Abstract—In verifiable outsourcing, an untrusted server runs an expensive computation and produces a succinct proof (called a SNARK) of the results. In many scenarios, the computation accesses a RAM that the server maintains a commitment to (persistent RAM) or that is initially zero (volatile RAM). But, SNARKs for such scenarios are limited by the high overheads associated with existing techniques for RAM checking. We develop new proofs about volatile, persistent, and sparse persistent RAM that reduce SNARK proving times. Our results include both asymptotic and concrete improvements—including a proving time reduction of up to $51.3 \times$ for persistent RAM. Along the way, we apply two tools that may be of independent interest. First, we generalize an existing construction to convert any algebraic interactive proof (AIP) into a SNARK. An AIP is a public-coin, non-succinct, interactive proof with a verifier that is an arithmetic circuit. Second, we apply Bézout’s identity for polynomials to construct new AIPs for uniqueness and disjointness. These are useful for showing the independence of accesses to different addresses.

1. Introduction

In verifiable outsourcing, a weak client outsources a computation to a powerful server [1, 2]. The server returns the result of the computation and a proof that the computation was done correctly. The proof is succinct: it is small and easy to verify.

Verifiable outsourcing is widely deployed in the context of decentralized systems, such as blockchains, where it is used to greatly reduce the amount of replicated effort to verify a computation. In this setting, an off-chain service performs a computation, such as processing a batch of transactions, and produces a succinct proof that the computation was done correctly. The result of the computation along with the succinct proof are posted on chain, and every validator in the network verifies the proof instead of redoing the computation from scratch. Having a succinct proof is important: its short size reduces total on-chain storage, and its fast verification time saves validator load.

In these deployments—and in many other applications of verifiable outsourcing—the outsourced computation reads from and writes to some persistent state. For example: a RAM that holds the account balances of every participant in a payment system. The computation operates on some initial memory state $M$, which is a map from a set of addresses $\{1, \ldots, N\}$ to values. To reduce storage and verification costs, the weak client (or blockchain) stores a succinct digest $d$ of $M$, e.g., the root of a Merkle tree. The server stores all of $M$, and executes the computation to derive a new state $M'$. The server sends a new digest $d'$ to the client, along with a proof that running the computation on a starting state consistent with $d$ gives a final state consistent with $d'$.

The proof used here is called a SNARK, which we define in more detail in Section 2. In existing SNARKs, proof size and verification time are very small: for example, a few hundred bytes and a few milliseconds [3]. However, proving costs—both time and memory—are enormous: generally thousands of times higher than that of the original computation [4–6]. While costs can be defrayed with parallelism [7–12], they are still substantial, so optimizing SNARKs is an essential and active area of research [2–4, 13–25].

Most of the above research focuses on SNARKs for computations expressed as arithmetic circuits. Since many computations are not circuits, there has been significant research into extending SNARKs to richer computational models including control flow [4, 6, 9], persistent state [26–31], lookup tables [32–41], and RAM [6, 42–46]. These extensions aim to make SNARKs more practical, by improving their compatibility with existing software toolchains. However, they also introduce significant overhead.

We develop improved SNARKs for three kinds of RAM:

- The foregoing has discussed the case in which a SNARK proves the correctness of a computation that accesses a RAM of size $N$ whose initial and final state are committed to with a digest. We call this a persistent RAM.
- Second, we consider a sparse persistent RAM, in which $N$ is very large (e.g., $N \approx 2^{256}$), but at most $C$ (e.g., $C \approx 2^{20}$) cells are non-zero at a time. This case naturally arises when addresses are hashes of unstructured identifiers (e.g., an email).
- Third, we consider a RAM that is initialized to zero for each computation, and whose final state is uncommitted. We call this a volatile RAM. This case is essential for SNARKs of RAM program execution.

We obtain significant asymptotic and concrete improvements. First, for volatile RAM, we give the first construction with proving cost independent of $N$. Second, for persistent RAM, our implementation reduces concrete proving costs
by up to $51.3\times$ relative to the state of the art [30]. Finally, we estimate that our sparse persistent RAM reduces costs by up to $143\times$ over approaches based on prior ideas [30].

Our improvements rely on three ideas. First, we build on a special kind of SNARK called a commit-and-prove (CP) SNARK, which allows for efficient commitments to (and proofs about) moderately sized arrays [47]. CP-SNARKs are promising because they do not require expensive cryptographic operations to be encoded as an arithmetic circuit—unlike state commitments for SNARKs based on Merkle trees or RSA accumulators [30]. However, a CP-SNARK does not immediately provide a RAM, because the array cannot be accessed at data-dependent locations.

Second, we give new interactive proofs to show that a sequence of RAM accesses is consistent with (committed) initial and final memory states. In doing this, one challenge that emerges is proving that accesses (or groups thereof) have unique addresses (or disjoint address sets). We show that using interaction and randomness gives better proofs for these properties, by leveraging Bézout’s identity for polynomials. Our proofs are public-coin, non-succinct, and interactive. In them, the verifier’s final test is an arithmetic circuit, so we call them algebraic interactive proofs (AIPs).

Third, we adapt a recent SNARK, Mirage [48], so that it can be used to compress an AIP into a SNARK. In this transformation, SNARK proving time depends on the size of the verification circuit for the AIP. Thus, by optimizing the AIP verifier, we reduce final proving times.

Our contributions are:

- We give new AIPs for the consistency of a sequence of RAM accesses, where the RAM is volatile (§4), persistent (§5), or sparse and persistent (§6). We define and prove security for all of our protocols.

- Our proofs apply Bézout’s identity for univariate polynomials to prove various uniqueness and disjointness properties, which may be of independent interest.

- We construct CP-Mirage+: a generalization of Mirage that converts a (public-coin, multi-round, non-succinct, non-zero-knowledge) AIP into a CP-zkSNARK. We define and prove security for our construction (§7).

- We implement (§8) and evaluate (§9) our constructions.

For persistent RAM, our implementation performs best on midsize RAMs ($2^{10} \leq N \leq 2^{20}$), where it reduces proving time by up to 51.3×, relative to prior work.

For volatile RAM, our construction is an asymptotic improvement: its proving costs are independent of $N$, while prior constructions scale as $(\log N) / (\log \log N)$. However, the concrete improvement for $N \leq 2^{60}$ is modest: at most a 32.9% proving time reduction.

We also compare our sparse RAM against constructions based on prior ideas, via estimates of rank-1 constraint counts (a standard cost model; §3). For large RAMs of moderate capacity ($2^{10} \leq C \leq 2^{20}$), we predict an improvement of up to $143\times$.

In the rest of the paper, we discuss background and related work (§2), define an arithmetization for AIPs (§3), and then present our contributions (§4–§9).

2. Background and related work

Notation. Let $\mathbb{F}$ be a finite field of prime order $p$, (with $p \approx 2^{255}$), represented as $\{0, \ldots, p-1\}$. The order of $\omega \in \mathbb{F}$, denoted $|\omega|$, is the minimal natural $n > 0$ such that $\omega^n = 1$. Let $[i]$ denote $\{1, \ldots, i\}$ for $i \in \mathbb{N}$. For vectors $x, y$, let $(x, y)$ denote their concatenation: $(x_1, x_2, x_3, y_1, \ldots, y_n)$. For boolean $b$ and any values $t, f$, let $\text{ite}(b, t, f)$ be $t$ if $b$ is true and $f$ otherwise (“if-then-else”). Let $\text{toe}(b) = \text{ite}(b, 1, 0) \in \mathbb{F}$. Let $\mathbb{F}^{n \times m}$ denote $n$ by $m$ matrices of elements from $\mathbb{F}$.

Let $G_1, G_2, G_T$ be cyclic groups of order $p$, generated by $G_1, G_2, G_T$, with the group operation $\cdot$. Let $c : G_1 \times G_2 \rightarrow G_T$ be a pairing: an efficient, non-trivial bi-linear map. For vector $x \in \mathbb{F}^n$ and group element $G$ let $xG = (x_1G, \ldots, x_nG)$.

Bézout’s identity. Let $\mathbb{F}^{<n}[X]$ denote the set of univariate polynomials in $X$ with coefficients in $\mathbb{F}$, of degree less than $n$. Let $f, g, h$ be univariate polynomials in $X$ of any degree. Bézout’s identity states that there exist polynomials $s, t$ such that $fs + gt = h$ if and only if $h$ is divisible by $\gcd(f, g)$. Two corollaries follow. First, take $h = 1$; then suitable $s, t$ exist if and only if $f$ and $g$ share no factors. Second, take $h = X^c$; then suitable $c \in \mathbb{N}, s, t$ exist if and only if powers of $X$ are the only common factors of $f, g$.

Keyed hashing. The coefficient hash $H_c(\alpha \in \mathbb{F}, x \in \mathbb{F}^n)$ for key $\alpha$ and data $x$ is defined as $\sum_{i=1}^n \alpha_i x_i$. The root hash $H_r(\alpha, x)$ is defined as $\prod_{i=1}^n (\alpha - x_i)$. $H_c$ is a universal hash function for vector inputs. That is, for all $x \neq y$, if $\alpha$ is sampled uniformly and independently, then the probability of a collision is at most $n/|\mathbb{F}|$. Meanwhile, $H_r$ is a universal hash function for inputs as multisets. That is, if $x$ and $y$ differ as multisets then the probability of a collision is at most $n/|\mathbb{F}|$. Note that the collision resistance of these keyed hash functions requires keys sampled independently from the data.

Proofs. First, we introduce interactive proofs from the complexity perspective. An interactive proof (IP) [49, 50] is a protocol between a prover $P$ and a PPT (probabilistic, polytime) verifier $V$, by which $P$ convinces $V$ that an instance $x$ is in a language $L$. Two properties must hold:

- completeness: if $x \in L$, then $V$ accepts with overwhelming probability.
- soundness: if $x \notin L$, then $V$ accepts with negligible probability.

Cryptographic Proofs. Now, we introduce additional cryptographic properties for proofs of witness relations.

Let $P$ and $V$ be as before, and let $R(x, w)$ be a witness relation with public instance $x$ (initially known to both

---

1 Our goal is efficiency, which requires only a CP-SNARK. Yet, CP-Mirage+ also guarantees zero-knowledge, making it a CP-zkSNARK.
\( \mathcal{P} \) and \( \mathcal{V} \) and private witness \( w \) (initially known to only \( \mathcal{P} \)). Define the instance language for \( \mathcal{R} \): \( L_{x,R} = \{ x : 3w, R(x,w) = 1 \} \), and regard \( (\mathcal{P}, \mathcal{V}) \) and an IP for \( L_{x,R} \). We say the IP is complete (resp. sound) for \( R \) if it is complete (resp. sound) for \( L_{x,R} \).

If soundness holds only against PPT \( \mathcal{P} \), then the IP is called an argument rather than a proof. One can also require knowledge soundness [51], which informally means that whenever \( \mathcal{V} \) accepts, \( \mathcal{P} \) must “know” a witness \( w \) (slightly more formally, there exists a PPT extractor which can compute a valid \( w \) through interaction with \( \mathcal{P} \)). One can also require zero-knowledge [49], which informally means that the protocol conveys no information about \( w \) beyond its validity (slightly more formally, there exists a PPT simulator which can generate protocol transcripts—without using \( w \)—that are indistinguishable from real transcripts). A proof system is public-coin (also known as Merlin-Arthur [52]) if \( \mathcal{V} \)'s messages are uniformly random. A proof system is non-interactive if there is just one message (from \( \mathcal{P} \) to \( \mathcal{V} \)); in this case, that message is called the proof.

Syntactically, a non-interactive proof system for relation class \( \mathcal{R} \) (with pre-processing) is three PPT algorithms:

- **Setup**\((R \in \mathcal{R}) \rightarrow (pk, vk)\): sample proving and verifying keys
- **Prove**\((pk, x, w) \rightarrow \pi\): create a proof that \( R(x,w) \) holds
- **Verify**\((vk, x, \pi) \rightarrow \{0, 1\}\): check a proof

It is succinct if the time and memory costs of Verify are at most \( poly(|R| + |x|) \), where \( |x| \) is the size of \( x \) and \( |R| \) is the time needed to check \( R \) directly. A SNARK [25] is a succinct, non-interactive argument that is knowledge sound. A zkSNARK is additionally zero-knowledge. There are many zkSNARK constructions, we will build on a construction of Groth [3] that was extend by Kosba et al. [48].

### 2.1. Related work

Memory checking has a long history [53], and there is much prior work on SNARKs with volatile [32, 34, 43–45, 54] and persistent [6, 42, 55] RAM. These constructions use tools such as Merkle trees, routing networks [56], lookup proofs [34], and RSA accumulators [57, 58]. We describe this prior work—and compare it to our constructions—in Sections 4.5 and 5.3. A number of RAM-related works consider goals orthogonal to ours. For instance, vRAM focuses on RAM with computation-independent Setup, and other works [32, 44, 45, 59] and projects [60–62] build on SNARKs for RAM to prove executions of RAM programs.

Further afield, others have studied RAM checking for interactive, non-succinct zero-knowledge proofs [63–67].

There is also significant work on SNARKs that access other data structures. A long line of work [32], with several recent additions [33–41], develops “lookup” proofs for SNARKs. These are essentially read-only memories. Others consider sets [68], multisets [31], maps [69], concurrent maps [29, 70], and relations [71].

Protostar [72] builds a folding scheme for special-sound protocols; this is closely related to our CP-zkSNARK for \( \mathcal{V} \) and \( \mathcal{P} \). This is closely related to our CP-zkSNARK for
3.2. A zkSNARK for I-R1CS

In Section 7, we construct Mirage+: a zkSNARK for I-R1CS. We also prove its security. Here, we summarise the key security and efficiency properties of our construction. The first is that Mirage+ transforms a complete and knowledge-sound AIP \( \phi \) into a zkSNARK. We denote the application of Mirage+ to \( \phi \) as \( \text{Mirage}+\phi \).

**Theorem 1.** Let I-R1CS \( \phi \) be complete and knowledge-sound for witness relation \( R \). Then in the generic group model (GGM), \( \text{Mirage}+\phi \) is a zkSNARK for \( R \).

Since the AIP need not be succinct or zero-knowledge, it is easy to transform a sound AIP into a knowledge-sound AIP. Thus, Mirage+ can also transform a complete and sound AIP into a zkSNARK. So, it suffices to construct AIPs that are complete and sound.

**Corollary 1.** Let \( R \) be a witness relation and let I-R1CS \( \phi \) be complete and sound for language \( L_R \) = \{(x, w) : R(x, w) = 1\}. Then there is a zkSNARK for \( R \) in the GGM.

**Proof.** Note that in \( L_R \), \( (x, w) \) is the instance. Let \( \phi' \) be \( \phi \), with \( w \) moved from the first message to the first message. Then, \( \phi' \) is complete and knowledge-sound for \( R \). Applying Mirage+ to \( \phi' \) gives a zkSNARK for \( R \).

