ \mathbb{F}, \mathbb{G}, \widetilde{O}(N)\ \mathbb{G}'$	$1P, O(1)\ \mathbb{G}'$
Our work (c.f. Table 2)	Updatable	$65\mathbb{G}_1,\ 1\mathbb{G}_2,\ 43\mathbb{F}$	$\widetilde{O}(\sqrt{mN})\ \mathbb{F}, \mathbb{G}_1$	$9P$

Table 1: Comparison with state-of-the-art lookup arguments. Here, N denotes the size of the RAM and m denotes the number of updates (typically $m \ll N$). We use $(\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, g_t)$ to denote a bilinear group ($\mathbb{G}$ denotes either $\mathbb{G}_1$ or $\mathbb{G}_2$, and is used in cases where the exact group is unspecified), and $\mathbb{G}'$ to denote an RSA group. P denotes a pairing evaluation. For Lasso, we report the overheads considering the polynomial commitment scheme Sona and assuming *structured* tables (here, c denotes an arbitrary positive integer). For our scheme, we report performance for the CQ-based instantiation.

and the complexity of $\mathcal{V}$ is polylogarithmic in the size of the witness. We provide formal definitions in Appendix A.2 of full version of the paper [19].

Fiat-Shamir. An interactive protocol is *public-coin* if the verifier's messages are uniformly random strings. Public-coin protocols can be transformed into non-interactive arguments in the Random Oracle Model (ROM) by using the Fiat-Shamir [22] heuristic to derive the verifier's messages as the output of a Random Oracle.

Modular Approach. A modular approach for designing efficient succinct arguments consists of two steps; constructing an information theoretic protocol in an idealized model, and then compiling the information-theoretic protocol via a cryptographic compiler to obtain an argument system. Informally, the prover and the verifier interact where the prover provides oracle access to a set of polynomials, and the verifier accepts or rejects by checking certain identities over the polynomials output by the prover and possibly public polynomials known to the verifier. Such a protocol can be compiled into a succinct argument of knowledge by realizing the polynomial oracles using a *polynomial commitment scheme*. A polynomial commitment scheme allows a prover to commit to polynomials, and later verifiably open evaluations at chosen points by giving evaluation proofs. This enables the verifier to probabilistically check polynomial identities at random points of $\mathbb{F}$. Many recent constructions of zkSNARKs [13, 17, 26] follow this approach where the information theoretic object is a polynomial interactive oracle proof (PIOP) (or a polynomial protocol), and the cryptoprimitive in the compiler is a polynomial commitment scheme.

3.2 Security Model

We describe public-coin interactive protocols in the structured reference string (SRS) model where both the parties have access to a SRS. The SRS in our protocols consists of encodings of monomials of the form $\{[x^i]_1\}_{a\le i\le b}$, $\{[x^i]_2\}_{c\le i\le d}$ for x chosen uniformly from $\mathbb{F}$ and a, b, c, d are bounded by some polynomial in λ. It then follows from [8] that such an SRS can be generated using a universal and updatable setup [27] requiring only one honest participant. In practice, this is a superior security model compared to requiring a fully trusted setup. We use $\mathsf{srs} = (\mathsf{srs}_1, \mathsf{srs}_2)$ to denote the structured reference string of the above form. We say that the srs has degree Q if all the elements of srs_i, $i = 1, 2$ are of the form $[f(x)]_i$ for a polynomial $f \in \mathbb{F}_{<Q}[X]$.

Algebraic Group Model. We analyze security of our protocols in the Algebraic Group Model (AGM) introduced in [23]. An adversary $\mathcal{A}$ is called *algebraic* if every group element output by $\mathcal{A}$ is accompanied by a representation of that group element in terms of all the group elements that $\mathcal{A}$ has seen so far (input and output). In the AGM, an adversary $\mathcal{A}$ is restricted to be *algebraic*, which in our SRS-based protocol means a PPT algorithm satisfying the following: for $i \in \{1, 2\}$, whenever $\mathcal{A}$ outputs an element $A \in \mathbb{G}_i$, it is accompanied by its representation, $\mathcal{A}$ also outputs a vector $\mathbf{v}$ over $\mathbb{F}$ such that $A = \langle \mathbf{v}, \mathsf{srs}_i \rangle$.

Real and Ideal Pairing Checks: For an algebraic adversary $\mathcal{A}$ interacting in a protocol with a degree Q SRS over a bilinear group, the verifier can use the pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ to perform "ideal check" of the form $(R_1 \cdot T_1) \cdot (R_2 \cdot T_2) \equiv 0$, where R_1, R_2 are vectors of polynomials over $\mathbb{F}$ and T_1, T_2 are public matrices over $\mathbb{F}$. Under the Q-DLOG assumption stated below, the aforementioned ideal check is equivalent (except with a negligible probability) to a real pairing check $(a \cdot T_1) \cdot (T_2 \cdot b) = 0$ with a and b denoting vectors in $\mathbb{F}$ encoding polynomials in R_1 and R_2 in groups $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively (see [26, Lemma 2.2]).

Definition 3.1 (Q-DLOG Assumption [23]). *Fix an integer Q. The Q-DLOG assumption for $(\mathbb{G}_1, \mathbb{G}_2)$ states that given $[1]_1, [x]_1, \ldots, [x^Q]_1, [1]_2, [x]_2, \ldots, [x^Q]_2$ for uniformly chosen $x \leftarrow \mathbb{F}$, the probability of an efficient $\mathcal{A}$ outputting x is $\mathsf{negl}(\lambda)$.*

3.3 KZG Commitment Scheme

A polynomial commitment scheme allows the prover to open evaluations of a committed polynomial succinctly (Appendix A.1 of full version of the paper [19]). We use the KZG commitment scheme introduced in [29] which satisfies succinctness, completeness and knowledge-soundness (extractability) in the algebraic group model, while additionally featuring a universal and updatable setup. We denote the KZG scheme by the tuple of PPT algorithms (KZG.Setup, KZG.Commit, KZG.Prove, KZG.Verify) as defined below.

Definition 3.2 (KZG Polynomial Commitment Scheme). *Let $(\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, g_t)$ be output of bilinear group generator $\mathsf{BG}(1^\lambda)$ for security parameter λ. The KZG polynomial commitment scheme is defined as follows:*

- KZG.Setup *on input $(1^\lambda, d)$, where d is the degree bound, outputs* $\mathsf{srs} = (\{[\tau]_1, \ldots, [\tau^d]_1\}, \{[\tau]_2, \ldots, [\tau^d]_2\})$.
- KZG.Commit *on input $(\mathsf{srs}, p(X))$, where $p(X) \in \mathbb{F}_{\leq d}[X]$, outputs $C = [p(\tau)]_1$*
- KZG.Prove *on input $(\mathsf{srs}, p(X), \alpha)$, where $p(X) \in \mathbb{F}_{\leq d}[X]$ and $\alpha \in \mathbb{F}$, outputs (v, π) such that $v = p(\alpha)$ and $\pi = [q(\tau)]_1$, for $q(X) = (p(X) - p(\alpha))/(X - \alpha)$.*
- KZG.Verify *on input $(\mathsf{srs}, C, v, \alpha, \pi)$, outputs 1 if the following equation holds, and 0 otherwise.*

$$e(C - v[1]_1 + \alpha\pi, [1]_2) = e(\pi, [\tau]_2)$$

Note that both sides of the verification equation involve a fixed generator, and hence several proof verifications can be batched together to reduce the number of pairing computations. We also assume (w.l.o.g) analogues of KZG.Commit, KZG.Prove and KZG.Verify defined over the group $\mathbb{G}_2$. We shall use the (non-standard) notation $[p(X)]_i$ to denote $[p(\tau)]_i$ for $i \in \{1, 2\}$. This allows us a convenient shorthand for referring to "commitment of the polynomial $p(X)$" in group $\mathbb{G}_i$. Our protocols also use batched KZG proofs to show that polynomial $p(X)$ satisfies $p(\alpha_i) = v_i$ for $i \in [n]$. Let $\alpha = (\alpha_1, \ldots, \alpha_n)$ denote the vector of evaluation points and $\mathbf{v} = (v_1, \ldots, v_n)$ denote the vector of claimed evaluations. Then the batched version of KZG.Prove is described as follows:

- KZG.Prove on input $(\mathsf{srs}, p(X), \alpha)$, where $p(X) \in \mathbb{F}_{\leq d}[X]$ and $\alpha \in \mathbb{F}^n$, outputs $(\mathbf{v}, \pi)$ with $\mathbf{v} \in \mathbb{F}^n$ such that $v_i = p(\alpha_i)$ and $\pi = [q(\tau)]_1$ where

$$q(X) = \frac{p(X) - r(X)}{a(X)}$$

In the above, $a(X) = (X - \alpha_1) \cdots (X - \alpha_n)$, while $q(X)$ and $r(X)$ are the quotient and remainder polynomials when $p(X)$ is divided by $a(X)$.

- KZG.Verify on input $(\mathsf{srs}, C, \mathbf{v}, \alpha, \pi)$, outputs 1 if the following equations hold, and 0 otherwise.

$$e(C - [r(\tau)]_1, [1]_2) = e(\pi, [a(\tau)]_2)$$

Here, the verifier interpolates the polynomial $r(X) \in \mathbb{F}_{<n}[X]$ such that $r(\alpha_i) = v_i$.

KZG for Vectors. For $\mathbf{f} \in \mathbb{F}^N$, let $\mathsf{Enc}_{\mathbb{H}}(\mathbf{f})$ denote the polynomial encoding of $\mathbf{f}$ over $\mathbb{H}$ given by $\sum_{i=1}^{N} f_i \mu_i(X)$. We use KZG to commit to *vectors* by committing to its polynomial encoding. In general a vector $\mathbf{g}$ of size m is encoded by a polynomial $g(X) \in \mathbb{F}_{<m}[X]$ which interpolates $\mathbf{g}$ over a subgroup $\mathbb{V}$ consisting of m^{th} roots of unity in some canonical order. We will explicitly state the subgroups for all sizes of vectors that we consider.

3.4 Lookup Arguments

Works on lookup arguments [20, 33, 38, 39] consider proving sub-vector relation over committed vectors, i.e, given commitments c_t and c_v to vectors $\mathbf{t} \in \mathbb{F}^N$ and $\mathbf{v} \in \mathbb{F}^m$, one proves that for all $i \in [m]$, there exists $j \in [N]$ such that $v_i = t_j$. We will use $\mathbf{v} \preceq \mathbf{t}$ to denote that $\mathbf{v}$ is a sub-vector of $\mathbf{t}$. The definition below summarizes the sub-vector relation as defined in prior works.