**Efficiency.** Mirage+ is very efficient. Verification uses \( O(t_x + \mu) \) time and memory; concretely, just a few milliseconds (Sec. 9). Proofs are \( \mu + 2 \) elements of \( G_1 \) and 1 element of \( G_2 \); concretely, just a few hundred bytes (Sec. 9). Generally, proving costs are dominated by \( O(n \log n) \) \( G_1 \) operations for Fourier transforms and \( O(n \lambda/(\log n)) \) \( G_1 \) operations for multi-scalar multiplications.

Shrinking Mirage+’s \( \mathcal{P} \) cost generally requires shrinking \( n \): the number of I-R1CS constraints. Interestingly, \( n \) is a measure of the verifier cost of the AIP that encoded by the I-R1CS. Thus, for the rest of this paper, our primary objective is to construct AIPs with small \( \mathcal{V} \) complexity.

**Commit and prove.** We also construct CP-Mirage+: an efficient zkSNARK for witness relations where parts of the witness are committed. It uses the commit-and-prove technique cite{CCS:CamFioQue19} to avoid encoding the commitment scheme as I-R1CS constraints, which would be very expensive. The following theorem states the properties of CP-Mirage+: the full construction and interface are in Appendix B.

**Theorem 2.** Let I-R1CS \( \phi \) be complete and knowledge-sound for witness relation \( R(x, (w_1, \ldots, w_k, w)) \). Let Com be a Pedersen commitment scheme in \( G_1 \). Then in the GGM, CP-Mirage+ is a zkSNARK for witness relation

\[
R_{\text{com}} = \{((x, c_1, \ldots, c_k), ((w_1, a_1), \ldots, (w_k, a_k), w)) : \\
\bigwedge_{i=1}^{k} c_i = \text{Com}(x, a_i) \land R(x, (w_1, \ldots, w_k, w))\}
\]

Mirage+ and CP-Mirage+ can transform AIPs into zkSNARKs (with optionally committed witnesses). Thus, as we will see, we can build SNARKs for RAM consistency by building AIPs and then applying either Mirage+ (for volatile RAM) or CP-Mirage+ (for persistent RAM).

4. Volatile memory

In this section, we develop an improved AIP for volatile memory checking in a zkSNARK. First we define what an AIP for volatile memory is. Then, we explain our solution.

4.1. Problem and overview

An access acc is a tuple \((a, v, w)\) with address \(a\) \(\in\) \(\mathbb{F}\), value \(v\) \(\in\) \(\mathbb{F}\), and write bit \(w\) \(\in\) \{0, 1\} \(\subset\) \(\mathbb{F}\). A transcript \(\text{tr}\) is an access sequence \(\text{acc}_1, \ldots, \text{acc}_\lambda\). A memory state (of size \(N \leq |\mathbb{F}|\)) is a function \(M : [N] \rightarrow \mathbb{F}\), where \(M(a)\) is the value at address \(a\) \(\in\) \([N]\). (We will consider different representations of \(M\), including as a vector in \(\mathbb{F}^N\).

A transcript is valid with respect to initial state \(M_0\) and final state \(M_A\) if:

\[
\text{valid}(\text{tr}, M_0, M_A) \triangleq \exists M_1, \ldots, M_{A-1}, \forall i \in [A], \forall a \in [N], \left(M_i(a) = M_i'(a) \lor \right) \land \land a = a_i \land M_i(a) = v_i)
\]

This condition is called reads-over-write (RoW), because it requires the values of future reads to agree with past writes. Volatile memory captures the case where the initial state is zeroed and the final state is unspecified. (It is easy to generalize to various other structured initial states.) If \(\mathbf{0}\) denotes the memory state that is zero everywhere, then the language of valid volatile transcripts is the set:

\[
L_{\text{vmem}} = \{\text{tr} : \exists M_A, \text{valid}(\text{tr}, \mathbf{0}, M_A)\}
\]

Now we explain our new AIP for \(L_{\text{vmem}}\), which is built on a conditional uniqueness proof. First, we explain the role of ordering proofs in volatile memory, and how to build a sufficient ordering proof from a conditional uniqueness proof (§4.2). Second, we design novel AIPs for uniqueness (§4.3) and conditional uniqueness (§4.4). Third, we compare the overall asymptotic cost of our volatile memory proof with those from prior work (§4.5).

4.2. Reducing to conditional uniqueness

Figure 2 illustrates the architecture of prior volatile memory proofs. The parties start with a \(t\)-order transcript \(\text{tr}\). The high-level idea is to sort the transcript by address, and then check the address-order transcript (which is easier). First, \(\mathcal{P}\) and \(\mathcal{V}\) augment each access \(\text{acc}_i\) in the original transcript \(\text{tr}\) with a new “time” field \(t_i = i\), for \(i \in [A]\). Notice that \(\text{tr}\) is \(t\)-ordered. Then, \(\mathcal{P}\) sends a new transcript \(\text{tr}'\) that should be lexicographically ordered by \((a, t)\). Finally, \(\mathcal{P}\) proves that \(\text{tr}'\) (1) is a permutation of \(\text{tr}\), (2) is \((a, t)\)-ordered and (3) respects RoW [43].

Prior work gives good permutation and RoW proofs for \(\text{tr}'\) [32, 43, 54]. In Appendix C, we present the state-of-the-art techniques for each of these, expressed as AIPs for suitable languages. Ultimately, the AIPs for both problems.
have perfect completeness, soundness error $O(A/|F|)$, $\mathcal{V}$ complexity $O(A)$, and $\mathcal{P}$ complexity $O(A + N)$.

However, prior $(a, t)$-ordering proofs have some drawbacks. In particular, no prior $(a, t)$-ordering proof has $\mathcal{V}$ complexity $O(A)$—all have some dependence on $N$.

Our first insight is that $(a, t)$-ordering is not necessary for soundness (though it is sufficient). A sufficient and necessary condition is that $tr'$ be $a$-grouped and $t$-ordered within those groups. That is, define $g_i$ (“group start”) to be 1 if $(i = 1 \lor a_{i-1} \neq a_i)$ or 0 otherwise. Then, it suffices for adjacent accesses to satisfy $g_i = 1 \lor t_i > t_{i-1}$ and for the sequence $(a_i: i \in [A], g_i = 1)$ to contain unique elements. Intuitively, this suffices because the address are independent of one another—so their order is irrelevant. We prove this sufficiency in Appendix C.1, by constructing and proving the soundness of $\Pi_{\text{uniq}}$: an AIP for $\mathcal{L}_{\text{uniq}}$ which follows Figure 2 but only uses $a$-grouping.

So, it remains to build a conditional ordering and conditional uniqueness proof. The former is easy with existing tools (see App. C.1), but uniqueness requires new ideas.

4.3. Uniqueness proof

First, we consider the case when all $g_i = 1$: i.e., a proof that a sequence $a = (a_1, \ldots, a_A) \in [A]^A$ contains unique elements. That is, we give an AIP for $\mathcal{L}_{\text{uniq}}$.

$$L_{\text{uniq}} = \{ a : \forall i \neq j, a_i \neq a_j \}$$

Naively enforcing pairwise disequalities has $\mathcal{V}$ complexity $O(A^2)$. Our AIP will have $\mathcal{V}$ complexity $4A + O(1)$.

At a high-level, our AIP applies Bézout’s identity to a polynomial encoding of $a$. Define the polynomial $z(X) \in [F][X]$ by $\prod_i (X - a_i)$ and let $z'(X)$ be its formal derivative. If $a \in L_{\text{uniq}}$, then $z$’s roots have multiplicity one. Thus, $z$ and $z'$ must be co-prime, and there exist Bézout polynomials $s$ and $t$ with degrees less than $A - 1$ and $A$ respectively, such that $z(X)s(X) + z'(X)t(X) = 1$ (Sec. 2).

Protocol 1 is our AIP. In it, $\mathcal{P}$ computes the Bézout polynomials $s \in \mathbb{F}^{<A-1}[X]$ and $t \in \mathbb{F}^{<A}[X]$ such that $zs + z't = 1$ using $\Pi_{\text{uniq}}$ and sends them to $\mathcal{V}$ as coefficients. Then, $\mathcal{V}$ samples $\alpha \in \mathbb{F}$ and tests $z(\alpha)s(\alpha) + z'(\alpha)t(\alpha) = 1$. $\mathcal{V}$ obtains $s(\alpha)$ and $t(\alpha)$ through Horner evaluation. For $z(\alpha)$ and $z'(\alpha)$ it uses the following recursion. For $j \in [A]$, define $z_j(X) = \prod_{i \leq j} (X - a_i)$. Then, $z_1(\alpha) = (\alpha - a_1)$, $z'_1(\alpha) = 1$, and for $j > 1$, we have:

$$z_j(\alpha) = z_{j-1}(\alpha) \times (\alpha - a_j)$$
$$z'_j(\alpha) = z_{j-1}(\alpha) + z'_{j-1}(\alpha) \times (\alpha - a_j)$$

Thus, $z(\alpha) = z_A(\alpha)$ and $z'(\alpha) = z'_A(\alpha)$ are computable with $2A + O(1)$ constraints. So, $\mathcal{V}$’s circuit has $4A + O(1)$ constraints in total.

**Theorem 3.** The AIP $\Pi_{\text{uniq}}$ for language $\mathcal{L}_{\text{uniq}}$ has perfect completeness, soundness error $(2A - 2)/|\mathbb{F}|$, and $\mathcal{V}$ complexity $4A + O(1)$.

**Proof.** Completeness follows immediately from Bézout’s identity, and the $\mathcal{V}$ complexity has been analyzed above.

For soundness, consider the probability that $\mathcal{V}$ accepts if $a$ contains duplicates. In this case, $z$ contains at least one root with multiplicity at least two. Therefore, $z$ and $z'$ share this root. Therefore, $zs + z't$ cannot equal 1 (Sec. 2), so $zs + z't - 1$ is a non-zero polynomial (of degree $\leq 2A - 2$).

Then, the $\mathcal{V}$ accepts (i.e., $z(\alpha)s(\alpha) + z'(\alpha)t(\alpha) = 1$) only when $\alpha$ is a root of said polynomial. The probability of this is at most $(2A - 2)/|\mathbb{F}|$, by the fundamental theorem of algebra.
Algorithm 1: A simple algorithm, SimpleUniqBez, for computing Bézout polynomials \(s, t\) for \(z(X) = \prod_{i=1}^{L}(X - a_i)\) and its derivative. ProductTree\((p_1, \ldots, p_L)\) computes \(\prod_i p_i\), Evaluate\((p, a)\) computes \(p(a_1), \ldots, p(a_L)\), and Interpolate\((y, x)\) computes \(f \in \mathbb{F}^{\mathbb{F}[X]}\) such that \(f(x_i) = y_i\). The optimized UniqBez is in Appendix D.

\[
\text{SimpleUniqBez}(a_1, \ldots, a_L) \rightarrow (s, t)
\]

\[
z(X) \leftarrow \text{ProductTree}(X - a_1, \ldots, X - a_L)
\]

\[
b \leftarrow \text{Evaluate}(z'(X), a)
\]

\[
t(X) \leftarrow \text{Interpolate}(b_1^{-1}, \ldots, b_L^{-1}, a)
\]

\[
\text{return } s \leftarrow (1 - z't)/z \text{ and } t
\]

\(O(\log^2 A),^2\) but the FEEA is the concrete bottleneck, for two reasons. First, the FEEA’s constants are large [75, Chapter 9]. Second, the FEEA is not divide-and-conquer, so parallelizing it is tricky and has higher overhead.

Instead, we give a simple, specialized, divide-and-conquer algorithm for computing the coefficients of \(s\) and \(t\) from \(a\). SimpleUniqBez (Alg. 1) is simplification of our algorithm. First, it computes \(z\) using a product tree. Second, it evaluates \(z'\) at each \(a_i\). Third, it interpolates \(t\) as the polynomial whose evaluation at each \(a_i\) is \(z'(a_i))^{-1}\). Fourth, it computes \(s\) as \((1 - z't)/z\).

To see that SimpleUniqBez is correct, consider \(t(a_i)\). Per Bézout’s identity, we have \(z(a_i)s(a_i) + z'(a_i)t(a_i) = 1\). But, \(z(a_i) = 0\), so \(t(a_i)\) is just the inverse of \(z'(a_i)\). Thus, \(t\) can be interpolated from the \((z'(a_i))^{-1}\). Then, per Bézout’s identity again, we have that \(z\) divides \(1 - z't\) without remainder, and that the quotient is \(s\).

SimpleUniqBez is efficient. The last step (computing \(s\)) takes only \(O(\log A)\) time. The first three steps are the bottleneck, and all take \(O(\log^2 A)\) time. Also, they are all divide-and-conquer algorithms that are easy to parallelize [75].

We present the full UniqBez algorithm in Appendix D. It optimizes SimpleUniqBez by exploiting connections between the uses of ProductTree, Evaluate, and Interpolate. Specifically, in evaluating the third, one incidentally evaluates the first two. The use of ProductTree in Interpolate is unsurprising. However, the special role that evaluations of \(z'\)'s play in Interpolate is quite fortuitous. We prove the following theorem about UniqBez:

**Theorem 4.** Given \(a_1, \ldots, a_A \in \mathbb{F}\), UniqBez outputs \(s, t \in \mathbb{F}[X]\) satisfying \(sz + tz' = 1\) (where \(z = \prod_{i=1}^{L}(X - a_i)\)) in time at most \(4.5M(A) \log A + O(\log A).

Whereas, the FEEA requires time \(22M(A) \log A + O(\log A)\) in the worst case [75, Theorem 11.7].

\(\overline{2}\)Throughout, we assume that \(M_p(A)\) (abbreviated \(M(A)\))—the number of \(\mathbb{F}_p\) operations needed to multiply polynomials in \(\mathbb{F}_p^{\mathbb{F}[X]}\)—is \(O(\log A)\). In our fields of interest—which are smooth—the FFT gives this. In other fields, understanding \(M_p(A)\) is an old (but still active) problem [76–78].

---

**Table 1:** The \(\mathcal{V}\) complexity of checking \(A\) accesses to a volatile RAM of size \(N\). Prior works order accesses by address \(a\) using various techniques. Our approach groups accesses by \(a\) using a uniqueness proof.

<table>
<thead>
<tr>
<th>Reference</th>
<th>((a, t))-property</th>
<th>Overall (\mathcal{V}) complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyRam [44]</td>
<td>((a, t))-order</td>
<td>(\Theta(A(\log A + \log N)))</td>
</tr>
<tr>
<td>Arya [32]</td>
<td>((a, t))-order</td>
<td>(\Theta_1(\log N/\log \log N), \Theta_N(A))</td>
</tr>
<tr>
<td>Haböck [34]</td>
<td>((a, t))-order</td>
<td>(\Theta_1(\log N/\log \log N), \Theta_N(A))</td>
</tr>
<tr>
<td>This work</td>
<td>(a)-group, (t)-order</td>
<td>(\Theta(A))</td>
</tr>
</tbody>
</table>

---

**4.4. Conditional uniqueness**

One can generalize Protocol 1 to be a conditional uniqueness proof. That is, a proof for the language:

\[
\mathcal{L}_{c\text{-uniq}} = \{(a \in \mathbb{F}^A, g \in \{0, 1\}^A) : \\
\forall i \neq j, g_i = 0 \lor g_j = 0 \lor a_i \neq a_j\}
\]

\(P\) simply omits \(a_i\) with \(g_i = 0\) from the computation of \(z, z', s, t\). Also, \(V\)'s circuit uses a conditional recurrence:

\[
z_j(a) = z_{j-1}(a) \times (g_j [a - a_j - 1] + 1)
\]

\[
z'_j(a) = g_j [z_{j-1}(a) + z'_{j-1}(a) \times (a - a_j - 1)] + z'_{j-1}(a)
\]

Thus, \(z, z'\) are evaluable with a size \(4A + O(1)\) circuit. The result is an AIP for \(\mathcal{L}_{c\text{-uniq}}\) with the same characteristics as Protocol 1. Save that \(V\)'s circuit now has size \(6A + O(1)\).

**4.5. Comparison to prior work**

We have presented a novel AIP for conditional uniqueness with \(\mathcal{V}\)-cost \(\Theta(A)\). As discussed, this gives an AIP for \(\mathcal{L}_{vmem}\). Now, we compare our memory proof against prior works. As we will see, ours is the first which has cost independent of \(N\); the address space size. Table 1 shows the comparison. The approaches differ in (a) whether they group or order the accesses \(a\) and \((b)\) how they do that.

TinyRAM [44] orders values through bit-splitting. Informally, \(a_i \geq a_i - 1\) reduces to \(a_i - a_i - 1\) being “small,”—fitting in \([\log N]\) bits. Ordering times requires \(O(\log A)\) more bits, so each access requires \(O(\log A + \log N)\) constraints.

To avoid this log overhead, Arya [32] range-checks though lookup proofs. Essentially, it argues that for any \(a_i - a_i - 1\) is in the set \(\{0, \ldots, N\}\). Each lookup uses \(O(1)\) constraints, but there is an \(O(N)\) one-time overhead for the set, which is untenable for large \(N\). One can defray this cost by combining lookups with bit-splitting [33, 35, 39, 41]. One splits the differences into medium-size “digits” that are each shown to be in a set \(\{0, \ldots, k-1\}\). The net cost is \(O(k + A \log k(N))\). When \(k\) is set optimally, the asymptotics of this approach become non-trivial. We analyze them in Appendix C, finding that the overhead now depends quasi-logarithmically on \(N\) (Table 1).\(^3\) Recently, Haböck improved the concrete multiplicative costs of this approach [34].

\(^3\)Recall that \(O_A(f(N))\) allows the big-\(O\) constant to depend on \(A\).
Figure 3: An overview of our persistent memory proof. \( m, m' \) are the initial and final memory states. \( a \) includes all addresses accessed by \( \text{acc}_1, \ldots, \text{acc}_A \). For all \( i \in [A] \), \( v_i, v'_i \) are the initial and final values at \( a_i \).

Our AIP is the first with cost independent of \( N \). In Section 9, we do a concrete comparison, finding that our approach reduces final proving time by a much as 32.9\%, for 60-bit addresses.

5. Persistent memory

Now we turn to persistent memory. Here, the problem is to show that a transcript \( tr \) is consistent with specified initial and final memory states. Thus, to build an AIP for:

\[ \mathcal{L}_{\text{pmem}} = \{(tr, M_0, M_A) : \text{valid}(tr, M_0, M_A)\} \]

Through CP-Mirage+ (with commitment scheme Com), we can transform an AIP for \( \mathcal{L}_{\text{pmem}} \) into a zkSNARK for a witness relation in which \( M_0 \) and \( M_A \) are committed as \( c, c' \) with commitment randomness \( o, o' \):

\[ R_{\text{pmem}} = \{(c, c', (M_0, M_A, o, o')) : c = \text{Com}(M_0, o) \land c' = \text{Com}(M_A, o') \land \text{valid}(tr, M_0, M_A)\} \]

In our AIP for \( \mathcal{L}_{\text{pmem}} \), the initial and final memory states are represented as vectors \( m, m' \in \mathbb{F}^N \). That is, address \( i \in [N] \) has initial value \( m_i \) and final value \( m'_i \).

Figure 3 illustrates our high-level approach.\(^4\) First, \( \mathcal{P} \) sends a vector of “active” addresses \( a \)—those that are touched by \( tr \). They also send the initial and final values at those addresses \( v, v' \in \mathbb{F}^A \). Then, \( \mathcal{P} \) and \( \mathcal{V} \) run two sub-protocols, which we explain in the rest of this section. First, to show that \( (v, v', a) \) represent the active cells of \( m \) and \( m' \) (\S5.1). Second, to show that \( (v, v', a) \) is consistent with the transcript \( tr \) (\S5.2). We conclude with a comparison to prior work (\S5.3).

\(^4\) Assume that \( A \leq N \); lifting this requirement is straightforward.

5.1. Selecting active cells

Our first sub-protocol proves the connection between \( v, v', a, m, \) and \( m' \). Each \( (a_i, v_i) \) pair should match some unique \( (i, m_i) \), and likewise for the \( a_i, v'_i \), and \( m'_i \). Further, every unmatched \( m_i \) should equal \( m'_i \). More precisely:

\[ \mathcal{L}_{\text{active}} = \{(m, m' \in \mathbb{F}^N; v, v', a \in \mathbb{F}^A) : \]

\[ \bigwedge_{i=1}^A (v_i = m_{a_i} \land v'_i = m'_{a_i}) \land \]

\[ \bigwedge_{i=1}^N (i \notin a \implies m_i = m'_i) \land \]

\[ \bigwedge_{i=1}^N \bigwedge_{j=1}^N (a_i = a_j) \}

Figure 4 illustrates our protocol \( \Pi_{\text{active}} \) for this language. The idea is to sample a key \( \beta \) and represent every value-address pair \((v, a)\) by its hash \((H_c(\beta, (v, a)) = v + \beta a)\) (represented by the boxes with the hashes \( H_c \)). \( \mathcal{P} \) sends the hashes \( h = (h_1, \ldots, h_{N - A}) \) for inactive memory cells (top-center). Then, it suffices to show that \((h, v_1 + \beta a_1, \ldots, v_A + \beta a_A)\) is a permutation of the initial memory state and \((h, v'_1 + \beta a_1, \ldots, v'_A + \beta a_A)\) is a permutation of the final memory state. For this, we use the \( H_c \) hash to test multiset equality (the two vertical rectangles). Protocol 2 defines \( \Pi_{\text{active}} \) in full. We prove the following in Appendix A:

**Theorem 5.** If \( A \leq N \), the protocol \( \Pi_{\text{active}} \) (for language \( \mathcal{L}_{\text{active}} \)) has perfect completeness, soundness error \( \leq (N^2 + 2N + A)/|\mathcal{F}| \), and \( \mathcal{V} \) complexity \( 3N + 2A + O(1) \).

**Why send \( h' \)?**. In \( \Pi_{\text{active}} \), \( \mathcal{P} \) sends the hashes \( h_i \) of inactive memory cells rather than their address-value pairs. In an alternative protocol for \( \mathcal{L}_{\text{active}} \), \( \Pi'_{\text{active}} \), \( \mathcal{P} \) sends these pairs,
and \( V \) computes the \( h_i \). This change increases \( V \) complexity by \( N - A \), which is why we use \( \Pi_{\text{active}} \) instead of \( \Pi'_{\text{active}} \). But, \( \Pi_{\text{active}} \) has two advantages that might matter in other contexts. First, it has one fewer round, since the keys \( \alpha \) and \( \beta \) can now be sampled simultaneously. Second, the soundness error of \( \Pi_{\text{active}} \) is \( N/|F| \), which is better than our \( O(N^2/|F|) \) bound for \( \Pi_{\text{active}} \). These advantages are not meaningful for us: CP-Mirage+ makes extra rounds very cheap, and its soundness error far exceeds that of \( \Pi_{\text{active}} \).

5.2. Transcript validity

It remains to be shown that the transcript
\[
(a_1, v_1), \ldots, (a_A, v_A), \text{acc}_1, \ldots, \text{acc}_A, (a_1', v_1'), \ldots, (a_A', v_A')
\]
is consistent. We give our full proof for this, \( \Pi_{\text{pmem}} \), in Appendix C.3. Here, we sketch our approach by describing how it differs from our volatile memory proof. First, we ensure that the initialization accesses \((a_i, v_i)\) come first, and the finalization accesses \((a_i, v_i')\) come last. Second, we eliminate the uniqueness argument from the last section.

Initialization and finalization. To ensure that each \( a_i \) has initial value \( v_i \) and final value \( v_i' \), it suffices to ensure that each address is initially written to by access \((a_i, v_i)\) and finally read from access \((a_i, v_i')\). We ensure this by setting the time \( t \) associated with each access appropriately. As before, for \( i \in [A] \), standard access \( \text{acc}_i \) is assigned time \( t_i = i \). However, each initialization access \((a_i, v_i)\) is assigned time \( t_i = 0 \) and finalization access \((a_i, v_i')\) is assigned time \( t_i = A+1 \). These times are respectively before and after all standard accesses.

Omitting uniqueness. Further, since the \((a_i, v_i)\) must be the first accesses to each address, we set their “group start” (Sec. 4.2) \( g \) to be 1 and we set \( g = 0 \) for every other access. Now, the addresses with \( g = 1 \) are uniquely by construction. Thus, we can omit the last section’s conditional uniqueness argument. Note that with this change, we set the \( g_i \) in the \( t \)-order transcript, and they are included in the permutation proof. Whereas, for volatile memory we computed the \( g \)’s based on the \( a \)’s in the \((a, t)\)-order transcript.

5.3. Comparison to prior work

Prior persistent memory proofs for zkSNARKs use Merkle trees or RSA accumulators [30]. Table 2 summarizes the costs of these approaches and of ours.

In the Merkle approach, \( V \) checks one collision-resistant (CR) hash per read and per level of a \((\log N)\)-depth tree. For a write, this is two hashes. Keyless hash functions are highly non-linear, so even multiplication-optimized hashes (e.g., Poseidon [80]), require \( \approx 300 \) multiplications.

RSA accumulators give a very different persistent memory proof [30]. The \( V \) cost is independent of \( N \), but their are three downsides. First, there is a large constant overhead (about 10M multiplications [30]) for verifying two Wesolowski proofs [79]. Second, each access requires many multiplications (about 2k [30]), to compute and accumulate a division-intractable hash [58]. Third, while the \( V \) complexity is independent of \( N \), proving cost is not. \( P \) must additionally evaluate a 2048-\( N \) bit exponentiation modulo the RSA modulus (which is concretely expensive) in order to compute its witness.

Our approach has \( V \) complexity \( 3N + 41A \). The \( N \) constant comes directly from \( \Pi_{\text{active}} \) and we measure the \( A \) constant in our experiments (Sec 9.2). Our experiments also see that this gives a significant concrete improvement for many \( N \) and \( A \). It reduces constraint counts by up to 98.2% and overall proving times by up to 51.3x. Intuitively, the improvement is because both RSA and Merkle encode expensive cryptographic operations (e.g., Wesolowski’s proof or CR hashing) in constraints, while we do not.

6. Sparse persistent memory

Now, we turn to our final problem: sparse persistent memory. The previous section considered a dense memory: each address \( i \in [N] \) had its value explicitly represented by \( m_i \) in the memory state \( m \). Now, the address space will be nearly all of \( \mathbb{F} \), but we will assume that at most \( C \) cells are non-zero. Thus, \( C \) is the capacity for a sparse memory.

The initial state is represented by \( m, a \in \mathbb{F}^C \). Each non-zero \( a_i \) indicates that address \( a_i \) has value \( m_i \). Thus, \( 0 \in \mathbb{F} \) is the only invalid address, and when \( a_i = 0 \), we require that \( m_i = 0 \). Further, the non-zero \( a_i \) must be unique. The final state is represented by \( m', a' \in \mathbb{F}^C \) with the same meaning.

First (§6.1), we summarize our approach and explain how constructing a sparse persistent memory proof reduces to proving the uniqueness of valid addresses. Second (§6.2), we explain how to prove the uniqueness of valid addresses. Third (§6.3), we compare against alternatives.

6.1. Overview

Our high-level protocol for sparse persistent memory is similar to the one for dense persistent memory. In the first round, \( P \) sends encodings of the initial and final states of active cells: those whose values might change. That is, \( P \) sends \( b, v, b', v' \in \mathbb{F}^A \). The pair \((b_i, v_i)\) represents a cell at address \( b_i \) with value \( v_i \), before the accesses \( \text{acc}_i \). The pair

<table>
<thead>
<tr>
<th>Approach</th>
<th>Constraints</th>
<th>Other bottlenecks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merkle</td>
<td>( 600 \cdot A \log N )</td>
<td></td>
</tr>
<tr>
<td>Linear scan</td>
<td>( 3 \cdot N \cdot A )</td>
<td></td>
</tr>
<tr>
<td>RSA [30]</td>
<td>( 10M + 4k \cdot A )</td>
<td>( 2048N )-bit RSA exp.</td>
</tr>
<tr>
<td>This work</td>
<td>( 3 \cdot N + 41 \cdot A )</td>
<td>( 2 \times \text{MSM}_{\text{eq}}(N) )</td>
</tr>
</tbody>
</table>

Table 2: Constraint counts for prior persistent memory approaches and ours. Merkle and RSA estimates are based on Ozdemir et al. [30], who require: \( \approx 5M \) constraints for a Wesolowski [79] verifier, \( \approx 2k \) constraints per RSA accumulator insert/delete, and \( \approx 300 \) constraints per CR hash (Poseidon [80]). We compute the \( N \)-dependence of our approach directly, and the \( A \) dependence through a regression on the results of Section 9.2.
Figure 5: A visualization of our active-cell selection proof for sparse persistent memories. Two tests are not shown. First, for all \( i \in [A] \), \( b_i \) must be zero, or \( b_i = b'_i \); this is enforced as: \((b_i - b'_i) b_i = 0\). Second, the non-zero \( a'_i \) must be unique; we discuss how to ensure this in Section 6.2.

\[
(b'_i, v'_i) \text{ represents a cell at address } b'_i \text{ with value } v'_i, \text{ after the accesses. We call a cell valid if its address is zero-nondegenerate.}
\]

\( \mathcal{P} \) must show a number of facts about the active cells. Figure 5 illustrates its proof, at a high-level. This proof follows \( \Pi_{\text{active}} \) very closely, so we will only discuss the differences. First, in \( \Pi_{\text{active}} \) value \( m_i \) lived at address \( i \); now, it lives at address \( a_i \) (and likewise for \( m_i', a_i' \)). This creates the main challenge of this section—proving the uniqueness of the valid \( a'_i \)—which we will address in Section 6.2.