Definition 3.3. *We define the* committed sub-vector *relation $\mathcal{R}^{\mathsf{subvec}}_{\mathsf{srs},N,m}$ to consist of tuples $((c_t, c_v), (\mathbf{t}, \mathbf{v}))$ where $c_t, c_v \in \mathbb{G}_1$, $\mathbf{t} \in \mathbb{F}^N$, $\mathbf{v} \in \mathbb{F}^m$ such that $\mathbf{v} \preceq \mathbf{t}$ and $c_t = \mathsf{KZG.Commit}(\mathsf{srs}, \mathsf{Enc}_{\mathbb{H}}(\mathbf{t}))$ and $c_v = \mathsf{KZG.Commit}(\mathsf{srs}, \mathsf{Enc}_{\mathbb{V}}(\mathbf{v}))$.*

A committed sub-vector argument is an argument of knowledge for the relation $\mathcal{R}^{\mathsf{subvec}}_{\mathsf{srs},N,m}$. Next, we consider a slightly modified relation that we call *committed index lookup* (called indexed lookup in [34]) where there is a commitment to the indices where $\mathbf{v}$ appears in $\mathbf{t}$. Formally, we define it as below:

Definition 3.4. *We define the* committed index lookup *relation $\mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m}$ to consist of tuples $((c_t, c_a, c_v), (\mathbf{t}, \mathbf{a}, \mathbf{v}))$ where $c_t, c_a, c_v \in \mathbb{G}_1$, $\mathbf{t} \in \mathbb{F}^N$, $\mathbf{a}, \mathbf{v} \in \mathbb{F}^m$ such that $v_i = \mathbf{t}[a_i] = t_{a_i}$ for all $i \in [m]$ and $c_t = \mathsf{KZG.Commit}(\mathsf{srs}, \mathsf{Enc}_{\mathbb{H}}(\mathbf{t}))$, $c_a = \mathsf{KZG.Commit}(\mathsf{srs}, \mathsf{Enc}_{\mathbb{V}}(\mathbf{a}))$ and $c_v = \mathsf{KZG.Commit}(\mathsf{srs}, \mathsf{Enc}_{\mathbb{V}}(\mathbf{v}))$.*

A committed lookup argument is a succinct argument of knowledge for the relation $\mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m}$.

4 TECHNICAL OVERVIEW

As we have alluded to earlier, existing memory-checking based techniques to model RAM computations incur a cost that is linear in the size of the RAM. We are interested in the setting where the number of operations whose execution is to be verified is much smaller than the size of the RAM. Thus, our goal is to achieve prover complexity which is *sublinear* in the size of the RAM. Before we proceed, we establish a working definition of RAM for the rest of the paper. Informally, a RAM maps indices (addresses) to values, where we assume that values come from a finite field $\mathbb{F}$, while indices come from a subset I of $\mathbb{F}$. For us, I will generally be the set $\{1, \ldots, k\}$ for some integer k (which may be different from size of the RAM n). Finally, for an index, there should be at most one value in the RAM, i.e., the association is unambiguous. The formal definition of RAM is as follows:

Definition 4.1 (RAM). *Given $n \in \mathbb{N}$, finite field $\mathbb{F}$ and a set $\mathcal{I} \subseteq \mathbb{F}$, a RAM of size n over indices $\mathcal{I}$ is a tuple $\mathbf{T} = (\mathbf{a}, \mathbf{v}) \in \mathcal{I}^n \times \mathbb{F}^n$ such that $\forall\, i, j \in [n]$ $v_i = v_j$ whenever $a_i = a_j$. We think of $\mathbf{T}$ as a table with vectors $\mathbf{a}$ and $\mathbf{v}$ denoting its columns. The set of all such tables will be denoted by $\mathsf{RAM}_{\mathcal{I},n}$.*

For a table $\mathbf{T} = (\mathbf{a}, \mathbf{v}) \in \mathsf{RAM}_{\mathcal{I},n}$, we refer to tuples (a_i, v_i), $i \in [n]$ as records of the table $\mathbf{T}$. We use the access notation $v = \mathbf{T}[a]$ to mean that (a, v) is a record of $\mathbf{T}$ (note there can be multiple such records according to our definition). When we consider RAMs where the first column (of indices) is of the form $\mathbf{I}_n = (1, 2, \ldots, n)$, we simply denote such RAMs by $\mathbf{T} \in \mathbb{F}^n$. For a RAM $\mathbf{T} \in \mathsf{RAM}_{\mathcal{I},n}$, a RAM operation is a three tuple (op, a, v) with $op \in \{0, 1\}$, $a \in I$ and $v \in \mathbb{F}$. An operation with $op = 0$ is called a *load* operation which denotes reading a value v mapped to index a in the RAM. Similarly, an operation with $op = 1$ is called a *store* operation, which denotes associating the value v with index a in the RAM. We use $O_{\mathcal{I}}$ to denote the set of all RAM operations with index set I.

4.1 Batching-Efficient RAM: Blueprint

We will use a vectors in $\mathbb{F}^N$ to denote the "large" RAMs, where index column is implicitly assumed to be $(1, \ldots, N)$. Let $\mathbf{T}, \mathbf{T}' \in \mathbb{F}^N$ denote the initial and final RAM states, and let $\mathbf{o}$ be a sequence of m operations $(m < N)$ which updates $\mathbf{T}$ to $\mathbf{T}'$. Let $\mathbf{a} \in \mathbb{F}^m$ denote the vector of RAM indices referenced by the operations in $\mathbf{o}$, i.e, a_i is the index referenced by the i^{th} operation. To prove the transformation of $\mathbf{T}$ to $\mathbf{T}'$ via operation sequence $\mathbf{o}$, we proceed as follows:

- We isolate sub-tables $\mathbf{S} = (\mathbf{a}, \mathbf{v})$ and $\mathbf{S}' = (\mathbf{a}, \mathbf{v}')$ of $\mathbf{T}$ and $\mathbf{T}'$ consisting of rows corresponding to indices in $\mathbf{a}$. This requires proving $\mathbf{v} = \mathbf{T}[\mathbf{a}]$ and $\mathbf{v}' = \mathbf{T}'[\mathbf{a}]$, which we show using *committed index lookup* argument discussed in Section 6.1.
- On the isolated sub-tables S and S′ of size m, we use the standard memory checking arguments (c.f. argument presented in Appendix B of full version of the paper [19]) to prove that sequence $\mathbf{o}$ correctly updates S to S′ with prover complexity of $\widetilde{O}(m)$.
- Finally, we show that the RAMs $\mathbf{T}$ and $\mathbf{T}'$ are identical outside indices in $\mathbf{a}$. We describe the protocol for proving the same in Section 6.2.

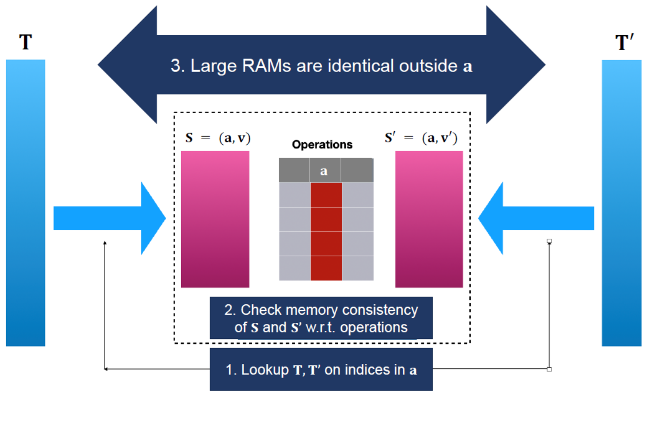
T
T′
3. Large RAMs are identical outside a
S = (a, v)
Operations
a
S′ = (a, v′)
2. Check memory consistency of S and S′ w.r.t. operations
1. Lookup T, T′ on indices in a

Figure 1: Illustrating different steps of sublinear lookup protocol between large RAMs T and T′.

The blueprint for the above approach is illustrated in Figure 1.

4.2 Batching-Efficient RAM: Components

We now elaborate on the key technical components in realizing the above blueprint.

Committed Index Lookup. To limit the size of the RAM on which we use memory-checking techniques, our first step is to isolate sub-tables of RAMs $\mathbf{T}$ and $\mathbf{T}'$ corresponding to addresses which are involved in the operations. This is achieved by looking up RAMs $\mathbf{T}$ and $\mathbf{T}'$ at indices in the committed vector $\mathbf{a}$. We could leverage the recent work on efficient lookup arguments to verifiably extract m indices from a table of size N, in time dependent only on m. However, there are two technical challenges here. First, the aforementioned lookup arguments only prove the sub-vector relation, without linking the extracted vector to the indices in $\mathbf{a}$. This is easily solved, as there is an efficient realization of a *committed index lookup* from a *committed sub-vector* argument, where the commitment scheme is homomorphic. The details appear in Section 6.1, with the complete protocol presented in Figure 2. The second challenge is much more formidable: the efficiency of sub-vector arguments (and the committed index lookup argument derived from them) depends on *expensive* table-specific pre-processing. This is acceptable when the table in question is static, but is infeasible in our setting requiring updatable tables. This motivates our next technical component.

Fast Lookup from Approximate Setup. We build upon the rich body of work on polynomial protocols enabling efficient lookups from static tables [20, 33, 38, 39], which rely on expensive table-dependent pre-computation to optimise online proving performance. We make the first attempt towards breaking this rigid dependence. Our key idea is to extend the utility of pre-computed parameters for a table $\mathbf{T}$, to proving lookups from tables $\mathbf{T}' \neq \mathbf{T}$. We show that for $\delta = \Delta(\mathbf{T}, \mathbf{T}')$, an argument for m lookups from $\mathbf{T}'$ incurs an additional prover overhead of $(m + \delta)\log^2(m + \delta)$ over the lookup argument for static tables. We note that the overhead is *quasi*-linear in both m and δ. Our competitive overhead rests on several innovative applications of algebraic algorithms, which are summarised in Appendix D.1 of the full version [19]. We then leverage this ability to use "approximate" setup into a *base + cache* strategy; where at all times we maintain pre-computed parameters corresponding to a base table $\mathbf{T}_b$, and use this setup to prove lookups from the current table $\mathbf{T}$. We achieve optimal prover effort on average by using parameters for $\mathbf{T}_b$ till the current table is at a hamming distance at most $\sqrt{mN}$ from $\mathbf{T}_b$, beyond which we recompute full parameters for the current table with $O(N \log N)$ prover effort. The cycle then repeats with current table as the base table.