Second, in \( \Pi_{\text{active}} \) the initial address \( b_i \) and final address \( b'_i \) of an active cell were always the same. Now, \( b'_i \) need only equal \( b_i \) if \( b_i \) is valid. If \( b_i \) is invalid, then \( b'_i \) can be valid: this corresponds to creating a new cell in the memory that is potentially non-zero. \( \nu \) enforces the conditional equality of \( b_i \) and \( b'_i \) with the equation \((b'_i - b_i) b_i = 0\).

Beyond active-cell selection, \( \mathcal{P} \) must show that the active cells are consistent with the transcript. This amounts to proving the consistency of the following transcript: \((b'_1, v_1), \ldots, (b'_A, v_A)\), \(a_{cc}, \ldots, a_{ccA}, (b'_1, v'_1), \ldots, (b'_A, v'_A)\). Note that the initialization accesses have addresses \( b' \), not \( b \). This is because \( b' \) contains both the pre-existing and newly created valid addresses, while \( b \) contains only the pre-existing ones. This consistency proof is done exactly as in Section 5.2.

### 6.2. Valid address uniqueness

How can \( \mathcal{P} \) show the uniqueness of the (final) valid addresses? Naively, \( \mathcal{P} \) could apply the proof \( \Pi_{\text{uniq}} \) (Sec. 4.3) to the \( a'_i \). However, this approach is incorrect (incomplete) because it does not permit repetitions of the dummy address 0. Instead, \( \mathcal{P} \) could provide an indicator sequence \( c \in \{0, 1\}^C \subseteq \mathbb{F}^C \), where each \( c_i = \text{toe}(a'_i) \neq 0 \). \( \mathcal{P} \) would show that the \( c_i \) satisfy their definition, and then apply a conditional uniqueness argument to \( a' \) and \( c \) (Sec. 4.4). This approach is complete and sound. However, it’s \( \nu \) complexity is \( 8C + O(1) \) (this is because the conditional uniqueness argument has cost \( \approx 6C + O(1) \) and proving that the \( c_i \) satisfy their definition has cost \( 2C \) [81]).

\[
\Pi_{\text{PuncUniq}}(a_1, \ldots, a_C) \text{ for } \mathcal{L}_{\text{PuncUniq}}
\]

\[
\begin{align*}
\mathcal{P}(\ldots) & \quad \mathcal{V}(\ldots) \\
S & \leftarrow \{a_i\}_{i=1}^C \\
c & \leftarrow C - |S| \\
s, t & \leftarrow \text{UniqBez}(S) \\
\end{align*}
\]

Protocol 3: Our punctured uniqueness proof for the \( a_i \). \( \mathcal{P} \)’s messages are \( c \leq C, s \in \mathbb{F}^{\leq C-2}[X], t \in \mathbb{F}^{\leq C-1}[X]. \) \( \nu \)’s test is as a size \( 4C + O(\log C) \) circuit; see the text.

In the sections below we give two other approaches that improve \( \nu \) complexity. First (\S 6.2.1), we give a protocol based on punctured uniqueness, with \( \nu \) complexity \( 4C + O(\log C) \). Second (\S 6.2.2), we give a protocol based on punctured disjointness, with \( \nu \) complexity \( 2C + O(A + \log C) \).

#### 6.2.1. Punctured uniqueness

To inspire our punctured uniqueness proof, consider what goes wrong if we apply our (non-conditional) uniqueness proof (Sec. 4.3) to the \( a_i \). First, \( \mathcal{P} \) defines \( z(X) = \prod(X - a_i) \) and computes the derivative \( z' \). Now, \( \mathcal{P} \) must show that \( \gcd(z, z') = 1 \) by providing \( s, t \) such that \( zs + z't = 1. \) But, if a contains \( d \geq 2 \) instances of 0, then \( X^{d-1} \) is a common factor of \( z \) and \( z' \). So, \( \mathcal{P} \) cannot complete the protocol.

Our idea is simple: have \( \mathcal{P} \) show that \( \gcd(z, z') \) has form \( X^c \). This shows that 0 is the only common root of \( z \) and \( z' \), and is thus the only duplicate root of \( z \). Our proof is for the language \( \mathcal{L}_{\text{PuncUniq}} \), defined as follows:

\[
\mathcal{L}_{\text{PuncUniq}} = \{a : \forall i \neq j, a_i \neq a_j \land 0 = a_i = a_j\}
\]

Protocol 3 shows \( \Pi_{\text{PuncUniq}} \). First, \( \mathcal{P} \) sends \( c \): the number of extra 0s. \( \mathcal{P} \) also sends Bézout coefficients \( s, t \) for the duplicated list. \( \nu \) tests \( zs + z't = X^c \) at a random \( \alpha \), evaluating \( z(\alpha), s(\alpha), z'(\alpha), t(\alpha) \) as in \( \Pi_{\text{uniq}} \) (using \( 4C + O(1) \) multiplications) and \( c^\alpha \) using \( O(\log C) \) multiplications.

**Theorem 6.** The protocol \( \Pi_{\text{PuncUniq}} \) (for language \( \mathcal{L}_{\text{PuncUniq}} \)) has perfect completeness, soundness error \( \leq (C + A - 1)/|\mathbb{F}| \), and \( \nu \) complexity \( 4C + O(\log C) \). (proved in Appendix A)

This is already better than a conditional uniqueness proof. But, we can do even better by proving the uniqueness of the valid \( a'_i \) assuming the uniqueness of the valid \( a_i \).

#### 6.2.2. Punctured disjointness

Our first insight is that individual memory proofs do not need to prove valid address uniqueness from scratch. A persistent memory proof is used to allow a chain of zkSNARKs \( \pi_1, \ldots, \pi_B \) to be provably consistent with a single RAM that is represented between proofs by a commitment. Thus, in each proof, the initial \( a_i \) come from a previous proof. So, if each proof assumes
avoid collision-resistant hashing, division-intractable hashing, Wesolowski proofs, and multi-precision arithmetic. However, for very large $C$ (say $C \gg 2^{25}$), Merkle seems best, because only its costs are $C$-independent.

**Our approach.** We can determine the $C$-dependence of our $\mathcal{V}$ complexity precisely. We start from the $3C$ cost of $\Pi_{\text{active}}$. Then, the $H_c$ evaluations in Figure 5 add one multiplication each, and the $\Pi_{\text{PunctDisj}}$ adds another $2C$. Thus, our final $\mathcal{V}$ complexity is $7C + O(\log C + A)$. The $A$-dependence is very similar to a dense RAM; we conservatively estimate it increases by 10. Finally, to handle address commitment, the extra MSMs of CP-Mirage+ grow slightly.

**Merkle.** Sparse merkle trees give a natural (but expensive) construction of sparse persistent RAM: a Merkle tree with $N = |\mathcal{F}| - 1$ leaves. Here, the log $N$ factor of Table 2 is now $\approx 256$, so $\mathcal{V}$ complexity is approximately 150k per access.

**RSA.** One could also build a sparse persistent RAM from RSA accumulators, by combining RSA-based RAM with batched RSA non-membership proofs [30, 58]. The approach would use two RSA accumulators: one for $(m_i, a_i)$ pairs and the other for just the addresses $a_i$. RSA’s constant overhead increases by 10M relative to Table 2 just to update the new address accumulator (two more Wesolowski proofs). Further, addresses are proved unique with batched non-membership proofs, which add another 10M constraints (this is 2456-bit exponentiations, at $\approx 2.5M$ constraints each).

The additional (batched) accumulator operations also increase per-access costs. These costs should approximately double because their bottleneck (a multi-precision modular product of division intractable hashes) is done twice: once for $(m_i, a_i)$ pairs and once for the $a_i$ on their own.

## 7. Mirage+: A zkSNARK for AIPs

Now, we present Mirage+: our zkSNARK for AIPs expressed as I-RICS. We first give I-Mirage+: a zkSARK (zero-knowledge succinct interactive argument of knowledge) for I-RICS. It comprises a non-interactive Setup and two interactive algorithms $\mathcal{P}$ and $\mathcal{V}$ (Protocol 5).

I-Mirage+ generalizes Mirage [48] and Groth16 [3]. We highlight the differences between I-Mirage+ and Groth16 in blue. We first replace Groth’s $P_C$ (which encodes the full witness) with $P_{C,1}, \ldots, P_{C,\mu}$ (which respectively encode messages $w_1, \ldots, w_\mu$). Second, for zero-knowledge, each $P_{C,i}$ must include a fresh blind $\kappa_i$, for $i < \mu$. Third, for completeness, we tweak $P_{C,\mu}$ (adding $-\sum_{i=1}^{\mu-1} \kappa_i G_i$) to cancels these blinds in the verification equation. If $\mu$ is 1 or 2, then I-Mirage+ is equivalent to Groth’s construction or Mirage, respectively. Theorem 8 states the security of I-Mirage+; the proof is in Appendix A.4.

### 6.3. Comparison to alternatives

Now, we explain alternatives to our approach and present cost models for all options. Table 3 shows the cost models. We summarise our conclusions here, and explain the alternative approaches and cost models below.

For moderate $C$, our approach is better than one based on RSA or Merkle. The reason is just as in Section 5.3: we
Setup(φ) → (pk, vk):

sample α, β, γ, δ₁, . . . , δₚ, τ ∈ F \ {0}

for M ∈ {A, B, C}, i ∈ [m]:
    f_M,i(X) ← Interp(ω, M, j=1)

for i ∈ [m]: f_i(X) ← βf_A,i(X) + αf_B,i(X) + f_C,i(X)

Y ← {ℓr + ∑ᵢ=1 ℓr₁}

z(X) ← \prodᵢ=1(X - ωᵢ)

for i ∈ [m]; Wᵢ ← {k + |X| + ∑ⱼ<ᵢ ℓwᵢ} k∈[w₁]

pk contains:
(α, β, δ₁, . . . , δₚ, (τᵢ)ᵢ=1, (δᵢ−₁ τ i(z(τ))ᵢ=0) · G₁
(δ−₁ f_i(τ))ⱼ∈[µ],j∈Wᵢ · G₁; (β, δᵢ, (τᵢ)ᵢ=0) · G₂

vk contains:
(α, γ, δ₁, . . . , δₚ) · G₁; (β, γ, δ₁, . . . , δₚ) · G₂

\[ P(pk, x, w): \]

\[ \text{compute } f_i, f_M,i, z \text{ as in Setup.} \]

for any I ∈ [m], define f_I(X) ≜ \sumᵢ∈I zᵢfᵢ(X)

sample ρ, α, κ₁, . . . , κₚ−1 ∈ F

for round i in [µ − 1]:
    compute wᵢ from (x, w, r₁, . . . , rᵢ−1)
    send to \( P: P_C,i ← (δ_\rho, κ_i + δ_i f_W, i(τ))G₁ \)
    receive rᵢ

compute wᵢ and z ← (x, r₁, . . . , rᵦ−₁, w₁, . . . , wₚ)

for M ∈ {A, B, C}: f_M,i(X) ← \sumᵢ zᵢfᵢ(X)

h(X) ← (f_A,i · f_B,i · f_C,i)/z

define β' ≜ β + σδᵦ + f_B(i)(τ)

send to \( P: P_A ← (α + ρδᵦ + f_A(i)(τ))G₁ \)

\[ P_B ← β'G₂ \]

\[ P_{C,i} ← δᵦ(hz + f_W, i(τ))(τ) + ρβ' - ρσ\delta G₁ \]

\[ -(\sumᵢ=1 κᵦδ_i)G₁ + σ P_A \]

\[ \mathcal{V}(vk, x): \]

for round i in [µ − 1]:
    receive from \( P: P_{C,i}; \) sample and send to \( P: rᵦ \in F \)
    receive from \( P: P_A, P_B, P_{C,i} \)
    define f_I for I ∈ [m] as in \( \mathcal{P} \)
    assert: e(P_A, P_B) = e(αG₁, βG₂) + e(γ−₁ f_V, i(τ)G₁, γG₂) + \sumᵢ=1 e(P_C,i, δ_i G₂)

Protocol 5: I-Mirage+: our zkSARK for I-R1CS. It is a slight generalization of Mirage, itself a generalization of Groth16. Differences from Groth16 are in \text{blue}. Let ω have order n and define Interp(ω, v ∈ Fⁿ) as the unique f ∈ Fⁿ[X] satisfying f(ωᵢ) = vᵢ.

8. Implementation

Cryptosystem. We implement CP-Mirage+, starting from the bellman [84] Rust implementation of Groth’s zkSNARK [3]. Our patch (2.4k LOC and 1.3k LOC for tests) includes the linear subspace proof of Kiltz and Wee [85], the commitment linking proof of Campanelli, Fiore, and Querol [47], a generalization of bellman’s R1CS interface to I-R1CS, and CP-Mirage+ itself.

Compilation and RAM proofs. We implement our volatile and persistent RAM proofs as extensions to the CirC compiler infrastructure [86]. This is a non-trivial extension. First, we generalize CirC’s Intermediate Representation (IR), to represent not just circuits (NP verifiers), but also public-coin IPs. Second, we adapt CirC’s R1CS backend, to compile these IPs to I-R1CS. These changes are about 4k LOC, Rust.

CirC’s Intermediate Representation (IR) can already express programs that manipulate memory. We implement a compiler pass that replaces IR memory operations with uses of our RAM proofs. This compiler pass essentially generates CirC IR that encodes our proofs. We also implement the proofs from TinyRAM [45] and Arya [32], for comparison. In sum, these changes are about 2k LOC.

Programming Interface. Implementing our proofs within CirC allows us to make their benefits very accessible to programmers. We do this by modifying one of CirC’s input languages and one of its compiler driver binaries.

First, we add new keywords to CirC’s Z# input language that trigger our new compiler pass. Figure 6 shows an example program that uses persistent memory via this syntax. The memory is the field array mem, which is an input to the program and is annotated as committed. Standard array access syntax is used for both reads and writes. The compiled CP-zkSNARK will include commitments to the initial and final states of this array. Memory accesses must be in-bounds, or the proof will fail. Volatile memory is available via a similar syntax, but volatile memories must have local scope and are annotated with transcript.

Second, we extend the CirC executable for compiling and using Z# programs. It already had subcommands for compiling a Z# instance to (I-)R1CS, as well as zkSNARK setup, proving, and verifying. We add subcommands for the new CP-zkSNARK operations (commitment setup, creation,
and opening) and extend the existing subcommands to work for a CP-zkSNARK. In some, our modifications to Z# and CirC’s compiler driver are about 1k LOC.

9. Evaluation

We evaluate our zkSNARKs for RAM against prior work. Our metrics are proving time and the number of rank-1 constraints in the I-R1CS, which is platform-independent and correlated with proving time (Sec. 7). In a few experiments, we also measure Setup time (expensive but run only once per application), V time (concretely small for all systems we benchmark) and proof size (also small).

9.1. Volatile memory

We compare our volatile memory proof, (§4; “Uniq”), against three baselines: TinyRAM [45], Arya [32], and Haböck [87] (Table 1). All proofs are about A accesses to a memory of size N, with no further computation.

Figure 7 shows constraint counts for \( N \in \{2^{20}, 2^{40}, 2^{60}\} \) and \( A \in \{2^5, \ldots, 2^{10}\} \). Uniq reduces constraints (relative to the best baseline) by up to 20%, at \( A = 2^5 \) and \( N = 2^{60} \). This is the expected location for the optimum (large \( N \), small \( A \)), since Arya and Haböck have worse \( N \)-scaling (Sec. 4.5). Uniq’s concrete advantage is modest because the \( N \)-dependence of the baselines is modest, and because Uniq only optimizes one part (ordering) of the overall RAM proof.

Figure 8 compares proving times. Proving times increase less smoothly than constraint counts because they depend on Fourier domain sizes [4] and noise. Uniq reduces proving time by up to 32.9%. Importantly, for both proving time and constraint count, Haböck is sometimes better than Uniq, so the best choice for an application depends on \( N \) and \( A \).

9.2. Persistent memory

We compare our persistent memory proof (“C&P”; §5) against Merkle trees [88] (“Merkle”), RSA accumulators [30] (“RSA”), and a naive linear scan (“Scan”). For “Merkle” and “RSA” we use the implementation of Ozdemir et al. [30].

Figure 9 shows constraint counts for \( N \in \{2^{10}, 2^{15}, 2^{20}\} \) and \( A \in \{2^5, \ldots, 2^{12}\} \). At \( N = 2^{15} \), C&P reduces constraints by up to a factor of 98.2×, (at \( A = 2^{11} \)). For smaller \( A \) or larger \( N \), C&P’s advantage shrinks relative to Merkle. For very small \( N \), C&P’s advantage shrinks relative to a naive linear scan. For large \( A \), the RSA approach large constant overhead (Sec. 4.5) begins to amortize away, shrinking C&P’s relative advantage. Of course, since C&P’s per-\( A \) cost is lower than RSA’s (the latter includes a division-intractable hash), C&P will be still be better for any \( A \).

We also measure \( \mathcal{P} \) time, \( V \) time, Setup time, and proof size. Figure 10 shows \( \mathcal{P} \) time for \( N \in \{2^{15}, 2^{20}\} \) and
Table 4: Costs for a persistent memory of size $N = 2^{20}$.

| Approach | $\log N$ | $A$ | Setup (s) | $P$ (s) | $|\pi|$ (B) | $V$ (s) | Extra (s) |
|----------|----------|-----|-----------|---------|------------|---------|-----------|
| C&P      | 15       | 3000 | 28        | 2       | 500        | 0.009   | 2         |
| RSA      | 15       | 3000 | 1214      | 99      | 308        | 0.009   | 142       |
| RSA      | 20       | 3000 | 1216      | 140     | 308        | 0.006   | 146       |
| Merkle   | 15       | 3000 | 2087      | 208     | 308        | 0.007   | 693       |
| Merkle   | 20       | 3000 | 2914      | 328     | 308        | 0.010   | 912       |

Figure 11: Modelled constraint counts for sparse persistent memory proofs.

$A \in \{1, 500, 1000, \ldots, 3000\}$. We omit Scan, which is unreasonably expensive. For a memory of size $2^{15}$, C&P reduces $P$ time by 51.3×, (at $A = 2^{11}$). This is slightly less than the constraint reduction, but still very significant. We think that the improvement in proving time is slightly degraded for two reasons. First, this work is the extra commitment linking protocol used by CP-Mirage+ (Sec. 2). Second is the increased bit size of prover witness in C&P relative to RSA. In RSA, many witnesses are small values, but in C&P many witnesses depend on 255-bit (uniformly random) $V$ challenges. This slows down the bellman library’s MSMs and—by extension—proving.

Table 4 shows all metrics for this experiment. Setup and $P$ time are both highly correlated with constraint counts. A C&P proof is slightly larger than Merkle or RSA, but is still only 500 bytes. $V$ time is small for all systems.

9.3. Sparse persistent memory

We compare the estimated costs of our sparse persistent memory proof (“C&P”, §5) against the cost of alternate approaches based on Merkle trees [88] (“Merkle”), RSA accumulators [30] (“RSA”), and a naive linear scan (“Scan”). In all cases, we compute the number of constraints based on our cost model in Table 3.

Figure 11 shows constraints for $C \in \{2^{10}, 2^{15}, 2^{20}\}$ and $A \in \{2^5, \ldots, 2^{12}\}$. C&P reduces constraints by up to a factor of 143×, at $C = 2^{17}$ and $A = 2^{12}$.

We implement a simple zk-Rollup [89] (a zkSNARK about $A$ transactions in a payment system) using our persistent RAM. We use CirC’s Z# input language. Figure 12 shows the Z# entry function for our rollup. There are two arguments. The first is a committed array of accounts. Each account is three scalars: the owner’s public key (two field elements) and a balance (one field element) between 0 and MAX_BALANCE. The second is a list of transactions. Each transfers some amount of money from one public key to another. It includes a signature and the indices of the relevant public keys in the array. The function main asserts the validity of all transactions and updates the account balances.

Figure 13 shows the constraint counts for the rollup as a function of $A$ (the number of accesses) and $N$ (the number of accounts). We benchmark the three persistent RAMs from the prior subsection. C&P’s improvement is greatest when the number of accesses is high and the RAM size is high, but not too high. C&P’s relative advantage here is less than in Figure 9 because the cost of verification is small (identical in all systems) is large. Despite that, C&P reduces constraint count (relative to prior state of the art) by up to 2.55×.

10. Conclusion

We have presented new AIPs for the consistency of accesses to volatile, persistent, and sparse persistent memories. Our AIPs are based on proofs for different kinds of uniqueness and disjointness that rely on Bézout’s identity for univariate polynomials. Our AIPs can be compiled into zkSNARKs using CP-Mirage+. We expect that proofs of uniqueness and disjointness, as well as our approach of
compiling AIPs into zkSNARKs will have utility beyond memory-checking.

References


D. Heath and V. Kolesnikov, “PrORAM - fast $P(\log n)$ authenticated shares ZK ORAM.” 2021.


D. Harvey and J. Van Der Hoeven, “Polynomial multiplication over finite fields in time $\text{O}(\text{log}^2 n)$.” JACM, 2022.


Appendix A.
Deferred proofs

A.1. Theorem 5: $\Pi_{\text{active}}$

We restate and prove Theorem 5:

**Theorem.** If $A \leq N$, the protocol $\Pi_{\text{active}}$ (for language $L_{\text{active}}$) has perfect completeness, soundness error $\leq (N^2 + 2N + A)/|F|$, and $\mathcal{V}$ complexity $3N + 2A + O(1)$.

**Proof.** We prove completeness, then soundness, and then compute $\mathcal{V}$ complexity.

**Completeness.** Define $I = \{i \in [N] : i \in a\}$ and define $J = [N] \setminus I$. Since the $a_i$ are distinct, $|I| = A$ and $|J| = N - A$, so $h$ has length $N - A$ as required. Now, we will show that $\mathcal{V}$’s first test passes. First, disjointness of $I$ and $J$ gives:

$$\prod_{i \in I \cup J} (\alpha + m_i + i\beta) = \prod_{i \in I} (\alpha + m_i + i\beta) \times \prod_{i \in J} (\alpha + m_i + i\beta)$$

Then, re-indexing the products gives:

$$\prod_{i=1}^{N} (\alpha + m_i + i\beta) = \prod_{i \in [N], i \notin a} (\alpha + m_i + i\beta) \times \prod_{i=1}^{A} (\alpha + v_i + a_i\beta)$$

Which, using the definition of $h$, gives:

$$\prod_{i=1}^{N} (\alpha + m_i + i\beta) = \prod_{i=1}^{N-A} (\alpha + h_i) \times \prod_{i=1}^{A} (\alpha + v_i + a_i\beta)$$

which is $\mathcal{V}$’s first test. Applying this argument with $m'$ and $v'$ instead of $m$ and $v$ shows that $\mathcal{V}$’s other test succeeds too.

**Soundness.** Suppose that the instance is not in the language, and that $\mathcal{V}$ accepts. We will show that this is unlikely.

First, consider $\mathcal{V}$’s tests as equalities between polynomials in $\alpha$. Per the fundamental theorem of algebra, these equalities hold except with probability $\leq 2N/|F|$, so the factorizations must be equal; that is, these multisets must be equal:

$$\{m_i + i\beta\}_{i=1}^{N} = \{h_i\}_{i=1}^{N-A} \lor \{v_i + a_i\beta\}_{i=1}^{A}$$

$$\{m'_i + i\beta\}_{i=1}^{N} = \{h_i\}_{i=1}^{N-A} \lor \{v'_i + a_i\beta\}_{i=1}^{A}$$

Now, consider the left multisets. The $(m_i, i)$ are all distinct, so the $m_i + i\beta$ must all be distinct as well, except with probability $(2N)/|F|$. Similarly for the $(m'_i, i)$, except with probability $(2N)/|F|$. Thus our multiset equalities are set equalities and the unions are disjoint.

$$\{m_i + i\beta\}_{i=1}^{N} = \{h_i\}_{i=1}^{N-A} \lor \{v_i + a_i\beta\}_{i=1}^{A}$$

$$\{m'_i + i\beta\}_{i=1}^{N} = \{h_i\}_{i=1}^{N-A} \lor \{v'_i + a_i\beta\}_{i=1}^{A}$$

That implies the existence of injection $\pi : [A] \rightarrow [N]$ such that for all $i$, $m_{\pi(i)} + \pi(i)\beta = v_i + a_i\beta$. And similarly, the existence of injection $\pi' : [A] \rightarrow [N]$. Now, we have that for all $i$, $(m_{\pi(i)}, \pi(i)) = (v_i, a_i)$ and similarly for the primed terms, except with probability $2A/|F|$.

Since $\pi$ is an injection, this implies that the $a_i$ are all distinct. Further, this implies that for all $i$, $v_i = m_{a_i}$ and $v'_i = m'_{a_i}$.

Next, from (3) and the definition of $\pi$, we have the existence of a bijection $\rho$ from $[N] \setminus \text{Im}(\pi)$ to $[N - A]$ such that $m_i + i\beta = h_{\rho(i)}$. And similarly, we have a $\rho'$. Let $\tau = \rho'^{-1} \circ \rho$. Then we have that for all $i \in \text{Dom}(\rho)$, $m_i + i\beta = m'_{\tau(i)} + \tau(i)\beta$. Except with probability $(N - A)/|F|$, this implies that $(m_i, i) = (m'_{\tau(i)}, \tau(i))$. That, in turn, implies that $m_i = m'_i$. On the other hand, if $i \notin \text{Dom}(\pi)$, then it is in the image of $\pi$, so $i \in a$. Either way, we have that $m_i = m'_i \vee i \in a$.

Thus, we’ve shown that if $\mathcal{V}$’s tests succeed, then the instance is in the language, except with probability

$$\leq \frac{2N}{|F|} + \frac{2A}{|F|} + \frac{N - A}{|F|} = \frac{N^2 + 2N + A}{|F|}$$

**Complexity.** Computing $c$ requires $N - A + O(1)$ non-linear multiplications. Then, the first test requires $N + 2A + O(1)$. However, the second test requires only $N + A + O(1)$ new non-linear multiplications because all the $a_i\beta$ products were already computed for the first test. Thus, $\mathcal{V}$ complexity is $3N + 2A + O(1)$.

A.2. Theorem 6: $\Pi_{\text{PuncUniq}}$

We restate and prove Theorem 6:

**Theorem.** The protocol $\Pi_{\text{PuncUniq}}$ (for language $L_{\text{PuncUniq}}$) has perfect completeness, soundness error $\leq (C + A - 1)/|F|$, and $\mathcal{V}$ complexity $4C + O(\log C)$.

**Proof.** We prove completeness, then soundness, then $\mathcal{V}$ complexity.
Completeness. Suppose a ∈ L_{PuncUniq}. Let d be the number of occurrences of 0 in a. Then, e = max(0,d − 1). Let v(X) = \prod_{s \in S}(X − s). Then, gcd(v, v′) = 1, and UniqBez returns s, t such that vs + vt = 1 (per the correctness of UniqBez; Theorem 4). Further, since z(X) = Xcv(X), we can show that z′(X) = Xcv′(X). (Note that, since v may contain a factor X, showing this requires a little case-work.) Then, we have zs + z′t = Xc. Then, V’s test succeeds with probability 1.

Soundness. Suppose a ∈ L_{PuncUniq}. Then some y \neq 0 occurs in a at least twice. Then, X − y divides gcd(z, z′). So, there are no s, t such that zs + z′t is indivisible by X − y, and in particular zs + z′t cannot equal Xc. Now, consider the degree of the polynomial equation. The polynomials f, g have degrees C and A respectively. The polynomials s, t have degrees at most A − 1 and C − 1 respectively. Further, e can be at most C. Thus, the equation’s degree is at most C + A − 1.

V complexity. Per the analysis of Uniq, z, z′, s, t can be evaluated with 4C + O(1) multiplications. Then, repeated squaring can evaluate αc in O(log C) multiplications. □

A.3. Theorem 7: I_{PuncDisj}

We restate and prove Theorem 7:

Theorem. The protocol I_{PuncDisj} (for language L_{PuncDisj}) has perfect completeness, soundness error \leq (C + A − 1)/|P|, and V complexity 2C + 2A + O(log C).

Proof. We prove completeness, then soundness, then V complexity.

Completeness. Suppose (a, b) ∈ L_{PuncDisj}. Then, I contains only 0s, and S′ and T′ are disjoint. Define \bar{f}(X) = \prod_{s \in S}(X − s) and \bar{g}(X) = \prod_{t \in T}(X − t). Then, s, t satisfy \bar{f} + t\bar{g} = 1 (per the correctness of DisjBez). Define c as in the protocol. Then, \bar{f} = \bar{f}Xc and \bar{g} = \bar{g}Xc. sf + tg = Xc. Then, V’s test succeeds with probability 1.

Soundness. Suppose (a, b) ∈ L_{PuncUniq}. Then some y \neq 0 occurs in a and b. Then, X − y divides gcd(z, z′). So, there are no s, t such that zs + z′t is indivisible by X − y, and in particular zs + z′t cannot equal Xc. Now, consider the degree of the polynomial equation. The polynomials f, g have degrees C and A respectively. The polynomials s, t have degrees at most A − 1 and C − 1 respectively. Further, e can be at most C. Thus, the equation’s degree is at most C + A − 1. So, the soundness error is at most (C + A − 1)/|P|.

V complexity. Per their degrees, f, g, s, t can be evaluated in 2C + 2A − 2 multiplications, and αc requires O(log C) multiplications to compute. □

A.4. Theorem 8: I-Mirage+

We restate and prove Theorem 8:

Theorem. Let I-RICS φ be complete and knowledge-sound for witness relation R. Then in the generic group model, I-Mirage+ is a zkSARK for R.

Proof. We prove completeness, zero-knowledge, and knowledge-soundness separately.