Naive Approaches are Inadequate. We notice that the aforementioned constructions of lookup arguments require linear combination of encoded quotients of the form $\left[(T(X) - T(\xi^i))/(X - \xi^i)\right]_g$ for upto m values of i during the proof generation. While constructions [33, 38] consider quotients encoded in the group $\mathbb{G}_2$, the protocol in [20] encodes them in $\mathbb{G}_1$. We use a generic $[\,\cdot\,]_g$ to account for protocol-specific choices. We also see that even a small change to the table requires one to update all the quotients (the polynomial $T(X)$ is common to all quotients). Updating all the quotients after each batch is clearly infeasible. One could consider delaying the updation of the quotients, till the time they are actually required in a proof, which happens when the corresponding index

Component	Protocol	Prover Work	Verifier Work	Communication
Committed Sub-vector Lookup	CQ [20]	$O(m \log m)\,\mathbb{F}$, $O(m)\,\mathbb{G}_1$	$5P$	$8\mathbb{G}_1$, $3\mathbb{F}$
Committed Index Lookup	Figure 2	$O(m \log m)\,\mathbb{F}$, $O(m)\,\mathbb{G}_1$	$5P$	$8\mathbb{G}_1$, $3\mathbb{F}$
Localized Update in RAM	Figure 3	$O(m \log^2 m)\,\mathbb{F}$, $O(m)\,\mathbb{G}_1$	$8P$	$19\mathbb{G}_1$, $1\mathbb{G}_2$, $10\mathbb{F}$
Table Specific Preprocessing	Fast KZG [21]	$O(N \log N)\,\mathbb{F}$, $\mathbb{G}$	-	-
Lookup from Approximate Setup	Section 7	$O((m+\delta)\log^2(m+\delta))\,\mathbb{F}$, $O(m+\delta)\,\mathbb{G}_1$	-	-
Polynomial Protocol for RAM	Figure 11 [19]	$O(m \log m)\,\mathbb{F}$, $O(m)\,\mathbb{G}$	$7P$	$36\mathbb{G}_1$, $30\mathbb{F}$
Batching-Efficient RAM	Figure 4	$\widetilde{O}(\sqrt{mN})\,\mathbb{F}, \mathbb{G}$	$9P$	$65\mathbb{G}_1$, $1\mathbb{G}_2$, $43\mathbb{F}$

Table 2: Asymptotic efficiency of the component protocols for our scheme. Here, N denotes the size of the RAM, m denotes the number of operations, and δ denotes Hamming distance of table for which pre-computed parameters are available from the current table. As before, we use $(\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2, g_t)$ to denote a bilinear group, and P to denote a pairing evaluation. The performance figures reported here correspond to the CQ-based realization of our batching-efficient RAM scheme.

in the table is involved in lookup. However, each of the m quotients is now potentially "lagging" by δ updates, so we would need $\Omega(m\delta)$ group operations to refresh all of them. This gives us multiplicative degradation with δ, and is clearly unsustainable for reasonable values of δ. In Section 7, we present an efficient method to directly compute linear combination of upto $O(m)$ encoded quotients of the form $\left[(T(X) - T(\xi^i))/(X - \xi^i)\right]_g$.

Localizing changes in RAMs. While the above two components allow us to reliably extract sub-RAMs corresponding to indices in vector $\mathbf{a}$, we still need to prove that RAMs are identical outside indices in $\mathbf{a}$. Looking ahead, in terms of polynomials this requires proving that $T(\xi^i) = T^*(\xi^i)$ for $i \notin \{a_i : i \in [m]\}$. Assuming $Z_I(X)$ to be the vanishing polynomial of the set $\{\xi^{a_i} : i \in [m]\}$, this is equivalent to proving that $Z_I(X)(T(X) - T^*(X)) = D(X)Z_{\mathbb{H}}(X)$ for some polynomial D. However, naively this involves working with polynomials with degree $O(N)$, which is expensive. In Section 6.2 we show a polynomial protocol for the above relation which requires only $O(m \log^2 m)$ prover effort. The protocol appears in Figure 3.

Polynomial Protocol for Memory Checking. To complete the verification, we need to show that the smaller RAMs, $\mathbf{S} = (\mathbf{a}, \mathbf{v})$ and $\mathbf{S}' = (\mathbf{a}, \mathbf{v}')$ extracted from larger RAMs $\mathbf{T}, \mathbf{T}'$ are consistent with respect to the operations. This can be accomplished using standard memory checking techniques based on address ordered transcripts, which we formalize in Section 5. Futher in Appendix B and C of full version of the paper [19] we assemble known techniques to present a polynomial protocol for memory consistency based on address ordered transcripts. This involves encoding several artefacts such as operations, transcripts etc., as polynomials and relations among them such as concatenation, permutation and monotonicity as polynomial identities. Our modelling is simple and implementation friendly, and helps in realizing a "circuit-free" overall construction. Complete polynomial protocol for memory checking appears in Figure 11, while constituent protocols appear in Figures 9, 8 and 10 of the full version of the paper [19].

Efficiency. We conclude the overview with a discussion of efficiency achieved by our scheme, and how different components discussed in this section contribute to the overall efficiency. The asymptotic performance of our scheme using CQ [20] is summarized in Table 2, with efficiency of the overall scheme highlighted in gray. The table also serves as a ready-reckoner for component protocols involved in the overall scheme. A more detailed discussion and break-up of the polynomial protocol for RAM appears in Table 6 in Appendix B of full version of the paper [19]. We note that the verification complexity of the overall solution is substantially less than the aggregate of component protocols; this is due to the fact that several pairing checks required for KZG verification proofs can be batched together. For concrete instantiation using BLS12-381 curve, RAM size of 1 million, the online cost of proving an update of 1000 operations on a table as a function of its Hamming distance from the "pre-processed" table is described in Figure 6. Other performance metrics are summarised in Tables 5 and 3. We refer to Section 8 for a more detailed performance evaluation and comparison with prior work. As is clear from the tables above, the (offline) parameter re-computation is the most expensive operation. We reiterate that all of our reported costs throughout the paper are for a single-threaded implementation on a consumer-grade laptop. We believe that parallel implementations can substantially speed up parameter re-computation as their cost is dominated by FFTs over group polynomials, which are highly parallelizable.

Continuity. To support applications such as rollups, we also consider it imperative to ensure that online proof generation does not halt during offline parameter re-generation. In other words, offline parameter re-generation should not hinder the operational continuity of the system. In our scheme, we can ensure this by carefully overlapping the offline computation with online proof generation such that the system can *instantly* switch to using the more recently generated parameters before the online proving time becomes prohibitive. We present more concrete discussions around this scheduling at the end of Section 8. We note that prior works [16, 31] have not addressed this issue of continuity in detail. To the best of our knowledge, we are the first to highlight the issue and present a discussion on a viable approach.

5 MEMORY CONSISTENCY FOR RAM

In this section, we briefly review and formalize existing memory-checking techniques to ensure correctness of RAM operations. The formal definitions for various relations involved in memory checking will be used to describe polynomial protocol for RAM in Appendix B of the full version of the paper [19].

5.1 Correctness of RAM Update

The versatility of the RAM primitive stems from its updatability. While a load operation leaves the RAM unchanged, the store operation updates the value in the RAM associated with the referenced index. We model the update via the function $\mathsf{Upd}_{\mathcal{I}}$ which takes RAM $\mathbf{T} \in \mathsf{RAM}_{\mathcal{I},n}$, operation $o = (op, a, v) \in O_{\mathcal{I}}$ as inputs and returns an updated RAM $\mathbf{T}' \in \mathsf{RAM}_{\mathcal{I},n}$. The updated RAM $\mathbf{T}' = \mathsf{Upd}_{\mathcal{I}}(\mathbf{T}, o)$ satisfies $\mathbf{T}' = \mathbf{T}$ if $op = 0$ while for $op = 1$ it satisfies $\mathbf{T}'[\,a\,] = v$ and $\mathbf{T}'[\,x\,] = \mathbf{T}[\,x\,]$ for $x \neq a$. The central problem in verifiable RAM protocols is to establish that a sequence of operations $\mathbf{o} = (o_1, \ldots, o_m)$ are correct with respect to the initial RAM state $\mathbf{T}$ and the final RAM state $\mathbf{T}'$. This involves ensuring that all load operations read the value which is consistent with updates to the RAM as a result of preceding store operations, and that $\mathbf{T}'$ is the final state. We say that an operation $o = (op, a, v)$ is *load-consistent* with respect to RAM $\mathbf{T}$ if $v = \mathbf{T}[a]$ whenever o is a load operation (store operations are vacuously defined to be load-consistent). We formally define the notion of consistency below:

Definition 5.1 (Consistent Operations). *Let $n \in \mathbb{N}$ and $\mathbf{T}, \mathbf{T}' \in \mathsf{RAM}_{\mathcal{I},n}$ for some index set I. We say that a sequence of operations $\mathbf{o} = (o_1, \ldots, o_k) \in O_{\mathcal{I}}^k$ over I is* consistent *with RAM states $\mathbf{T}, \mathbf{T}'$ if for all $i \in [k]$, $\mathbf{T}_i = \mathsf{Upd}_{\mathcal{I}}(\mathbf{T}_{i-1}, o_i)$ and operation o_i is load-consistent with respect to $\mathbf{T}_{i-1}$. Here we assume $\mathbf{T}_0 = \mathbf{T}$ and $\mathbf{T}_k = \mathbf{T}'$.*

For $m, n \in \mathbb{N}$, let $\mathsf{LRAM}_{\mathcal{I},m,n}$ denote the language consisting of tuples $(\mathbf{T}, \mathbf{o}, \mathbf{T}')$ with $\mathbf{T}, \mathbf{T}' \in \mathsf{RAM}_{\mathcal{I},n}$ and $\mathbf{o} \in (O_{\mathcal{I}})^m$ such that $\mathbf{o}$ is consistent with $\mathbf{T}, \mathbf{T}'$. Next, we formalize the folklore technique of checking correctness of RAM operations using *address-ordered transcripts.*

5.2 Consistency Check via Transcripts

A *transcript* is time-stamped sequence of operations executed on a RAM. More formally, given a RAM $\mathbf{T} = (\mathbf{a}, \mathbf{v}) \in \mathsf{RAM}_{\mathcal{I},n}$, operation sequence $\mathbf{o} = (o_1, \ldots, o_m)$ with $o_i = (\bar{op}_i, \bar{a}_i, \bar{v}_i) \in O_{\mathcal{I}}$ and RAM $\mathbf{T}' = (\mathbf{a}', \mathbf{v}') \in \mathsf{RAM}_{\mathcal{I},n}$, the *time ordered transcript* for the tuple $(\mathbf{T}, \mathbf{o}, \mathbf{T}')$ is given by the table tr with $k = 2n + m$ rows and four columns $\mathrm{tr} = (\mathbf{t}, \mathbf{op}, \mathbf{A}, \mathbf{V})$ defined as follows: (i) $\mathbf{t} = I_k = (1, \ldots, k)$, (ii) $\mathbf{op} = 0^n||(\bar{op}_1, \ldots, \bar{op}_m)||0^n$, (iii) $\mathbf{A} = \mathbf{a}||(\bar{a}_1, \ldots, \bar{a}_m)||\mathbf{a}'$ and (iv) $\mathbf{V} = \mathbf{v}||(\bar{v}_1, \ldots, \bar{v}_m)||\mathbf{v}'$. The i^{th} row of the table tr is (t_i, op_i, A_i, V_i) for $i \in [k]$. The first n records in tr correspond to the contents of $\mathbf{T}$, the next m records correspond to the operations in $\mathbf{o}$ and final n records correspond to contents of $\mathbf{T}'$. The timestamp column $\mathbf{t}$ is added to order operations with the same index. Notationally, we write $\mathrm{tr} = \mathsf{TimeTr}(\mathbf{T}, \mathbf{o}, \mathbf{T}')$.

We call a transcript $\mathrm{tr} = (\mathbf{t}, \mathbf{op}, \mathbf{A}, \mathbf{V})$ to be *address ordered* if $A_i \leq A_{i+1}$ for $i \in [k-1]$ and $t_i < t_{i+1}$ whenever $A_i = A_{i+1}$. For a transcript $\mathrm{tr} = (\mathbf{t}, \mathbf{op}, \mathbf{A}, \mathbf{V})$ with k records and a permutation $\sigma : [k] \to [k]$, we use $\sigma(\mathrm{tr})$ to denote the transcript $(\sigma(\mathbf{t}), \sigma(\mathbf{op}), \sigma(\mathbf{A}), \sigma(\mathbf{V}))$ obtained by permuting the records of tr according to the permutation σ. An address ordered transcript for tuple $(\mathbf{T}, \mathbf{o}, \mathbf{T}')$ is defined as $\mathrm{tr}^* = \sigma(\mathrm{tr})$ where $\mathrm{tr} = \mathsf{TimeTr}(\mathbf{T}, \mathbf{o}, \mathbf{T}')$ and σ is a permutation such that tr^* is address ordered. We denote it by $\mathrm{tr}^* = \mathsf{AddrTr}(\mathbf{T}, \mathbf{o}, \mathbf{T}')$. We say that an address ordered transcript $\mathrm{tr} = (\mathbf{t}, \mathbf{op}, \mathbf{A}, \mathbf{V})$ satisfies *load-store correctness* if for all pairs of consecutive records (t_i, op_i, A_i, V_i) and $(t_{i+1}, op_{i+1}, A_{i+1}, V_{i+1})$ we have $V_{i+1} = V_i$ whenever $op_{i+1} = 0$ (load operation) and $A_i = A_{i+1}$, i.e, a load operation does not change the value at an index. We formally state the folklore technique for enforcing memory consistency in our setting.

Lemma 5.1. *Let $\mathbb{F}$ be a finite field, $m, n \in \mathbb{N}$ be positive integers and $I \subseteq \mathbb{F}$. Then $(\mathbf{T}, \mathbf{o}, \mathbf{T}') \in \mathsf{LRAM}_{\mathcal{I},n,m}$ if and only if the address ordered transcript $\mathrm{tr}^* = \mathsf{AddrTr}(\mathbf{T}, \mathbf{o}, \mathbf{T}')$ satisfies load-store correctness.*

The consistency check in Lemma 5.1 can be encoded as an arithmetic circuit of size $\widetilde{O}(m+n)$, thus yielding an argument of knowledge for the language $\mathsf{LRAM}_{\mathcal{I},n,m}$ with prover complexity quasi-linear in $m + n$. For completeness, we present a self-contained argument of knowledge for $\mathsf{LRAM}_{\mathcal{I},m,m}$ $(m = n)$ based on the "polynomial protocol" framework defined in [26].

6 IMPROVED BATCHING-EFFICIENT RAM

We now detail the steps required to realize batching efficient RAM outlined in the technical overview.

6.1 Committed Index Lookup

In this section, we "lift" any committed sub-vector argument to a committed index lookup argument, where the latter makes a black-box use of the former. We use the trick of random linear combination of vectors to infer indexed lookup relation among them from sub-vector relation over the aggregated vectors. Similar use of random linear combinations has been made in the context of proving permutations in literature (e.g. [18]).

Lemma 6.1. *Let $\mathbf{t} \in \mathbb{F}^n$ and let $\mathbf{a}, \mathbf{v} \in \mathbb{F}^m$ for some positive integers m, n. Let $\mathbf{I}_n$ denote the vector $(1, \ldots, n)$. Then for $\gamma \leftarrow \mathbb{F}$, $(\mathbf{v} + \gamma \mathbf{a}) \preceq (\mathbf{t} + \gamma \mathbf{I}_n)$ implies $\mathbf{v} = \mathbf{t}[\,\mathbf{a}\,]$ except with probability $mn/|\mathbb{F}|$.*

Proof. We define vectors of linear polynomials $\mathbf{p} = (p_1, \ldots, p_m)$ and $\mathbf{q} = (q_1, \ldots, q_n)$ where $p_i(X) = v_i + a_i X$, $i \in [m]$ and $q_i(X) = t_i + iX$, $i \in [n]$. Now, we see that $\mathbf{v} = \mathbf{t}[\,\mathbf{a}\,]$ if and only if $\mathbf{p} \preceq \mathbf{q}$. For $\gamma \in F$, let $\mathbf{p}_\gamma$ and $\mathbf{q}_\gamma$ denote the vectors $(p_1(\gamma), \ldots, p_m(\gamma))$ and $(q_1(\gamma), \ldots, q_n(\gamma))$ respectively. It is obvious that $\mathbf{p} \preceq \mathbf{q}$ implies $\mathbf{p}_\gamma \preceq \mathbf{q}_\gamma$ for all $\gamma \in \mathbb{F}$. Using Schwartz-Zippel Lemma, it can also be seen that $\Pr_{\gamma \leftarrow \mathbb{F}}[\mathbf{p} \npreceq \mathbf{q} \,|\, \mathbf{p}_\gamma \preceq \mathbf{q}_\gamma] \leq mn/|\mathbb{F}|$. The bound follows from the observation that the event occurs only when γ is a common root of at least one pair of linear polynomials $\{(p_i(X), q_j(X)) : i \in [m], j \in [n]\}$. □

In Figure 2, we invoke Lemma 6.1 to construct a committed index lookup argument using a committed sub-vector argument $(\mathcal{P}_{\mathsf{sv}}, \mathcal{V}_{\mathsf{sv}})$. We formally state the following lemma, whose proof essentially follows from Lemma 6.1.

Lemma 6.2. *Assuming that $(\mathcal{P}_{\mathsf{sv}}, \mathcal{V}_{\mathsf{sv}})$ is an argument of knowledge for the relation $\mathcal{R}^{\mathsf{subvec}}_{\mathsf{srs},N,m}$ in the AGM, the interactive protocol in Figure 2 is an argument of knowledge for the relation $\mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m}$ in the AGM.*

6.2 Almost Identical RAM States

For a vector $\mathbf{a} \in [N]^m$, let $\mathsf{uniq}(\mathbf{a}) = \{a_i : i \in [m]\}$ denote the subset of unique values in $\mathbf{a}$. We call two RAM states $\mathbf{T}, \mathbf{T}' \in \mathbb{F}^N$ to be $\mathbf{a}$-*identical* if $\mathbf{T}[i] = \mathbf{T}'[i]$ for all $i \notin \mathsf{uniq}(\mathbf{a})$. As before, let

Common Input: srs, c_t, c_a, c_v, $c_I = [I(X)]_1$ where $I(X) = \mathsf{Enc}_{\mathbb{H}}(\mathbf{I})$ encodes the vector $\mathbf{I} = (1, \dots, N) \in \mathbb{F}^N$.
Prover's Input: Vectors $\mathbf{t} \in \mathbb{F}^N$, $\mathbf{a}, \mathbf{v} \in \mathbb{F}^m$.

1. $\mathcal{V}$ sends $\gamma \leftarrow \mathbb{F}$.
2. $\mathcal{P}$ and $\mathcal{V}$ compute: $\tilde{c}_t = \gamma c_I + c_t$, $\tilde{c}_v = \gamma c_a + c_v$.
3. $\mathcal{P}$ computes: $\tilde{\mathbf{t}} = \gamma \mathbf{I} + \mathbf{t}$, $\tilde{\mathbf{v}} = \gamma \mathbf{a} + \mathbf{v}$.
4. $\mathcal{P}$ and $\mathcal{V}$ run sub-vector argument $(\mathcal{P}_{\mathsf{sv}}, \mathcal{V}_{\mathsf{sv}})$ with $(\mathsf{srs}, \tilde{c}_t, \tilde{c}_v)$ as the common input and $(\tilde{\mathbf{t}}, \tilde{\mathbf{v}})$ as $\mathcal{P}_{\mathsf{sv}}$'s input.
5. $\mathcal{V}$ outputs $b \leftarrow \langle \mathcal{P}_{\mathsf{sv}}(\tilde{\mathbf{t}}, \tilde{\mathbf{v}}), \mathcal{V}_{\mathsf{sv}} \rangle(\mathsf{srs}, \tilde{c}_t, \tilde{c}_v)$.

Figure 2: Committed Index Lookup Argument

$T(X), T^*(X)$ and $a(X)$ be polynomials encoding the vectors $\mathbf{T}, \mathbf{T}'$ (over $\mathbb{H}$) and $\mathbf{a}$ (over $\mathbb{V}$). Let c_T, c'_T and c_a be the commitments to vectors $\mathbf{T}, \mathbf{T}'$ and $\mathbf{a}$ respectively in the group $\mathbb{G}_1$. The polynomial protocol to prove that $\mathbf{T}, \mathbf{T}' \in \mathbb{F}^N$ and $\mathbf{a} \in \mathbb{F}^m$ are $\mathbf{a}$-identical requires proving the relation $Z_I(X)(T(X) - T^*(X)) = 0$ over the set $Z_{\mathbb{H}}$ where $I = \mathsf{uniq}(\mathbf{a})$ and $Z_I(X) = \prod_{i \in I}(X - \xi^i)$ is the vanishing polynomial for the set $\mathbb{H}_I = \{\xi^i : i \in I\}$. To proceed, the honest prover commits to polynomial $Z_I(X)$ and proves (i) $Z_I(X) \cdot (T(X) - T^*(X)) = 0 \bmod Z_{\mathbb{H}}$ and (ii) the zeroes of $Z_I(X)$ form a subset of zeroes of $\mathbb{H}_I(X)$ as defined. Together, the two conditions imply that $T(\xi^i) = T^*(\xi^i)$ for $i \notin \mathsf{uniq}(\mathbf{a})$. To prove the first relation, the prover computes the polynomial $D(X)$ as below:

$$\begin{aligned} D(X) &= \frac{(T(X) - T^*(X)) \cdot Z_I(X)}{Z_{\mathbb{H}}(X)} \\ &= \sum_{i \in I} \frac{(T(\xi^i) - T^*(\xi^i))\mu_i(X)}{Z_{\mathbb{H}}(X)} Z_I(X) \end{aligned}$$

Substituting, $\Delta_i = T(\xi^i) - T^*(\xi^i)$, $\mu_i(X) = Z_{\mathbb{H}}(X)/(Z'_{\mathbb{H}}(\xi^i)(X - \xi^i))$

$$= \sum_{i \in I} \frac{\Delta_i}{Z'_{\mathbb{H}}(\xi^i)} \left(\frac{Z_I(X)}{X - \xi^i} \right) = \sum_{i \in I} \frac{\Delta_i Z'_I(\xi^i)}{Z'_{\mathbb{H}}(\xi^i)} \kappa_i(X) \qquad (1)$$

In the above, the summation only runs over indices in I, as $\Delta i = 0$ for $i \notin I$. In the final equality, we use $\kappa_i(X) = Z_I(X)/(Z'_I(\xi^i)(X - \xi^i))$ for $i \in I$ which we recognize as the lagrange basis polynomials for the set $\{\xi^i : i \in I\}$. Thus, Equation (1) implies that $D(X)$ is at most degree $|I| - 1$ polynomial, with $D(\xi^i) = \Delta_i Z'_I(\xi^i)/Z'_{\mathbb{H}}(\xi^i)$ for $i \in I$. The prover can therefore interpolate $D(X)$ (in power basis) in $O(|I| \log^2 |I|)$ $\mathbb{F}$-operations and compute $[D(X)]_1$ in $O(|I|)$ $\mathbb{G}_1$-operations. The prover sends the commitment $[D(X)]_1$ to the verifier. Finally, the verifier can check the identity $Z_I(X) \cdot (T(X) - T^*(X)) = D(X) \cdot Z_{\mathbb{H}}(X)$ by a pairing check. For this, since the tables are committed in $\mathbb{G}_1$, prover will need to send $[Z_I(X)]_2$.