Completeness. We will show that verification succeeds with overwhelming probability when P follows the protocol. The completeness of φ as an AIP implies that (with overwhelming probability), P constructs z satisfying \( A_z \circ B_z = C_z \). Now, it suffices to show that verification equation for I-Mirage+ holds. Consider that equation:

\[
e(P_A, P_B) = e(\alpha G_1, \beta G_2) + e(\gamma^{-1} f'_X(\tau) G_1, \gamma G_2) + \sum_{i=1}^{r} e(P_{C,i}, \delta_i G_2)
\]

Taking the logarithm of both sides with respect to the generator e(G_1, G_2) gives:

\[
(\alpha + \rho \delta_\mu + f_A(\tau))(\beta + \sigma \delta_\mu + f_B(\tau)) = \alpha \beta + (\gamma^{-1} f'_X(\tau)) \gamma + \sum_{i=1}^{\mu-1} (\delta_\mu \kappa_i + \delta'_i f'_W(\tau)) \delta_i + (\sigma \alpha' + \rho \beta' - \rho \sigma \delta_\mu + \delta'_i h + f'_W(\tau)) (\tau) - \sum_{i=1}^{\mu-1} \kappa_i \delta_i \delta_\mu
\]

where \( \alpha' = \alpha + \rho \delta_\mu + f_A(\tau) \). Expanding the products and the definitions of \( f'_W, f'_X \) gives:

\[
f_A(\tau) f_B(\tau) = f_C(\tau) + h(\tau) z(\tau)
\]

This follows if the following polynomial equality holds:

\[
f_A(X) f_B(X) = f_C(X) + h(X) z(X)
\]

This holds if

\[
z(X) \mid (f_A(X) f_B(X) - f_C(X))
\]

This holds if the right is zero at all zeros of z; i.e.,

\[
\forall i \in [n], f_A(\omega^i) f_B(\omega^i) = f_C(\omega^i)
\]

\[
\forall i \in [n], \left( \sum_{j=1}^{m} z_j A_{j,i} \right) \left( \sum_{j=1}^{n} z_j B_{j,i} \right) = \left( \sum_{j=1}^{m} z_j C_{j,i} \right)
\]

This follows from \( A_z \circ B_z = C_z \).

Zero-Knowledge. We construct a simulator S that, given \( x, \) outputs \( \pi \) (a V view for I-Mirage+), pk, and vk. First, S runs Setup, and saves the scalars \( \alpha, \beta, \gamma, \delta_1, \ldots, \delta_r, \tau \). Second, S samples \( r_1, \ldots, r_{\mu-1} \) uniformly. Third, S samples the group elements \( P_A, P_B, P_{C,1}, \ldots, P_{C,\mu-1} \) by sampling their exponents \( a, b, c_1, \ldots, c_{\mu-1} \). Fourth, S sets

\[
P_{C,\mu} = \delta_\mu^{-1} (ab - \alpha \beta - f'_X(\tau) - \sum_{i=1}^{\mu-1} c_i \delta_i) G_1
\]

(where \( f'_X \) is defined as in Verify). Finally, S outputs \( \langle r_1, \ldots, r_{\mu}, P_A, P_B, P_{C,1}, \ldots, P_{C,\mu}, \rangle, pk, vk \).

Now, we show that the output of S is distributed identically as in I-Mirage+. First, pk, vk, and the \( r_i \) are constructed identically in both. Second, all group elements but \( P_{C,\mu} \) are uniform and independent in both. Finally, in both, \( P_{C,\mu} \) is the unique solution to the verification equation. So I-Mirage+ has perfect zero-knowledge.
Knowledge Soundness. First, we sketch the proof. Our analysis is similar to Groth’s—without intermediate notions like split NILPs [3]. We also use the GGM to extract “witnesses” from \( P' \)’s group elements. In Groth, these witnesses are the solution to an R1CS. For us, they are AIP transcripts, that (via the AIP extractor) give an actual witness for \( R \). Groth used the equivalence of Laurent polynomials to argue R1CS satisfaction; we will use it to argue AIP transcript validity. Just as in Groth, this equivalence requires Setup to hide the scalars that it samples. Our analysis is in the bilinear version of Shoup’s GGM [90], with \(| S|/|G| = \Omega(2^s)\), where \( S \) is the encoding space.

Let \( \mathcal{V} \) be the verifier for an AIP that is knowledge-sound for \( R(x, w) \). Thus, there is an extractor \( \mathcal{E} \) that computes \( w \) from \( x \) and oracle access to a convincing prover \( \mathcal{P} \). We define an extractor \( \mathcal{E}' \) that gives \( x \) and oracle access to a I-Mirage+ prover \( \mathcal{P}' \), computes \( w \). Let \( \mathcal{V}' \) be the verifier for I-Mirage+. \( \mathcal{E}' \) interacts with \( \mathcal{P}' \) and with \( \mathcal{E} \) (to which it imitates \( \mathcal{P} \)). First, \( \mathcal{E}' \) receives \( P_{C,1} \) from \( \mathcal{P}' \). Since \( \mathcal{P}' \) is an adversary in the GGM, with an encoding space that is exponentially larger than the group, \( \mathcal{P}' \) can only obtain valid group elements by interacting with the group oracle (except with negligible probability). Thus, by observing the oracle queries of \( \mathcal{P}' \), \( \mathcal{E}' \) can compute scalar explanations of \( P_{C,1} \) [3]. That is, it can compute scalars \( C_{1,1}, C_{1,\beta}, (C_{1,i}\delta_i)_{i \in [r]}, (C_{i,i})_{i = 1}^{N-1} \), and scalar polynomials \( A(\tau) \in \mathbb{F}_p^{<N}[\tau], A_2(\tau) \in \mathbb{F}_p^{<N-1}[\tau] \) such that

\[
\log_{G_1}(P_{C,1}) = \log(C_{1,\alpha})
= C_{1,\alpha} + \sum \left( C_{i,i} \delta_i + \tau C_{i,i} \right)
+ \sum_{i \in X} \tau \left( \beta f_{A,i}(\tau) + \alpha f_{B,i}(\tau) + f_{C,i}(\tau) \right)
+ \sum_{j=1}^{\mu} \tau \left( \beta f_{A,j}(\tau) + \alpha f_{B,j}(\tau) + f_{C,j}(\tau) \right).
\]

In the future, we will refer to the collection of scalars and polynomials that represent group element \( P_{x} \) in terms of \( pk, vk \), as \( P_{x,x} \), and use the function \( \log(P_{x,x}) \) to refer to the linear combination of exponents in the SRS according to such a collection. One proof component, \( P_{B} \) is a \( G_2 \) element, so its explanation has form:

\[
\log_{G_2}(P_{B}) = \log(B_{\gamma}) = B_{\beta} \beta + B_{\gamma} \gamma + \sum_{i=1}^{r} B_{\delta_i} \delta_i + B(\tau)
\]

Now, \( \mathcal{E}' \) sends \( w_1 \leftarrow (C_{1,i})_{i \in W_1} \) as the first message to \( \mathcal{E} \). When \( \mathcal{E} \) replies with \( r_1 \), \( \mathcal{E}' \) forwards it to \( \mathcal{P}' \). \( \mathcal{E}' \) handles future \( P_{C,1} \) and \( r_1 \) in the same way. (During this process, it gets explanations \( C_{2,x}, \ldots, C_{r,x} \) too. It also gets explanations \( A_{x} \) and \( B_{x} \) for \( P_{A} \) and \( P_{B} \). At the end, \( \mathcal{E}' \) outputs the same \( w \) as \( \mathcal{E} \).

Now, by the knowledge soundness of \( \mathcal{V} \), we have that

\[
\Pr[w \text{ valid}] \geq \Pr[\mathcal{E}' \text{ convinces } \mathcal{V}] - \text{negl}
\]

We must show that

\[
\Pr[w \text{ valid}] \geq \Pr[\mathcal{P}' \text{ convinces } \mathcal{V}'] - \text{negl}
\]

Thus, it suffices to show that in the case that \( \mathcal{P}' \) convinces \( \mathcal{V}' \), \( \mathcal{E}' \) convinces \( \mathcal{V} \), except with negligible probability. In this case, the verification equation for I-Mirage holds, so we have equality of exponents:

\[
\text{lc}(A_{\alpha}) = \alpha \beta + \sum_{i=1}^{T} \delta_i \text{lc}(C_{i,\alpha})
\]

(1)

If this is true, then with the \( r_i \) and \( w_i \) that \( \mathcal{E}' \) computes, we will show that \( Az = Cz \) holds, where \( z = (x, r_1, \ldots, r_{\mu-1}, w_1, \ldots, w_\mu) \). First, we observe that (except with negligible probability), Equation 1 must hold as a Laurent polynomial equation in over variables sampled during Setup. The reason for this is exactly the reason given in Groth’s analysis [3, Lemma 1]: because \( pk \) and \( \lambda \) leak no information about Setup’s scalars (except with negligible probability). We omit this reasoning here, since it essentially the same as that given by Groth, with Groth’s \( \delta \) replaced by the \( \delta_i \). So, we have that Equation 1 holds as a polynomial identity; it holds coefficient-by-coefficient.

By the equality of \( \alpha \beta \) coefficients, we have \( A_{\alpha}B_{\beta} = 1 \). Then, we can multiply the \( A_{\alpha} \) by \( B_{\beta}/A_{\alpha} \) and divide the \( B_{\beta} \) by the same value to preserve Equation (1) and ensure that:

\[
A_{\alpha} = 1 \land B_{\beta} = 1 \tag{2}
\]

By the equality of \( \alpha \gamma \) coefficients, (and Eq. 2) we have

\[
A_{\alpha}B_{\gamma} = 0 \implies B_{\gamma} = 0 \tag{3}
\]

By the equality of \( \beta^2 \) coefficients, (and Eq. 2) we have

\[
A_{\beta}B_{\beta} = 0 \implies A_{\beta} = 0 \tag{4}
\]

By the equality of coefficients of monomials involving \( \beta \delta_j^{-1} \), for all \( j \in [r] \), we have

\[
B_{\beta} \sum_{i \in W_j} A_{j_i} f_{A,i}(\tau) = 0
\]

Similarly, by the equality of monomials with \( \beta^2 \gamma^{-1} \), we have

\[
B_{\beta} \sum_{i \in X} A_{j_i} f_{A,i}(\tau) = 0
\]

Putting these two together, we have

\[
A_{1} = 0 \land \cdots \land A_{m} = 0
\]

From the coefficients of \( \beta \delta_j^{-1} \), as polynomials in \( \tau \), we have

\[
B_{\beta} A_{\xi}(\tau) z(\tau) = 0 \implies A_{\xi}(\tau) = 0
\]

So, we have:

\[
\text{lc}(A_{\alpha}) = \alpha + \sum_{i=1}^{r} A_{i} \delta_i + A(\tau)
\]

\[
\text{lc}(B_{\alpha}) = \beta + \sum_{i=1}^{r} B_{i} \delta_i + B(\tau)
\]
We now consider the equality of terms involving $\delta_j \gamma^{-1} \tau^i$, for $j \in [r]$; their coefficients must be equal:
\[
0 = \delta_j \sum_{i \in X} C_{i,j} f_{C,i}(\tau)
\]
\[
\Rightarrow \forall j \in [r], \forall i \in X, C_{i,j} = 0
\]
Now, the remaining $\alpha \tau^i$ terms give:
\[
B(\tau) = \sum_{i \in X} z_i f_{B,i}(\tau) + \sum_{j=1}^n \sum_{i \in W_j} C_{i,j} f_{B,i}(\tau)
\]
which, with our definition of $w_j$ and $z$ is just:
\[
B(\tau) = \sum_{i=1}^m z_i f_{B,i}(\tau)
\]
Similarly, the $\beta \tau^i$ terms give an expression for $A(\tau)$:
\[
A(\tau) = \sum_{i=1}^m z_i f_{A,i}(\tau)
\]
And, (mod $z(\tau)$) the $\tau^i$ terms give:
\[
\left( \sum_{i=1}^m z_i f_{A,i}(\tau) \right) \left( \sum_{i=1}^m z_i f_{B,i}(\tau) \right) = \left( \sum_{i=1}^m z_i f_{C,i}(\tau) \right)
\]
Which holds at $\omega^j$ (a root of $z$) for all $j \in [n]$:
\[
\left( \sum_{i=1}^m z_i f_{A,i}(\omega^j) \right) \left( \sum_{i=1}^m z_i f_{B,i}(\omega^j) \right) = \left( \sum_{i=1}^m z_i f_{C,i}(\omega^j) \right)
\]
\[
\left( \sum_{i=1}^m z_i A_{i,j} \right) \left( \sum_{i=1}^m z_i B_{j,i} \right) = \left( \sum_{i=1}^m z_i C_{j,i} \right)
\]
\[
A z \circ B z = C z
\]
This concludes our GGM proof. □

**Appendix B.**

**CP-Mirage+ construction**

Let the relation $R_{\text{link}}$ be
\[
R_{\text{link}} = \left\{ (c_1, c_1')^k_{i=1}, (w_i, o_i, o'_i)_{i=1}^k : \bigwedge_{i=1}^k c_i = \text{Com}(\text{gens}, w_i, o_i) \land c'_i = \text{Com}(\text{gens}, w_i, o'_i) \right\}
\]

$\text{CP}_{\text{link}}$ is a zkSNARK for $R_{\text{link}}$ with the following interface:
- $\text{Setup}(\text{gens}, \text{gens}^k_{i=1}) \rightarrow (pk_{\text{link}}, vk_{\text{link}})$: generate proving and verifying keys.
- $\text{Prove}(pk_{\text{link}}, (w_i, o_i, o'_i)_{i=1}^k) \rightarrow \pi_{\text{link}}$: for $c_i = \text{Com}(\text{gens}, w_i, o_i)$ and $c'_i = \text{Com}(\text{gens}, w_i, o'_i)$, generate a proof that $R_{\text{link}}[(c_1, c_1')^k_{i=1}, (w_i, o_i, o'_i)_{i=1}^k]$ holds.
- $\text{Verify}(vk_{\text{link}}, (c_i, c'_i)_{i=1}^k, \pi_{\text{link}}) \rightarrow \{0, 1\}$: check the proof.

We present the full construction of CP-Mirage+ in Protocol 6. The main idea is to use $\text{CP}_{\text{link}}$ to prove that each prover message $P_{C,i}$ is a commitment to the same witness as each $c_i$ provided in the instance. The generators used by $\text{CP}_{\text{link}}$ are those used for the external commitments, and the implicit generators used to generate each $P_{C,i}$. The security of the scheme is straight-forward given the security of I-Mirage+ and $\text{CP}_{\text{link}}$.

---

**Appendix C.**

**AIPs for memory**

In this appendix, we describe our memory proofs in full, as AIPs. We do not give explicit I-1RICS instances for them. When compiled to I-1RICS, most of our proofs acquire one additional round (Sec. 7).