Next, the prover needs to show that zeroes of Z_I are indeed in the set $\mathbb{H}_I = \{\xi^i : i \in I\} = \{\xi^{a_i} : i \in [m]\}$. Clearly, it suffices to show that $Z_I(X)$ divides the polynomial $\prod_{i \in [m]}(X - \xi^{a_i})$. To obtain a polynomial protocol, the prover commits to an auxiliary polynomial $h(X) = \sum_{i=1}^m \xi^{a_i} \tau_i(X)$ which interpolates the vector $\mathbf{h} = (\xi^{a_1}, \dots, \xi^{a_m})$. The correctness of h polynomial can be established by showing that the interpolated vector $\mathbf{h}$ satisfies committed index lookup relation $\mathbf{h} = \mathbf{T}_{\mathsf{exp}}[\,\mathbf{a}\,]$ where $\mathbf{T}_{\mathsf{exp}} = (\xi^1, \dots, \xi^N)$. Moreover, we notice that the polynomial interpolating the table $\mathbf{T}_{\mathsf{exp}}$ is particularly simple, i.e, $T_{\mathsf{exp}}(X) = X$, and thus the setup need not be augmented with table-specific parameters for $\mathbf{T}_{\mathsf{exp}}$. Finally, it remains to show that $Z_I(X)$ divides $K(X) = \prod_{i=1}^m (X - h(v^i))$. To do so, the prover commits to $K(X)$ and the quotient polynomial $q(X) = K(X)/Z_I(X)$. The verifier checks the polynomial identities at α, i.e $K(\alpha) = q(\alpha)Z_I(\alpha)$ and $K(\alpha) = \prod_{i=1}^m (\alpha - h(v^i))$. The former is easily accomplished using evaluation proofs for K, q and Z_I at α. For checking the latter, the prover commits to another polynomial $u(X)$ satisfying $u(v^i) = \prod_{j=1}^{i-1} ((\alpha - h(v^j))/(1 + \beta\tau_1(v^j)))$ for $i \in [m]$ where $\beta = K(\alpha) - 1$. The verifier ensures the correctness of $u(X)$ by checking:

$$\begin{aligned} \tau_1(X)(u(X) - 1) &= 0 \bmod Z_{\mathbb{V}} \\ u(vX)(1 + \beta\tau_1(X)) - u(X)(\alpha - h(X)) &= 0 \bmod Z_{\mathbb{V}}. \end{aligned} \qquad (2)$$

We prove that the above constraints imply that $K(\alpha) = \prod_{i \in [m]}(\alpha - h(v^i))$ in Lemma 6.3. Note that in this protocol we require commitment to the polynomial Z_I in both $\mathbb{G}_1$ and $\mathbb{G}_2$, and thus another pairing check is required to show that the $Z_I(X)$ committed in $\mathbb{G}_1$ is the same as the $Z_I(X)$ committed in $\mathbb{G}_2$ (used for the real pairing check). The complete protocol for checking that RAMs $\mathbf{T}$ and $\mathbf{T}'$ are identical outside indices in $\mathbf{a}$ is given in Figure 3.

Lemma 6.3. *There exists a polynomial $u(X) \in \mathbb{F}[X]$ satisfying the identities in Equation (2) if and only if $K(\alpha) = 1 + \beta = \prod_{i \in [m]}(\alpha - h(v^i))$.*

Proof. Assume that the identitites hold for some polynomial $u(X)$. The first identity implies $u(v) = 1$. From the second identity, we conclude that for all $i \in [m]$, we have $u(v^{i+1}) = u(v^i) \cdot ((\alpha - h(v^i))/(1 + \beta\tau_1(v^i)))$, and thus:

$$1 = u(v^{m+1})/u(v) = \prod_{i \in [m]} \left(\frac{\alpha - h(v^i)}{1 + \beta\tau_1(v^i)} \right).$$

We observe that the product of denominators in the above equation is simply $1 + \beta$ as $\tau_1(v^i)$ is 0 for all $i \neq 1$, and thus $1 + \beta = \prod_{i=1}^m (\alpha - h(v^i))$. In the other direction, it is easy to check that $u(X)$ as defined for an honest prover, satisfies the identities in Equation 2. ☐

6.3 Batching-Efficient RAM: Combined Protocol

We put the entire protocol together now. Let I denote the set of indices $\{1, \dots, N\}$, and $\mathbf{I}_N$ denote the vector $(1, \dots, N)$. We formally define the committed RAM relation for which we present an argument of knowledge in this section.

Definition 6.1. *We define the* committed ram *relation* $\mathcal{R}^{\mathsf{ram}}_{\mathsf{srs},N,m}$ *to consist of tuples* $((c_T, c'_T, c_{op}, c_a, c_w), (\mathbf{T}, \mathbf{T}', \mathbf{op}, \mathbf{a}, \mathbf{w}))$ *such that:*

- $(\mathbf{T}, \mathbf{o}, \mathbf{T}') \in \mathsf{LRAM}_{I,N,m}$ *for* $\mathbf{o} = (o_1, \dots, o_m)$ *where we have* $o_i = (op_i, a_i, w_i) \in O_I$ *for all* $i \in [m]$ *(here we implicitly view vectors* $\mathbf{T}$ *and* $\mathbf{T}'$ *as RAMs with index column* I_N*).*
- $c_T = \mathsf{KZG.Commit}(\mathsf{srs}, T(X))$, $c'_T = \mathsf{KZG.Commit}(\mathsf{srs}, T^*(X))$, $c_{op} = \mathsf{KZG.Commit}(\mathsf{srs}, op(X))$, $c_a = \mathsf{KZG.Commit}(\mathsf{srs}, a(X))$, $c_w = \mathsf{KZG.Commit}(\mathsf{srs}, w(X))$ *where polynomials* $T(X), T^*(X)$ *encode vectors* $\mathbf{T}, \mathbf{T}'$ *over* $\mathbb{H}$*, while* $op(X), a(X)$ *and* $w(X)$ *encode vectors* $\mathbf{op} = (op_1, \dots, op_m)$, $\mathbf{a}$ *and* $\mathbf{w}$ *over* $\mathbb{V}$.

As outlined in the blueprint, the prover first commits to "smaller" RAMs $\mathbf{S} = (\mathbf{a}, \mathbf{v})$ and $\mathbf{S}' = (\mathbf{a}, \mathbf{v}')$ where $\mathbf{v} = \mathbf{T}[\mathbf{a}]$ and $\mathbf{v}' = \mathbf{T}'[\mathbf{a}]$. The prover commits to $\mathbf{S}$ and $\mathbf{S}'$ by sending commitments c_v and

Common Input: srs, c_T, c'_T, c_a.
Prover's Input: Vectors $\mathbf{T}, \mathbf{T}' \in \mathbb{F}^N$, $\mathbf{a} \in \mathbb{F}^m$. Polynomials $T(X), T^*(X)$ and $a(X)$ encoding $\mathbf{T}, \mathbf{T}'$ and $\mathbf{a}$ respectively.

Round 1: Prover commits to auxiliary polynomials
1. $\mathcal{P}$ computes:
   - $I = \mathsf{uniq}(\mathbf{a})$, $Z_I(X) = \prod_{i \in I}(X - \xi^i)$.
   - $D(X) = Z_I(X)(T(X) - T^*(X))/Z_{\mathbb{H}}(X)$.
   - $h(X)$ such that $h(\nu^i) = \xi^{a_i}$ for $i \in [m]$.
   - $K(X) = \prod_{i=1}^m (X - h(\nu^i))$, $q(X) = K(X)/Z_I(X)$.
2. $\mathcal{P}$ sends $c_z = [Z_I(X)]_1$, $c'_z = [Z_I(X)]_2$, $c_d = [D(X)]_1$, $c_h = [h(X)]_1$, $c_k = [K(X)]_1$, $c_q = [q(X)]_1$.
3. $\mathcal{V}$ sends $\alpha \leftarrow \mathbb{F}$.

Round 2: Prover commits to polynomial $u(X)$.
1. $\mathcal{P}$ sets $\beta = K(\alpha) - 1$ and interpolates $u(X)$ on $\mathbb{V}$ such that $u(\nu^i) = \prod_{j=1}^{i-1}\left((\alpha - h(\nu^j))/(1 + \beta\tau_1(\nu^j))\right)$ for $i \in [m]$.
2. $\mathcal{P}$ sends $c_u = [u(X)]_1$.
3. $\mathcal{V}$ sends $r \leftarrow \mathbb{F}$.

Round 3: Prover batches checks in Eq (2).
1. $\mathcal{P}$ computes:
$$Q(X) = \big(u(\nu X)(1 + \beta\tau_1(X)) - u(X)(\alpha - h(X)) + r\tau_1(X)(u(X) - 1)\big)/Z_{\mathbb{V}}(X)$$
2. $\mathcal{P}$ sends $c_Q = [Q(X)]_1$.
3. $\mathcal{V}$ sends $s \leftarrow \mathbb{F}$.

Round 4: Prover sends evaluations.
1. $\mathcal{P}$ sends $\langle z\rangle_\alpha = Z_I(\alpha)$, $\langle q\rangle_\alpha = q(\alpha)$, $\langle K\rangle_\alpha = K(\alpha)$, $\langle Q\rangle_s = Q(s)$, $\langle u\rangle_s = u(s)$, $\langle u\rangle_{\nu s} = u(\nu s)$, $\langle h\rangle_s = h(s)$.
2. $\mathcal{V}$ sends $r_\alpha, r_s \leftarrow \mathbb{F}$.

Round 5: Prover batches evaluation proofs.
1. Compute:
   - $p_\alpha(X) = Z_I(X) + r_\alpha q(X) + r_\alpha^2 K(X)$.
   - $p_s(X) = Q(X) + r_s u(X) + r_s^2 h(X)$.
   - $\Pi_\alpha = \mathsf{KZG.Prove}(\mathsf{srs}, p_\alpha, \alpha)$.
   - $\Pi_s = \mathsf{KZG.Prove}(\mathsf{srs}, p_s, s)$, $\Pi_{\nu s} = \mathsf{KZG.Prove}(\mathsf{srs}, u, \nu s)$.
2. Send $\Pi_\alpha, \Pi_s, \Pi_{\nu s}$.

Round 6: Verifier checks identities.
1. $\mathcal{V}$ computes $[p_\alpha]_1 = c_z + r_\alpha c_q + r_\alpha^2$, $[p_z]_1 = c_Q + r_s c_u + r_s^2 c_h$.
2. $\mathcal{V}$ checks:
   - $\langle z\rangle_\alpha \cdot \langle q\rangle_\alpha = \langle K\rangle_\alpha$.
   - $\langle u\rangle_{\nu s}(1 + \beta\tau_1(s)) - \langle u\rangle_s(\alpha - \langle h\rangle_s) + r\tau_1(s)(\langle u\rangle_s - 1) = \langle Q\rangle_s Z_{\mathbb{V}}(s)$.
   - $e(c_T - c'_T, c'_z) = e(c_d, [Z_{\mathbb{H}}(X)]_2)$.
   - $e([1]_1, c'_z) = e(c_z, [1]_2)$.
   - $\mathsf{KZG.Verify}(\mathsf{srs}, [p_\alpha]_1, \langle z\rangle_\alpha + r_\alpha\langle q\rangle_\alpha + r_\alpha^2\langle K\rangle_\alpha, \alpha, \Pi_\alpha)$.
   - $\mathsf{KZG.Verify}(\mathsf{srs}, [p_z]_1, \langle Q\rangle_s + r_s\langle u\rangle_s + r_s^2\langle K\rangle_s, s, \Pi_s)$.
   - $\mathsf{KZG.Verify}(\mathsf{srs}, c_u, \langle u\rangle_{\nu s}, \nu s, \Pi_{\nu s})$.

Round 7: Check correctness of polynomial h.
1. $\mathcal{P}$ and $\mathcal{V}$ execute committed index lookup argument (Fig 2) to check $([X]_1, c_a, c_h) \in \mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m}$.
2. $\mathcal{V}$ accepts if the above argument accepts and all the preceding checks succeed.

Figure 3: Argument for showing RAMs are identical outside small set of indices.

c'_v to $\mathbf{v}$ and $\mathbf{v}'$. Then the prover and verifier execute the committed index lookup protocol to prove:

$$(c_T, c_a, c_v) \in \mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m} \wedge (c'_T, c_a, c'_v) \in \mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m} \tag{3}$$

The verifier uses a random challenge $\chi \leftarrow \mathbb{F}$ to reduce two instances of $\mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m}$ to one instance $(c_T + \chi c'_T, c_a, c_v + \chi c'_v) \in \mathcal{R}^{\mathsf{lookup}}_{\mathsf{srs},N,m}$. Then, we show that RAMs $\mathbf{T}$ and $\mathbf{T}'$ are $\mathbf{a}$-identical using the protocol in Figure 3, described in Section 6.2. All that remains is to prove is that the operation sequence $\mathbf{o}$ is consistent with small RAMs $\mathbf{S}$ and $\mathbf{S}'$. We check this using the argument in Appendix B [19], which is obtained by compiling the polynomial protocol for RAM in Appendix C [19] into an argument of knowledge in the AGM. Specifically, the prover and the verifier set $c_S = (c_a, c_v)$, $c'_S = (c_a, c'_v)$ and $c_o = (c_{op}, c_a, c_w)$, and execute the argument of knowledge for showing $(c_S, c_o, c'_S) \in \mathcal{R}^{\mathsf{LRAM}}_{\mathsf{srs},m}$ (see Definition C.1 [19]). We provide the complete protocol listing in Figure 4. The protocol in Figure 4 assumes pre-computed parameters for the tables $\mathbf{T}$ and $\mathbf{T}'$. The maintenance of these pre-computed parameters in the presence of updates is detailed in Section 7.

Theorem 6.1. *The protocol in Figure 4 is a succinct argument of knowledge for the relation $\mathcal{R}^{\mathsf{ram}}_{\mathsf{srs},N,m}$ in the AGM, under the Q-DLOG assumption for the bilinear group $(\mathbb{F}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$.*

7 FAST LOOKUPS FROM APPROXIMATE PRE-PROCESSING

In this section, we provide details of the algorithm to construct lookup argument for a table $\mathbf{T}$, using pre-computed parameters of a table which is a small hamming distance away. The dependence on pre-computed parameters in several recent lookup arguments such as [20, 33, 38, 39] stems from the need to compute an encoded quotient of the form:

$$[Q]_g = \sum_{i \in I} c_i \left[\frac{T(X) - T(\xi^i)}{X - \xi^i}\right]_g \tag{4}$$

for some $O(m)$ sized set I. The quotient in Equation (4) can be computed in $O(m)$ cost when the quotients $\left[(T(X) - T(\xi^i))/(X - \xi^i)\right]_g$ are available for all $i \in [N]$. In this section we exhibit an algorithm which computes the above with $O((m+\delta)\log^2(m+\delta))$ cost, given access to similar quotients for a table at hamming distance δ from $\mathbf{T}$. We now describe our approach.

7.1 Base + Cache approach

The key idea we employ is to express the current table $\mathbf{T} \in \mathbb{F}^N$ as $\mathbf{T}_\mathsf{b} + \mathbf{T}_\mathsf{ch}$, where $\mathbf{T}_\mathsf{b}$ is the table for which we assume that the encoded quotients are available (via the $O(N\log N)$ computation), and $\mathbf{T}_\mathsf{ch}$ captures the changes to the table since. We will periodically update (say after s batch updates) $\mathbf{T}_\mathsf{b}$ to current table state, and re-compute all the quotients (we call it the *offline* phase). We will revisit the question on choosing s optimally later. Let $I \subseteq [N]$ denote the set of indices in the current batch of m lookups. The *online* phase of our proof generation involves computing the sum in Equation (4) for the table $\mathbf{T}$. The following Theorem determines the efficiency of the online phase of our prover.

Setup $(1^\lambda, N, m, \mathbf{T}, \mathbf{T}')$:

- srs $= (\{[\tau^i]_1\}_{i=0}^N, \{[\tau^i]_2\}_{i=0}^N)$ for $\tau \leftarrow \mathbb{F}$.
- $W_2^i = [Z_\mathbb{H}(X)/(X-\xi^i)]_2$, $i \in [N]$ (needed by prover).
- $[Z_\mathbb{H}(X)]_1, [Z_\mathbb{H}(X)]_2$ (known to both $\mathcal{P}$ and $\mathcal{V}$).

Precompute $(\mathbf{T}, \mathbf{T}')$:

- $W_1^i = [(T(X) - T(\xi^i))/(X-\xi^i)]_2$, $i \in [N]$,
- $W_1^{i\prime} = [(T^*(X) - T^*(\xi^i))/(X-\xi^i)]_2$, $i \in [N]$.

Common Input: srs, $c_T, c'_T, c_{op}, c_a, c_w \in \mathbb{G}_1$.
Prover's Input: Vectors $\mathbf{T}, \mathbf{T}', \mathbf{op}, \mathbf{a}, \mathbf{w}$ and their encoding polynomials.

Round 1: Commit to sub RAMs.

1. $\mathcal{P}$ computes $\mathbf{v} = \mathbf{T}[\,\mathbf{a}\,]$, $\mathbf{v}' = \mathbf{T}'[\,\mathbf{a}\,]$ and the encoding polynomials $v(X)$ and $v^*(X)$.
2. $\mathcal{P}$ sends $c_v = [v(X)]_1$, $c'_v = [v^*(X)]_1$.
3. $\mathcal{V}$ sends $\chi \leftarrow \mathbb{F}$.

Round 2: Execute committed index lookup.

1. $\mathcal{P}$ and $\mathcal{V}$ compute $\hat{c}_T = c_T + \chi c'_T$, $\hat{c}_v = c_v + \chi c'_v$.
2. $\mathcal{P}$ computes $\hat{\mathbf{T}} = \mathbf{T} + \chi\mathbf{T}'$, $\hat{\mathbf{v}} = \mathbf{v} + \chi\mathbf{v}'$.
3. $\mathcal{P}$ and $\mathcal{V}$ execute committed index lookup argument in Fig 2, with $(\hat{c}_T, c_a, \hat{c}_v)$ as the common input and $(\hat{\mathbf{T}}, \mathbf{a}, \hat{\mathbf{v}})$ as prover's input.

Round 3: Prove RAMs are $\mathbf{a}$-identical.

1. $\mathcal{P}$ and $\mathcal{V}$ execute argument in Fig 3 with common input (c_T, c'_T, c_a) and prover's input as $(\mathbf{T}, \mathbf{T}', \mathbf{a})$.

Round 4: Prove sub RAMs are memory-consistent with update.

1. $\mathcal{P}$ and $\mathcal{V}$ execute argument in Fig 11 [19] to check $(c_S, c_o, c'_S) \in \mathcal{R}^{\mathsf{LRAM}}_{\mathsf{srs},m}$ with $c_S = (c_a, c_v)$, $c'_S = (c_a, c'_v)$ and $c_o = (c_{op}, c_a, c_w)$.
2. $\mathcal{V}$ accepts if all sub-protocols accept.

Figure 4: Our batching-efficient RAM protocol

Theorem 7.1. *Let N, ξ be as defined previously. Given* KZG *proofs $\{W_i\}_{i=1}^N$ with $W_i = [T_\text{b}(X) - T_\text{b}(\xi^i)/(X-\xi^i)]_g$, where $T_\text{b}(X) = \mathsf{Enc}_\mathbb{H}(\mathbf{T}_\text{b})$ encodes a vector $\mathbf{T}_\text{b} \in \mathbb{F}^N$, for any $I \subseteq [N]$, there exists an algorithm to compute $[Q]_g$ as given in Equation (4) for polynomial $T(X) = \mathsf{Enc}_\mathbb{H}(\mathbf{T})$ encoding the vector $\mathbf{T} \in \mathbb{F}^N$ using $O((\delta + |I|)\log^2(\delta+|I|))$ $\mathbb{F}$-operations and $O(\delta + |I|)$ $\mathbb{G}$-operations. Here, δ denotes the hamming distance between vectors $\mathbf{T}_\text{b}$ and $\mathbf{T}$.*

Proof. Let $\mathbf{T} = \mathbf{T}_\text{b} + \mathbf{T}_\text{ch}$ and thus $T(X) = T_\text{b}(X) + T_\text{ch}(X)$. Define $K = I \cup \{j \in [N] : \mathbf{T}_\text{ch}[\,j\,] \neq 0\}$ as a set which captures the indices where the current table $\mathbf{T}$ differs from the base $\mathbf{T}_\text{b}$, where we explicitly also include the lookup indices I in K. For $j \in K$, let $\mathbf{T}_\text{ch}[j] = \Delta t_j$. Then $T_\text{ch}(X) = \sum_{j\in K} \Delta t_j \mu_j(X)$. We write the quotient $Q(X)$ as:

$$Q(X) = \sum_{i\in I} c_i \left(\frac{T_\text{b}(X) - T_\text{b}(\xi^i)}{X-\xi^i}\right) + \sum_{i\in I} c_i \left(\frac{T_\text{ch}(X) - T_\text{ch}(\xi^i)}{X-\xi^i}\right)$$