**C.1. Volatile memory**

Protocol 7 ($\Pi_{\text{Vmem}}$) is our volatile memory proof. It uses sub-proofs for non-zero testing, permutations, conditional ordering, conditional uniqueness, and an ordered read-overwrite (RoW) property. Our contribution is the use of a conditional uniqueness proof and the construction of such a proof. We give the language for conditional uniqueness and our construction in Section 4.4. The other sub-proofs
Proof. We assume the following lemmas about protocols

\[ \Pi_{\neq 0}(g, d \in \mathbb{F}^n) \]

\[ \mathcal{P}(\cdots) \quad \mathcal{V}(\cdots) \]

\[ r \leftarrow (\text{ite}(d_i = 0, 0, d_i^{-1}))_{i=1}^n \]

\[ \forall i \in [n] : r_i d_i = g_i \]

\[ \forall i \in [n] : (1 - g_i) r_i = 0 \]

Protocol 8: Proof that shows \( g \in \{0, 1\}^n \) and that each \( g_i \) is 1 iff \( d_i \) is nonzero.

\[ \Pi_{\text{perm}}(A, B \in \mathbb{F}^{n \times k}) \]

\[ \mathcal{P}(\cdots) \quad \mathcal{V}(\cdots) \]

\[ \text{sample } \alpha, \beta \in \mathbb{F} \]

\[ H_r(\alpha, (H_c(\beta, A_i))_{i=1}^n) = H_r(\alpha, (H_c(\beta, B_i))_{i=1}^n) \]

Protocol 9: A proof that \( A \)'s rows are a permutation of \( B \)'s, using universal hashing. \( A_i \) denotes the \( i \)-th row of \( A \).

and their languages are adaptations from prior work. The languages are:

\[ L_{\neq 0} = \{ (g, d \in \mathbb{F}^n) : \]

\[ \bigwedge_{i=1}^n g_i \in \{0, 1\} \land (g_i = 1 \implies d_i \neq 0) \} \]

\[ L_{\text{perm}} = \{ (A, B \in \mathbb{F}^{n \times k}) : \{A_i\}_{i=1}^n = \{B_i\}_{i=1}^n \} \]

\[ L_{\text{ord}} = \{ (d \in \mathbb{F}^n, g \in \{0, 1\}^n) : \]

\[ \bigwedge_{i=2}^n g_i = 1 \implies d_i > d_{i-1} \} \]

\[ L_{\text{ordRoW}} = \{ (g, v, w) : \forall i \in [A], \]

\[ (w_i = 0) \implies (v_i = \text{ite}(g_i = 0, v_{i-1}, 0)) \]

\[ v_0 = 0 \}

Note that \( A_i \) denotes row \( i \) from matrix \( A \).

The proofs themselves are \( \Pi_{\neq 0} \) (Protocol 8), \( \Pi_{\text{perm}} \) (Protocol 9), \( \Pi_{\text{ord}} \) (Protocol 10), and \( \Pi_{\text{ordRoW}} \) (Protocol 11). \( \Pi_{\neq 0} \) is due to Braun [91]. The \( \Pi_{\text{perm}} \) proof is a combination of universal hashing for sequences \((H_r)\) and for multisets \((H_r)\). \( H_r \) (also known as a “polynomial fingerprint”) was first applied to cryptography by Neff [54], to cryptographic proof systems by Bayer and Groth [92], and to memory proofs by Bootle et al. [32]. The \( \Pi_{\text{ord}}, \Pi_{\text{ordRoW}}, \) and \( \Pi_{\text{range}} \) (used by \( \Pi_{\text{ord}} \)) are due to Ben-Sasson et al. [45]. However, while they instantiated \( \Pi_{\text{range}} \) with the bit-based \( \Pi_{\text{range,bit}} \), we instantiate it with \( \Pi_{\text{range,log}} \) [34]. We discuss both options in Appendix C.2.2.

Now, we state and prove the correctness of \( \Pi_{\text{vmem}} \).

Theorem 9. The protocol \( \Pi_{\text{vmem}} \) (for language \( L_{\text{vmem}} \)) has perfect completeness and soundness error \( \leq 3n/|\mathbb{F}| + n/(|\mathbb{F}| - n) \).

Proof. We assume the following lemmas about protocols from prior work:

- \( \Pi_{\neq 0} \) has perfect completeness and soundness.
- \( \Pi_{\text{ord}}(\mathbb{x} \in [l, u]^n, g \in \{0, 1\}^n, l, u) \)

\[ \mathcal{P}(\cdots) \quad \mathcal{V}(\cdots) \]

\[ \text{range}((g_2(x_2 - x_1 - 1), \ldots, g_n(x_n - x_{n-1} - 1)), u - l - 2) \]

Protocol 10: Proof that \( g_1 = 1 \) implies \( x_i > x_{i-1} \). Assumes that all \( x_i \) are in a range \([l, u]\), with \((u - l - 1)n\) less than \( \mathbb{F} \)'s characteristic.

\[ \Pi_{\text{ordRoW}}(g \in \{0, 1\}^A, v \in \mathbb{F}^A, w \in \{0, 1\}^A) \]

\[ \mathcal{P}(\cdots) \quad \mathcal{V}(\cdots) \]

\[ v_0 \leftarrow 0 \]

\[ \forall i \in [A] : (1 - w_i)(v_i - (1 - g_i)v_{i-1}) = 0 \]

Protocol 11: Proof of read-over-write (RoW) semantics for an in-order memory transcript.

- \( \Pi_{\text{perm}} \) has perfect completeness and soundness error \( \leq (k - 1)n/|\mathbb{F}| \). (This follows from Schwartz-Zippel lemma [93–95].)
- \( \Pi_{\text{ord}} \) has perfect completeness and soundness error \( \leq n/(|\mathbb{F}| - n) \)
- \( \Pi_{\text{ordRoW}} \) has perfect completeness and soundness.

Completeness. First, observe that the definition of \( g' \) implies that \( g_1 = 1 \), so \( V \)'s first test passes. Now, it suffices to show that all sub-proofs have arguments that are in their respective languages. Per the definition of \( g' \), \( \Pi_{\neq 0} \)'s argument is in \( L_{\neq 0} \). Per the construction of \( tr' \) as a sort of \( tr \), it is a permutation of \( tr \). Per the construction of \( tr' \) as an \((a, t)\)-sort of \( tr \) and per the uniqueness of the \( t \) values, if for some \( i, a_i = a_{i-1} \), then \( t_i > t_{i-1} \). Thus, \((a', g') \in L_{\text{ord}} \).

Finally, since \( tr \) respects read-over-write (i.e., \( tr \in L_{\text{vmem}} \)), its \((a, t)\)-ordering \( tr' \), together with \( g' \), is in \( L_{\text{ordRoW}} \).

Soundness. Suppose that \( tr \) does not respect read-over-write. Then, there exist \( i < j \) such that (a) \( a_i = a_j \), (b) \( w_j = 0 \), (c) \( v_i \neq v_j \), and (d) there is no intermediate \( k \) between \( i \) and \( j \) such that \( a_k = a_i \). Now, if \( tr' \) is not a permutation of \( tr' \), then \( \Pi_{\text{perm}} \) fails except with probability \( 3n/|\mathbb{F}| \) (note: we instantiate \( \Pi_{\text{perm}} \) with \( k = 4 \)). Assume it is a permutation. Now, if \( \Pi_{\text{uniq}} \) and \( \Pi_{\text{ord}} \) do not fail, then \( tr' \) is \( a \)-grouped and within groups strictly \( t \)-ordered, except with probability \( \leq n/(|\mathbb{F}| - n) \). Assume they are.

Then, since there is no intermediate \( k \), \( acc_i \) and \( acc_j \) are adjacent in \( tr' \), say the former is at index \( i' \). Then we have \( a_{i'} = a_{i'+1} \), so \( g_{i'+1} = 0 \) (or \( \Pi_{\neq 0} \) would fail). Also we have \( w_{i'+1} = 0 \). And finally, we have \( v_{i'} = v_{i'+1} \). These three facts imply that \( \Pi_{\text{ordRoW}} \) fails.

All of this was except with probability

\[ \frac{3n}{|\mathbb{F}|} + \frac{n}{|\mathbb{F}| - n} \]

so that is the soundness error of \( \Pi_{\text{vmem}} \). \( \square \)
C.2. Prior volatile memory proofs

We now summarize how prior AIPs for memory can be created by replacing different sub-proofs in our protocols with alternatives. First, we discuss alternative permutation proofs. Then, we discuss alternative range proofs, which are used to show the ordering of tr’.

C.2.1. Permutation. TinyRAM [45] gives a different volatile memory proof, in part by using a different permutation proof that does not require V randomness. It is based on a routing network [96]. In it, the rows of A and B are connected by a network of crossbar switches which can be programmed to implement any permutation. P sends the switch settings. Then, V checks the permutation. TinyRAM specifically uses the Waksman network [56], which has size $\Theta_k(n \log n)$, engendering V cost $\Theta(k n \log n)$.

C.2.2. Ordering and ranges. We show that the permuted transcript tr’ is grouped by a (and then ordered by t). Prior works show that it is ordered by (a, t). To show the micro-scale ordering by t, it suffices to use $\Pi_{\text{ord}}$ just as we do. To show the macro-scale ordering by a, prior works show that the sequence is monotonic. That is, that $a_1, \ldots, a_A \in [N]$ is in the language:

$$L_{\text{mon}} = \{ a : \forall i \in [A - 1], \ a_{i+1} \geq a_i \}$$

Since all the $a_i$ are known to be in range $[N]$, and since $N < \text{char}(F)$, this reduces to arguing that the differences $d_i = a_{i+1} - a_i \in \{0, \ldots, N\} \subset F$. That is, to give range proof:

$$L_{\text{range}} = \{ (d \in \mathbb{F}^A, N) : \forall i \in [A], \ 0 \leq d_i \leq N \}$$

Alternatively, one can give a range proof for the interval $0 \leq d_i < n$, where $n$ is any integer satisfying $N n < \text{char}(\mathbb{F})$ and $k > n$. By setting $n$ to be the next power of two, Ben-Sasson et al. give a range-proof $\Pi_{\text{range}, \text{bit}}$ that uses bit-splitting and has V complexity $\Theta(A \log N)$ [45]. This is the approached used in TinyRAM.

Another line of works give range proofs based on lookup tables. Bootle et al. give a range-proof $\Pi_{\text{range}, \text{ary}}$ that implements a lookup table using a permutation proof. The total V cost is $3(N + A) + O(1)$ [32]. This is the approached used in Arya. This approach is great if $N$ is small, but if $N \gg A$ (e.g., as when emulating a machine with a 64-bit address space) then it is abysmal.

Haböck gives a proof $\Pi_{\text{range}, \log}$ that improve the constants of $\Pi_{\text{range}, \text{ary}}$ (in 1-R1CS) to $N + A + O(1)$ [34], by using logarithmic derivatives. This is the approached used in the “Haböck” volatile memory proof that we benchmark. It is also the approached used in our implementation of $\Pi_{\text{ord}}$.

To mitigate the large-$N$ costs, one can hybridize lookup-tables and bit-decomposition. In this approach, P shows that $d_i$ can be represented as $\log_2(N)$ digits in the range $\{0, \ldots, k - 1\}$, and range-checks the digits using lookups. The cost is $\Theta(k + A \log_2 N)$ (from here on, we ignore rounding). As we will see, by setting $k$ appropriately, one can get good asymptotic performance, even for large $N$. Our experiments (Section 9.1) confirm that concrete performance is also good. In the rest of this paper, we assume use the names $\Pi_{\text{range}, \text{ary}}$ and $\Pi_{\text{range}, \log}$ to refer to the hybridized versions of the protocols, with $k$ set optimally.

As applied to tables of consecutive integers, this hybridization approach has been explored by recent works [33, 35, 36, 39], and generalized to tables with “Spark-only structure” [41] by Setty et al. There are applications to optimized zkSNARKs of virtual machine execution [97].

Theorem 10. With $k$ set optimally, $\Pi_{\text{range}, \text{ary}}$ and $\Pi_{\text{range}, \log}$ have V complexity that is both $\Theta_A(\log N/(\log \log N))$ and $\Theta_N(A)$.

Proof. First, we characterize the optimal k. Then, we compute the V complexity. We will reach a point where it is difficult to proceed with a simultaneous analysis in large $N$ or large $A$, so we will settle for two different asymptotic bounds.

To find the optimal $k$, we annihilate the derivative:

$$0 = 1 - A(\ln N)/k - 2k^{-1}$$

$$k \ln^2 k = A \ln N$$

$$k = e^{2W_0(1/2)\sqrt{A \ln N}}$$

where $W_0(x)$ is the principal branch of the Lambert W function. So, our cost is:

$$e^{2W_0(1/2)\sqrt{A \ln N}} + \frac{A \ln N}{2W_0(1/2)\sqrt{A \ln N}}$$

This is a somewhat challenging expression to analyze asymptotically in two variables, as $W_0$’s argument depends on both. And, while we have fairly simple asymptotic bounds on $W_0$ [98], its precise behavior is more complex. So, we settle for asymptotic bounds that depend on $N$ or $A$ but not both.

We begin with the fixed A case. The argument x to $W_0(x)$ is large, so we can apply a result of Hoofar and Hasseni: that $W_0(x) = \ln x - \ln \ln x + \Theta((\ln \ln x)/\ln x)$ [98]. Since $k$ is set optimally, it suffices to analyze either summand in the cost expression. We choose the second, and substitute the asymptotics for $W_0$:

$$\Theta_A \left( \frac{\ln N}{2 \ln(\sqrt{\ln N}) + o(\ln(\sqrt{\ln N}))} \right) = \Theta_A \left( \frac{\ln N}{\ln \ln N} \right)$$

Now, we consider the other case: when $N$ is fixed. It’s easy to see that cost is $\Theta_N(A)$ in this case.

C.3. Persistent memory

Protocol 12 is our persistent memory proof. It is very similar to our volatile memory proof, so we only explain the differences. First, both parties initially associate each access in tr with a time $t_i \leftarrow i$ and a flag $g_i \leftarrow 0$. Second, $P$ produces a vector $a$ of distinct addresses that include all those touched by tr, as well as the initial and final values at those addresses $v$ and $v’$. Third, both parties augment the transcript with initialization entries $(v_i, a_i, g_i = 1, w_i =$
\(\Pi_{\text{perm}}(m, m' \in \mathbb{F}^N; tr)\)

\[\begin{align*}
\mathcal{P}(\cdots) & \Rightarrow \mathcal{V}(\cdots) \\
\text{include } t_i, g_i, v_i \in 0 \text{ in } tr & \quad \forall i \in [t] \\
a \leftarrow A \text{ address covering } tr & \\
v \leftarrow (m_a)_{a = 1}^{A} \\
v' \leftarrow (m'_{a})_{a = 1}^{A} \\
(v', v', a) & \quad \leftarrow \mathcal{V}_{\text{perm}}((v, v, a)) \\
\text{for all } i \in [t] & \quad \text{(v', a', g', w', t')} \leftarrow \mathcal{V}'((v, a, g, w, t)) \\
\text{and for all } i \in [t] & \quad \Pi_{\text{active}}(m, m', v, v', a) \\
\text{we get } t' & \leftarrow \text{sort}(a, t)(t') \\
\text{for all } i \in [t] & \quad \Pi_{\text{active}}(m, m', v, v', a) \\
\text{for all } i \in [t] & \quad \Pi_{\text{perm}}((v, v, a)) \\
\text{for all } i \in [t] & \quad \Pi_{\text{collect}}(t', 1 - g', 0, A + 1) \\
\text{for all } i \in [t] & \quad \Pi_{\text{unique}}(a', g') \\
\text{for all } i \in [t] & \quad \Pi_{\text{attrib}}(v', v', a). \\
\end{align*}\]

Protocol 12: Our persistent memory proof.