From above, we have $[Q(x)]_g = [Q_\text{b}(x)]_g + [Q_\text{ch}(x)]_g$ where

$$Q_\text{b}(X) = \sum_{i\in I} c_i (T_\text{b}(X) - T_\text{b}(\xi^i))/(X-\xi^i)$$
$$Q_\text{ch}(X) = \sum_{i\in I} c_i (T_\text{ch}(X) - T_\text{ch}(\xi^i))/(X-\xi^i)$$

We can compute $[Q_\text{b}(X)]_g$ from the pre-computed KZG openings of $T_\text{b}(X)$ at points $\xi^i, i \in I$ using $O(|I|)$ group operations and $O(|I|\log^2|I|)$ field operations. Therefore, it suffices to compute $[Q_\text{ch}(X)]_g$ efficiently. Using $T_\text{ch}(X) = \sum_{j\in K}\Delta t_j \mu_j(X)$ we write $Q_\text{ch}(X)$ as linear combination of table-independent polynomials:

$$\begin{aligned} Q_\text{ch}(X) &= \sum_{i\in I} c_i \sum_{j\in K} \Delta t_j \frac{\mu_j(X) - \mu_j(\xi^i)}{X-\xi^i} \\ &= \sum_{i\in I} c_i \Delta t_i \frac{\mu_i(X) - 1}{X-\xi^i} + \sum_{i\in I}\sum_{j\in K\setminus\{i\}} c_i \Delta t_j \frac{\mu_j(X)}{X-\xi^i} \end{aligned}$$

Now, we can write $[Q_\text{ch}(X)]_g = [Q^{(1)}_\text{ch}(X)]_g + [Q^{(2)}_\text{ch}(X)]_g$ where:

$$Q^{(1)}_\text{ch}(X) = \sum_{i\in I} c_i\Delta t_i \frac{\mu_i(X)-1}{X-\xi^i},\ Q^{(2)}_\text{ch}(X) = \sum_{i\in I}\sum_{j\in K\setminus\{i\}} c_i\Delta t_j \frac{\mu_j(X)}{X-\xi^i}$$

The term $\left[Q^{(1)}_\text{ch}(X)\right]_g$ can be computed using $O(|I|)$ group operations by augmenting the setup with pre-computed KZG opening proofs of polynomials $\mu_i(X)$ at ξ^i for $i \in [N]$. This adds $O(N)$ to the setup parameters, while the computation can be done in $O(N\log N)$ time with methods similar to existing pre-computed parameters. This eventually leaves us with $[Q^{(2)}_\text{ch}(X)]_g$. Next, we synthesize the polynomial $Q^{(2)}_\text{ch}(X)$ in a form that reduces group operations required to compute its encoding.

$$\begin{aligned} Q^{(2)}_\text{ch}(X) &= \sum_{i\in I} c_i \sum_{j\in K\setminus\{i\}} \Delta t_j \mu_j(X)/(X-\xi^i) \\ &= \sum_{i\in I} c_i \sum_{j\in K\setminus\{i\}} \frac{\Delta t_j}{Z'_\mathbb{H}(\xi^j)} \frac{Z_\mathbb{H}(X)}{(X-\xi^i)(X-\xi^j)} \\ &= N^{-1}\sum_{i\in I} c_i \sum_{j\in K\setminus\{i\}} \frac{\xi^j\Delta t_j}{\xi^i-\xi^j}\left(\frac{Z_\mathbb{H}(X)}{X-\xi^i} - \frac{Z_\mathbb{H}(X)}{X-\xi^j}\right) \\ &= N^{-1}\sum_{i\in I}\left(c_i \cdot \sum_{j\in K\setminus\{i\}} \frac{\xi^j\Delta t_j}{\xi^i-\xi^j}\right)\frac{Z_\mathbb{H}(X)}{X-\xi^i} \\ &\quad + N^{-1}\sum_{j\in K}\left(\xi^j\Delta t_j \cdot \sum_{i\in I\setminus\{j\}} \frac{c_i}{\xi^j-\xi^i}\right)\frac{Z_\mathbb{H}(X)}{X-\xi^j} \end{aligned} \quad (5)$$

In the first step, we substituted $\mu_j(X)$, while in the final step we re-arranged the summation to accumulate the scalar factor for each distinct polynomial of the form $Z_\mathbb{H}(X)/(X-\xi^i)$. Define scalars a_i, $i \in I$ and b_j, $j \in K$ as below:

$$a_i = \sum_{j\in K\setminus\{i\}} \frac{\xi^j\Delta t_j}{\xi^i-\xi^j}, i \in I \quad b_j = \sum_{i\in I\setminus\{j\}} \frac{c_i}{\xi^j-\xi^i}, j \in K \quad (6)$$

Now, define $W_3^j := [Z_\mathbb{H}(X)/(X-\xi^j)]_g$. We see that W_3^j is just the KZG opening proof of the polynomial $Z_\mathbb{H}(X)$ evaluated at ξ^j for $j \in [N]$. These can be precomputed one time and it adds $O(N)$ to the setup parameters and the computation can be done in $O(N \log N)$ time.

Now, we see that $[Q_{\mathrm{ch}}^{(2)}(X)]_g$ can be written as linear combination of $O(|K| + |I|)$ group elements.

$$\left[Q_{\mathrm{ch}}^{(2)}(X)\right]_g = N^{-1}\left(\sum_{i\in I}(c_i a_i)\cdot W_3^i + \sum_{j\in K}(\xi^j \Delta t_j b_j)\cdot W_3^j\right) \quad (7)$$

Now, c_i are known constants depending on the specific lookup scheme. So, given $\{a_i\}_{i\in I}, \{b_j\}_{j\in K}$, $\left[Q_{\mathrm{ch}}^{(2)}(X)\right]_g$ can be computed in $O(|I| + |K|)$ group operations. While we have diligently reduced the group operations, we still seem to need $O(|I||K|) = O(m\delta)$ field operations. We clearly need better than naive way of computing the scalars in (6) to obtain additive overhead in δ. This is what we consider next. Let $d_j := \xi^j \Delta t_j$. Then we have from Eq (6):

$$a_i = \sum_{j\in K\setminus\{i\}} \frac{d_j}{\xi^i - \xi^j}, i \in I \quad b_j = \sum_{i\in I\setminus\{j\}} \frac{c_i}{\xi^j - \xi^i}, j \in K \quad (8)$$

So, to compute a_i and b_j, it suffices to compute *reciprocal sums* efficiently. Our next lemma claims that such reciprocal sums can be computed efficiently.

Lemma 7.1. *Let $I \subset K \subset [N]$ and let a_i for all $i \in I$ and b_j for all $j \in K$ be as described above. Then, a_i for all $i \in I$ and b_j for all $j \in K$ can be computed in $O(|K|\log^2|K|)$ $\mathbb{F}$ operations.*

Proof-Sketch. We provide a very high-level overview of the proof idea for a_i. First, we mention that the special case of the lemma when $d_j = 1$ for all $j \in K$ admits an efficient computation due to the following identity proved in Lemma D.1 of the full version of the paper [19]. For $Z_K(X) = \prod_{i\in K}(X - \xi^i)$, we have

$$\frac{Z_K''(\xi^i)}{Z_K'(\xi^i)} = 2\sum_{j\in K\setminus\{i\}} \frac{1}{\xi^i - \xi^j}.$$

The polynomial Z_K can be computed in $O(|K|\log^2|K|)$ and its first two derivatives can also be evaluated on the set $\{\xi^i : i \in I\}$ with the same complexity. However, to deal with arbitrary values of d_j we need more ingenuity. We defer the detailed proof to the full version of the paper [19]. □

From Lemma 7.1, we conclude that the scalars $a_i, i \in I$ and $b_j, j \in K$ can be computed in time $O(|K|\log^2|K|)$, which proves the bound in Theorem 7.1.

□

7.2 Amortized Sublinear Batching

We now return to the question of how frequently should we run the offline phase to compute full parameters. For concrete analysis, let s be the period after which the rebasing takes place; i.e., after s batches of m operations each, we set the base table $\mathbf{T}_\mathrm{b}$ to the current table, setting $\mathbf{T}_{\mathrm{ch}} = \mathbf{0}$. At this point we also compute all encoded quotients for $\mathbf{T}_\mathrm{b}$ using the $O(N \log N)$ algorithm of [21]. Consider $\delta \leq ms$ as the upper-bound on δ, and distributing the cost of re-basing, the amortized overhead for the batch of m operations is: $O(ms\log^2(ms) + \frac{N\log N}{s})$ $\mathbb{F}$-operations and $O(ms + \frac{N\log N}{s})$ $\mathbb{G}$-operations. Ignoring the logarithmic factors, the cost is minimized by setting $s \approx \sqrt{N/m}$, resulting in amortized prover overhead of $\widetilde{O}(\sqrt{mN})$. We note that the above analysis considers the worst case scenario, where each update affects a distinct position in the table. In settings, where frequency of updates is non-uniform accross positions in the table (e.g, in the blockchain example, if bulk of transactions come from small number of clients), we may be able to defer the offline phase even longer. Same is also true for settings where updates to the table are infrequent.

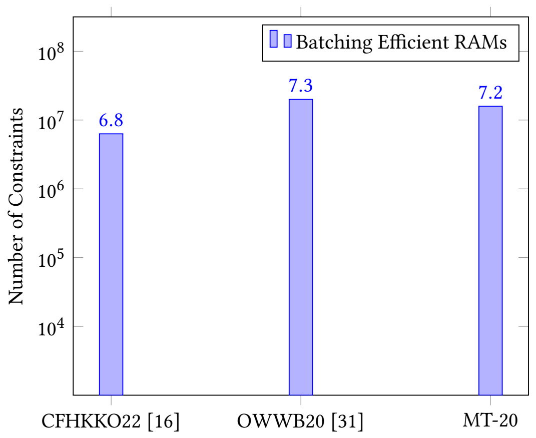

Figure 5: Comparison of R1CS constraints incurred by existing approaches for batching efficient RAMs. MT20 refers to Merkle-tree with depth 20, instantiated using Poseidon hash.

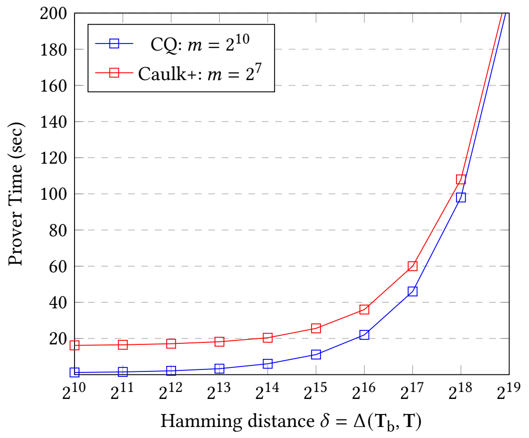

Figure 6: Online proving time of updates on table T given the pre-processing parameters for table $\mathbf{T}_\mathrm{b}$, plotted against the Hamming distance δ between $\mathbf{T}_\mathrm{b}$ and T. Here m denotes the batch size of updates, while the RAM size is $N = 2^{20}$. The blue plot corresponds to our scheme instantiated with CQ as the sub-vector argument (in a black-box manner). The red plot corresponds to the non-black-box adaptation of Caulk+ described in Appendix E of full version of the paper [19].

8 EXPERIMENTS

In this section we present a concrete evaluation of our batching efficient RAM and compare it to the prior works on batching efficient RAM [16, 31]. We also separately benchmark the effectiveness of our approach of computing encoded quotients presented in Section 7 over the naïve approach. Our implementation is built on top of existing implementation [1] of lookup argument Caulk [38]. We make our implementation available at https://anonymous.4open.science/r/updatableRAM/.

Experimental Test-bed. All the benchmarks were run single-threaded on a commodity configuration featuring a $2.1GHz$ Intel-I5 processor, 16 GB memory running Ubuntu Linux 22.04. The implementation was compiled using `-release` flag in Rust. Our protocol was instantiated for BLS12-381 curve, using the scheme CQ [20] as the underlying sub-vector argument.

Online Proof Generation. In Figure 6, we benchmark the time to prove a batch of $m = 2^{10}$ updates on a table of size $N = 2^{20}$. Here, the values on the x-axis denote the hamming distance between the table on which the updates are being proved and the table whose pre-processed parameters are used for the proof generation. Naturally, we expect the proving time to increase as the table becomes more distant from the one used to generate pre-processed parameters. Our proof generation time stays under a minute till the two table differ in almost 2×10^5 positions. In conjunction with offline pre-processing time from table 3, the graph in Figure 6 determines how the offline parameter generation should be scheduled to achieve optimal performance on average.

Offline Pre-Processing. In Table 3 we provide the time to compute table-specific parameters as a function of table size. This is the most computationally intensive step in our scheme involving FFTs over group polynomials as the main bottle-neck. We believe offline pre-processing can be made an order of magnitude more efficient by leveraging parallel implementation for the FFTs.

Table-Size (N)	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Time (s)	7	29	135	620	2766	12000

Table 3: Pre-processing time for tables of different sizes.

Proof Size and Verification Costs. Our proof sizes and verification costs are independent of the size of the table and the number of updates in a batch. For the instantiation of our scheme with BLS12-381 curve, we incur a proof size of $4.4KB$ while the verification takes around $15ms$.

Fast Update Benchmarks. We also benchmark the efficiency of the algorithm to compute scalar coefficients in Lemma 7.1, required to assemble the encoded quotients from pre-computed quotients. This algorithm is implemented and tested in the `fastupdate` module of the referenced repository. In Table 4, we compare it against the naive computation of the quotients. In the table, we vary the sizes of set I in Lemma 7.1 in the set $\{2^i : 7 \leq i \leq 10\}$ and set $K = 2^7|I|$. We clearly see $5 \times -20\times$ advantage over the naive computation.

Continuity. To maintain *continuity* of the system (this is particularly important for applications such as rollups), we must carefully align the online prover performance curve with the cost of offline computation. As an example, suppose $\mathbf{T}_0$ is the initial pre-processed table at time $t = 0$. We generate proofs using pre-computed parameters for $\mathbf{T}_0$ till the time $t = t_1$, when the table state $\mathbf{T}_1$ is at hamming distance 2^{17} from $\mathbf{T}_0$. At this point, from Figure 6, online proof generation takes around $40s$ for batch of 1000 updates. At $t = t_1$, we also start an offline parameter computation for the table $\mathbf{T}_1$, while continuing to generate online proofs using parameters for $\mathbf{T}_0$. We can generate the next 2^7 batches of updates at an average of approximately $12000/128 \approx 94s$ each, thus finishing with a table state $\mathbf{T}_2$ at hamming distance at most 2^{18} from $\mathbf{T}_0$ at $t_2 = t_1 + 12000$. At this point, we should have the pre-computed parameters for $\mathbf{T}_1$, which is at update distance of 2^{17} from $\mathbf{T}_2$, and thus online proof generation can switch to parameters for the table $\mathbf{T}_1$. This alignment gives us a proving time of $94s$ per batch of 1000, while ensuring system is live at all times. Clearly, a faster offline pre-computation using parallel implementation would allow us to stay at the cheaper end of online proving performance.

$\|I\|$	$\|K\|$	Lemma 7.1	Naive
2^7	2^{14}	3.3s	12s
2^8	2^{15}	7.7s	48s
2^9	2^{16}	16.8s	198s
2^{10}	2^{17}	39.2s	839s

Table 4: Comparison of Lemma 7.1 and naive computation for calculating scalar coefficients for encoded quotients.

Comparison with Prior Works. The proof generation in the prior batching efficient RAM constructions using RSA accumulators [16, 31] involves two key steps (i) generation of a SNARK proof showing knowledge of witness for a relation modeled as arithmetic circuit/R1CS and (ii) computing the witness for the proof generation. A platform agnostic metric to express the cost of the first step is the number of R1CS constraints needed to encode the relation, which is also the metric used in [16] for comparison. Using the R1CS constraints reported in [16, 31] (see Figure 5), we benchmark single-threaded proof generation using the Groth16 protocol (used in prior works) for R1CS of equivalent size on our test-bed. We use publicly available benchmarking suite in [37] for Groth16 benchmarks. Since the the prior works only report performance of parallel implementation of the second step which is common to both (with degree of parallelism not explicitly mentioned), we will use the reported parallel performance to estimate the overall proving time with this caveat.

Even without the benefit of parallelization, our average proof generation time of $\approx 90s$ for batch size of 2^{10} and RAM size of 2^{20} is $3 - 5\times$ faster than the prior works for the same setting. The proof sizes and verification complexity are constant for prior work and our work. Concrete proof sizes are smaller in [16, 31] owing to their usage of Groth16 proving backend, while our verification times are competitive with [31] and substantially less than [16]. One way to reduce proof size in our scheme would be to use a SNARK to prove memory-checking on smaller sub-RAMs, instead of explicit polynomial protocol that we employ for the same. For completeness, we also include the batching efficient RAM using Merkle tree (with Poseidon hash) in the performance comparison in Table 5 which was considered in prior works. Note that the Merkle tree-based approach is faster than that of [31] for batch size of 2^{10} (the break-even point reported to be batch size of ≈ 1200).

Scheme	$\mathcal{P}$ (s)	$\mathcal{V}$ (ms)	$\lvert\pi\rvert$ (KB)
MT20	450	7	0.26
OWWB20 [31]	550 + 43	7	0.26
CFHKKO22 [16]	226 + 43	120	1.3
Our Work	94	15	4.4

Table 5: Comparing performance of our batching-efficient RAM with prior works. $\mathcal{P}$ denotes proof generation time, $\mathcal{V}$ denotes verification time while $|\pi|$ denotes argument size. We mention proof generation time as $a + b$ for [16, 31] where a denote proving time of Groth16 and b denotes witness generation time. The latter is reported in respective works for a parallel implementation.

ACKNOWLEDGMENTS

The research of the second author is supported in part by IBM Global University Program Academic Award and Rising Star Award, Intel Corporation.

REFERENCES

[1] Caulk. https://github.com/caulk-crypto/caulk.

[2] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. In *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*, page 3–33, Berlin, Heidelberg, 2024. Springer-Verlag.

[3] barryWhiteHat. rollup. https://github.com/barryWhiteHat/roll_up.

[4] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[5] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 276–294. Springer, Heidelberg, August 2014.

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[7] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019.

[8] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017. https://eprint.iacr.org/2017/1050.

[9] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. Cryptology ePrint Archive, Report 2019/1021, 2019. https://eprint.iacr.org/2019/1021.

[10] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo2, 2020. https://github.com/zcash/halo2.

[11] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. SOSP '13, New York, NY, USA, 2013.

[12] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state (extended version). Cryptology ePrint Archive, Report 2013/356, 2013. https://eprint.iacr.org/2013/356.

[13] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.

[14] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002.

[15] Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. In *PKC 2024*, volume 14602, pages 337–369, 2024.

[16] Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 455–469. ACM Press, November 2022.

[17] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[18] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, Titouan Tanguy, and Michiel Verbauwhede. Efficient proof of RAM programs from any public-coin zero-knowledge system. In *SCN 2022*, pages 615–638, 2022.

[19] Moumita Dutta, Chaya Ganesh, Sikhar Patranabis, Shubh Prakash, and Nitin Singh. Batching-efficient ram using updatable lookup arguments. *IACR Cryptol. ePrint Arch.*, page 840, 2024.

[20] Liam Eagen, Dario Fiore, and Ariel Gabizon. cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Report 2022/1763, 2022. https://eprint.iacr.org/2022/1763.

[21] Dankrad Feist and Dmitry Khovratovich. Fast amortized KZG proofs. Cryptology ePrint Archive, Report 2023/033, 2023. https://eprint.iacr.org/2023/033.

[22] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, August 1987.

[23] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

[24] Ariel Gabizon and Dmitry Khovratovich. flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. Cryptology ePrint Archive, Report 2022/1447, 2022. https://eprint.iacr.org/2022/1447.

[25] Ariel Gabizon and Zachary J. Williamson. plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315, 2020. https://eprint.iacr.org/2020/315.

[26] Ariel Gabizon, Zachary J. Williamson, and Oana-Madalina Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019.

[27] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728. Springer, Heidelberg, August 2018.

[28] Ulrich Haböck. Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Report 2022/1530, 2022. https://eprint.iacr.org/2022/1530.

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

[30] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 369–378. Springer, Heidelberg, August 1988.

[31] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2075–2092. USENIX Association, August 2020.

[32] Shahar Papini and Ulrich Haböck. Improving logarithmic derivative lookups using GKR. *IACR Cryptol. ePrint Arch.*, page 1284, 2023.

[33] Jim Posen and Assimakis A. Kattis. Caulk+: Table-independent lookup arguments. Cryptology ePrint Archive, Report 2022/957, 2022. https://eprint.iacr.org/2022/957.

[34] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. Cryptology ePrint Archive, Paper 2023/1216, 2023. https://eprint.iacr.org/2023/1216.

[35] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS 2015*. The Internet Society, February 2015.

[36] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them. *Commun. ACM*, page 74–84, 2015.

[37] www.arkworks.com. ark-groth16.

[38] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3121–3134. ACM Press, November 2022.

[39] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. Baloo: Nearly optimal lookup arguments. Cryptology ePrint Archive, Report 2022/1565, 2022. https://eprint.iacr.org/2022/1565.

[40] Yuncong Zhang, Shifeng Sun, and Dawu Gu. Efficient kzg-based univariate sum-check and lookup argument. In *PKC 2024*, volume 14602, pages 400–425, 2024.

[41] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *2018 IEEE Symposium on Security and Privacy*, pages 908–925. IEEE Computer Society Press, May 2018.