1, \(t_i = 0\) and with finalization entries \((v'_i, a, g_i = 0, w_i = 0, t_i = A + 1)\). Fourth, \(\mathcal{P}\) sends the sorted transcript \(t'\) to \(\mathcal{V}\). Then, the parties engage in sub-protocol \(\Pi_{\text{active}}\), to show that the active memory cells that are represented by \((v, v, a)\) are consistent with \((m, m')\). Finally, the parties use the same sub-protocols as in \(\Pi_{\text{perm}}\) to show that \(t'\) (by extension tr) respect read-over-write.

**Theorem 11.** The protocol \(\Pi_{\text{perm}}(m, m' \in \mathbb{F}^N; tr)\) has perfect completeness and soundness error \(\frac{N^2 + 6N + A}{|\mathbb{F}|} + \frac{N}{|\mathbb{F}| - N}\).

**Proof.** We address completeness and then soundness.

**Completeness.** First, we argue that \(\Pi_{\text{active}}\) succeeds. Per the construction of \(v\) and \(v'\), for all \(i \in [A]\), \(v_{a_i} = m_{i}\) and \(v'_{a_i} = m'_{i}\). Further, since all \(i \notin a\) do not appear in \(tr\), \(m_i = m'_i\).

The addresses \(a\) are unique by construction. Thus, \((m, m', v, v', a) \in \mathcal{L}_{\text{active}}, \) so \(\Pi_{\text{active}}\) (which is perfectly complete) succeeds.

Now, it suffices to show that \(t'\in \mathcal{L}_{\text{perm}}\) (excluding the \(g\) flags), and that \(t'\) in \(\Pi_{\text{perm}}\) has \(g\) flags that have the same values as those in \(\Pi_{\text{perm}}\). If these conditions are met, then the completeness of \(\Pi_{\text{perm}}\) implies that of \(\Pi_{\text{perm}}\). First, observe that after sorting by \((a, t)\), the first access to each address has form \((v_i, b_i, g = 1, w = 1, t = 0)\) in \(tr\), since these addresses have minimal \(t\) among all the addresses in \(tr\). Second, observe that every other access in \(tr\) is a standard address or a finalization address, so it has \(g = 0\). Thus our second condition is met. The first condition follows from the fact that valid\((m, m', tr)\) holds.

**Soundness.** Suppose that valid\((m, m', tr)\) does not hold. Then, for some address \(a \in [N]\), one of the following must hold:

1. The first access to \(a\) in \(tr\) is a read whose value disagrees with \(m\). There are two sub-cases:
   a) \(a\) is equal to some \(a_i \in a\) with \(v_i = m'_{a_i}\), or
   b) not.
2. There are accesses to \(a\) indexed by \(i < j\) with no intermediate access to \(a\) such that \(w_j = 0\) and \(v_i \neq v_j\).
3. The last access to \(a\) in \(tr\) has a value that disagrees with \(m'\). There are two sub-cases:
   a) \(a\) is equal to some \(a_i \in a\) with \(v_i' = m'_{a_i}\), or
   b) not.

If any of cases 1a, 2, or 3a hold, then the soundness of \(\mathcal{V}_{\text{perm}}\) implies that \(\mathcal{V}\) rejects except with probability

\[\leq \frac{4N}{|\mathbb{F}|} + \frac{N}{|\mathbb{F}| - N}\]

This bound is slightly looser than Theorem 9 because \(\Pi_{\text{perm}}\) is applied to 5-tuples here, instead of to 4-tuples as in \(\Pi_{\text{perm}}\).

In case 1b or 3b, we have that \((m, m', v, v', a)\) is not in \(\mathcal{L}_{\text{active}}\). Thus, Theorem 5 implies that \(\mathcal{V}\) rejects except with probability \(\leq (N^2 + 2N + A)/|\mathbb{F}|\).

Thus, \(\mathcal{V}\) rejects except with probability at most

\[\frac{N^2 + 6N + A}{|\mathbb{F}|} + \frac{N}{|\mathbb{F}| - N}\]

\(\square\)

**C.4. Sparse persistent memory**

Protocol 13 is our sparse persistent memory proof, building on Protocol 14 for extracting the active cells from the memory. It is very similar to Protocol 12, so we explain only the differences. First, the vectors \(a, a'\) now denote part of the persistent storage, while \(b'\) denotes the “active” addresses. Second, the active addresses \(b'\) are computed slightly differently: \(b'\) now must contain elements that are a sub(multi)set of those in \(a'\), including all the addresses that appear in \(tr\).

Protocol 13 continues by building an augmented transcript \(\hat{tr}\) that contains initialization and finalization entries for each address, as in \(\Pi_{\text{perm}}\). It then runs the same collection of sub-protocols that \(\Pi_{\text{perm}}\) uses, save that it runs \(\Pi_{\text{s-active}}\) in place of \(\Pi_{\text{active}}\).

Checking the consistency of the \((m, m', a, a', v, v', b')\) tuple is deferred to the sub-protocol \(\Pi_{\text{s-active}}\) which was explained at a high level in Section 6. It is detailed as Protocol 14. At a high level, it shows that (a) each \((b'_i, v'_i)\) is equal to some \((a'_i, m'_{a'_i})\), (b) each \((b'_i, v_i)\) is equal to some \((a_i, m_i)\) or \(a_i\) is a dummy, and (c) the \(a'_i\) are all unique or dummies, assuming that the \(a_i\) are. More precisely, it proves membership in the following language:

\[\mathcal{L}_{\text{s-active}} = \{(m, m', a, a' \in \mathbb{F}^N; v, v', b' \in \mathbb{F}^A) : \exists I \subseteq [C], |I| = C - A, \exists b \in \mathbb{F}^A, \]

\[\bigwedge_{i=1}^{A} (b_i = 0 \lor b_i = b'_{i}) \land \]

\[\{(a_i, m_i)\}_{i=1}^{C} \subseteq \{(a_i, m_i)\}_{i=1}^{C} \cup \{(b_i, v_i)\}_{i=1}^{A} \land \]

\[\{(a'_i, m'_{a'_i})\}_{i=1}^{C} \subseteq \{(a'_i, m'_{a'_i})\}_{i=1}^{C} \cup \{(b'_i, v'_i)\}_{i=1}^{A} \land \]

\[\{a_i\}_{i=1}^{C} \cap \{b'_i - b_i\}_{i=1}^{A} \subseteq \{0\} \} \}


one can show that

\[ b' \leftrightarrow A \text{ addr } \text{covering } \text{tr} \]

\[ v \leftarrow (m_i')^A \]

\[ v', v', b' \]

\[ \text{for each active cell } i \]

\[ \text{the protocol } \text{P} \]

\[ \text{Theorem 12. The protocol } \Pi_{\text{s-active}} \text{ (for language } L_{\text{s-active}}) \text{ has perfect completeness and soundness error } (C^2 + 3C + 2A - 1)/|F| \]

\[ \text{Proof. We address completeness and then soundness.} \]

\[ \text{Completeness. Define } I \text{ to be } [C] \text{ with the items of } j \text{ removed. Per membership in } L_{\text{s-active}}, I \text{ and b exist. Moreover, one can show that b in this proof is equal to b in the protocol. The last condition of membership immediately implies that } \Pi_{\text{PuncDisj}} \text{ succeeds. Also, the first condition of membership (} b_i = 0 \) \]

\[ \text{for all } i \in [A] \text{ implies that } \mathcal{V}' \text{ first test succeeds. Let } J \text{ be the set of } j \text{ s contents, and } \]

\[ \Pi' = [C] \backslash J. \]

\[ \text{Consider the multisets} \]

\[ \{m_i + a_i \beta \}_{i \in J} = \{m_i + a_i \beta \}_{i \in I} \cup \{m_i + a_i \beta \}_{i \in J} \]

\[ \text{So, we have} \]

\[ \prod_{i \in I} (\alpha + m_i + a_i \beta) = \prod_{i \in J} (\alpha + m_i + a_i \beta) \times \prod_{i \in J} (\alpha + m_i + a_i \beta) \]

\[ \text{The factors in the second product are exactly the } \alpha + h_i, \text{ as computed by } \mathcal{P}, \text{ so we have} \]

\[ \prod_{i \in J} (\alpha + m_i + a_i \beta) = c \times \prod_{i \in J} (\alpha + m_i + a_i \beta) \]

\[ \text{Then, by the definition of the } b \text{ and } b', \text{ we have} \]

\[ \prod_{i \in J} (\alpha + m_i + a_i \beta) = c \times \prod_{i \in J} (\alpha + m_i + a_i \beta) \]

\[ \text{and } \mathcal{V}' \text{ s second test succeeds. And, a similar argument shows that } \mathcal{V}' \text{ s third test succeeds.} \]

\[ \text{Soundness. Suppose the } \mathcal{V} \text{ accepts. We will argue that the instance is in the language. Because of } \mathcal{V}' \text{ s last tree tests, } \beta \text{ is independent of } m, \text{m}', \text{a, a}', \text{v}, \text{v}', \text{b}, \text{b}', \text{and } \alpha \text{ is independent of all those and } h \text{ too, we have two multiset equations:} \]

\[ \{(a_i, m_i)\}_{i \in J} = \{(a_i, m_i)\}_{i \in I} \cup \{(b_i, v_i)\}_{i \in [A]} \]

\[ \{(a'_i, m'_i)\}_{i \in J} = \{(a_i, m_i)\}_{i \in I} \cup \{(b'_i, v'_i)\}_{i \in [A]} \]

\[ \text{for some } I \subseteq [C], \text{ except with probability } (C^2 + 2C + A)/|F| \]

\[ \text{by the same argument as in the proof of Theorem 5, with } C \text{ in place of } N. \text{ This satisfies the middle two conditions of } L_{\text{s-active}}. \text{ The first condition is ensured by } \mathcal{V} \text{'s first test, and the last condition is ensured by } \Pi_{\text{PuncDisj}}, \text{ except with probability (Thm. 7): } (C + A - 1)/|F| \]

\[ \text{Thus, the overall soundness error is} \]

\[ \frac{C^2 + 3C + 2A - 1}{|F|} \]

\[ \square \]

\[ \text{Theorem 13. The protocol } \Pi_{\text{Pmem}} \text{ (for language } L_{\text{Pmem}}) \text{ has perfect completeness and soundness error } O((C^2 + A)/|F|) \]

\[ \text{Proof. The proof of this theorem is similar to the proof of Theorem 11.} \]

\[ \square \]

\[ \text{Appendix D.} \]

\[ \text{Computing } \text{Bézout polynomials} \]

\[ \text{Figure 2 gives our algorithm } \text{UniqBez}. \text{ Like } \text{SimpleUniqBez}, \text{ its runtime is } O(M(L) \log L) \text{ and its } \text{is easy to parallelize. In addition, we implement one optimization not shown. In the first two loops, we} \]
UniqBez\( (x_1, \ldots, x_{2^r}) \)

\[
\text{for } i \in [2^r]: \\
p_{i,0} \leftarrow X - x_i \\
\text{for } j \in \{0, \ldots, 1\}, i \in [2^{l-1}]: \\
p_{j-1,i} \leftarrow p_{j,2i} \cdot p_{j,2i+1} \\
q_{0,0} \leftarrow p_{0,0} \\
\text{for } j \in [l], i \in [2^{l-1}]: \\
q_{j,2i} \leftarrow q_{j-1,i} \text{ mod } p_{j,2i} \\
q_{j,2i+1} \leftarrow q_{j-1,i} \text{ mod } p_{j,2i+1} \\
\text{for } i \in [2^r]: t_{i,i} \leftarrow (q_{i,i})^{-2} \\
\text{for } j \in \{0, \ldots, 1\}, i \in [2^{l-1}]: \\
p_{j-1,i} \leftarrow t_{j,2i} \cdot p_{j,2i+1} + t_{j,2i+1} \cdot p_{j,2i} \\
\text{return } s \leftarrow (1 - p_{0,0} \cdot t_{0,0})/p_{0,0}, t \leftarrow t_{0,0}
\]

Algorithm 2: Our algorithm for computing Bézout polynomials \( s, t \) for \( z(X) = \prod_{i=1}^{2^r} (X - x_i) \) and its derivative.

precompute all \( \text{rev}(p_{j,i}) \text{ mod } X^{2^{l-j}} \) \cite[Ex. 10.9]{75}, where \( \text{rev} \) reverses the coefficients of a polynomial. These values can be used to accelerate the modular reductions in the third loop.

Now, we restate and prove Theorem 4.

**Theorem.** Given \( a_1, \ldots, a_A \in \mathbb{F} \), UniqBez outputs \( s, t \in \mathbb{F}[X] \) satisfying \( sz + tz' = 1 \) (where \( z = \prod_{i}(X - a_i) \)) in time at most \( 4.5M(A) \log A + O(A \log A) \).

**Proof.** Let \( z(X) = \prod_{i}(X - x_i) \). First, we prove UniqBez returns \( s, t \) satisfying \( sz + tz' = 1 \). First, observe that \( p_{0,0} = z \). Next, observe that \( q_{i,i} \) is a scalar and is \( z'(x_i) \). Next, observe that for each \( x_i, 1 = z(x_i)s(x_i) + z'(x_i)t(x_i) = z'(x_i)t(x_i) \). Thus, \( t(x_i) = (z'(x_i))^{-1} \).

Now, observe that \( t(X) = \sum_{i=1}^{L}(t(x_i))^2 \prod_{j \neq i}(X - x_i) \).

This is because \( t(x_i) \prod_{j \neq i}(X - x_i) \) is the 1.

Now, we analyze the runtime of UniqBez. As is the convention in computer algebra, we quantify runtime as the number of scalar operations, and we use the function \( M(L) \) to denote the (asymptotic) cost of multiplying two polynomials of degree \( < L \); this is \( L \log L \). We will see that UniqBez has runtime \( 4.5M(L) \log L + O(L \log L) \). To see this, observe that UniqBez is very similar to the fast interpolation algorithm given by Gathen and Gerhard \cite[Alg. 10.11]{75}. The only difference is that UniqBez uses the \( (q_{i,i})^{-1} \) as both the Lagrange coefficients and the target evaluations, instead of having a separate list \( v \in \mathbb{R}^L \) of target evaluations. Thus, the runtime of UniqBez is the same as the runtime of interpolation.\(^8\) And that runtime is \( 4.5M(L) \log L + O(L \log L) \) \cite[Ex. 10.9]{75}. \( \square \)

---

\(^8\)UniqBez is very slightly faster, because it replaces \( O(n) \) multiplications between Lagrange coefficients and the \( v_i \) with \( O(n) \) squarings—but this is a low-order difference.