

A Black-Box Compiler from Static to Dynamic SNARKs

Anubhav Baweja
abaweja@upenn.edu
University of Pennsylvania

Charalampos Papamanthou
charalampos.papamanthou@yale.edu
Yale University

October 11, 2026

Abstract

Succinct Non-Interactive Arguments of Knowledge (SNARKs) for NP allow a prover to prove knowledge of a witness of an NP relation to a verifier with a short proof. The primary goal of a (static) SNARK is to make verification cheap: the proof should be short, and the verifier’s work should be minimal. However, recent work has also focused on making SNARKs more *prover-efficient*: cheap verification should ideally not come at the cost of large prover overhead. One avenue for achieving prover efficiency is via *Dynamic SNARKs*, which allow a prover to efficiently update the proof when the witness is updated. For example, one may use a SNARK to issue a proof of a query being performed correctly on a database, and then issue a new proof of the query once the database is updated. If these updates are small compared to the size of the witness (only a single entry of a large database is updated), then the prover should ideally be able to use the old proof to generate a new proof in time proportional to the distance between the old and the new witness.

While some constructions of dynamic SNARKs for NP have been recently proposed by Wang et al. [Eurocrypt 2026] and Zhang et al. [Eurocrypt 2026], they have many limitations, such as lack of post-quantum security, dependence on a setup by a trusted third-party, and poor asymptotic and concrete efficiency. In this work, we present a compiler for constructing a dynamic SNARK for Plonkish constraints (which capture arbitrary NP computations) by using a static SNARK for NP and a vector commitment scheme as black boxes. When instantiated with a state-of-the-art hash-based SNARK and Merkle trees, the compiler yields a dynamic SNARK that is post-quantum secure in the QROM, has a transparent setup, and achieves $O(n)$ initial prover time and $O(k \log^2 n)$ update prover time, with succinct proofs and verification, where k is the Hamming distance between the old and new instance-witness pairs. Our Ligerito-based implementation achieves cumulative prover speedups of $147\times$ and $11.7\times$ over a static HyperPlonk baseline for 10,000 successive one-gate updates and 1,000 successive 128-gate updates, respectively, on a 2^{23} -gate circuit, including the full dynamic initialization cost.

Contents

1	Introduction	3
1.1	Our contributions	3
1.2	Related work	4
2	Technical overview	7
3	Preliminaries	14
3.1	Cryptographic primitives	14
3.2	Merkle commitments	19
4	Dynamic SNARK compiler	22
4.1	Setup and indexing	24
4.2	Dynamic SNARK construction overview	25
4.3	Full construction	27
4.4	Proof sketch of Theorem 4.1	31
4.5	Instantiation with Merkle commitments	31
5	Concrete instantiation of Dynamic SNARKs	34
5.1	Preliminaries for PIOP	34
5.2	PIOPs for initial and update statements	37
5.3	PIOPs for Poseidon and Merkle relations	39
5.4	Implementation and evaluation	44
	AI use disclosure	48
	References	48
A	Proof of the dynamic SNARK compiler theorem	53
A.1	Amortized efficiency	53
A.2	Knowledge soundness	53
B	Deamortization of update time	56
C	Zero-knowledge compiler	57

1 Introduction

A succinct non-interactive argument of knowledge (SNARK) [Kil92; Mic00] is a cryptographic primitive that allows a prover to convince a verifier of the truth of a statement, with a short proof. Recent years have seen a surge of interest in SNARKs due to their applications in verifiable computation [PGHR13; BCG+13], private payments and smart contracts [BCG+14; KMS+16], zero-knowledge bridges and rollups [But21; XZC+22; LXZ+24], anonymous credentials [RWGM23; FS24], verifiable machine learning [LXZ21; CWSK24; APKP24; SLZ24], and more.

SNARKs make large computations cheap to verify, but not necessarily cheap to prove. This distinction is especially costly in stateful applications, where a prover repeatedly needs to prove computations whose witnesses differ in only a few locations. For example, a service may repeatedly need to prove that a private database satisfies a fixed set of consistency constraints, even though only a handful of entries change between versions. SNARK provers are typically *static*, meaning that a small change to an instance/witness requires recomputing the proof from scratch. In the above example, this means that the prover has to do work proportional to the size of the entire database, even if only a few entries have changed.

Dynamic SNARKs. Building on earlier work on proof updates for structured computations (see Section 1.2), Wang et al. [WPSP26b] initiated the general study of *dynamic* SNARKs, where the SNARK prover can efficiently update a proof when the instance/witness only changes in a few locations. They propose constructions based on the KZG [KZG10] polynomial commitment scheme (PCS), and show how dynamic SNARKs can be used to build SNARKs for sparse relations, as well as bounded incrementally-verifiable-computation (BIVC) without recursion. Recent work by Zhang et al. [ZOZ26] specifically targets the Groth16 SNARK [Gro16], and shows how to make it dynamic with little overhead. However, these constructions do not possess several desirable properties:

- **Post-quantum security:** Both works [WPSP26b; ZOZ26] rely on pairing-based assumptions, and are therefore not post-quantum secure.
- **Transparent and universal setup:** Both works require a trusted third-party for the setup. Dynark [ZOZ26] further inherits Groth16’s requirement for a circuit-specific setup.
- **Efficiency:** If n is the size of the input instance-witness tuple and k is the number of changed values in the tuple, Dynaverse [WPSP26b] and Dynark [ZOZ26] both achieve $O_\lambda(k\sqrt{n \log n})$ time to update a proof. This square-root overhead is far from optimal, and is especially costly for large statements with small updates. Dynalog [WPSP26b] achieves $O_\lambda(k \log^3 n)$ update time, but has a large concrete overhead in terms of setup time, prover key size, and update time. Dynalog also invokes generalized inner product arguments (GIPAs) [BCC+16; BMM+21], which adds further concrete overhead.
- **Generality and flexibility:** Dynaverse and Dynalog [WPSP26b] are based on KZG commitments, and Dynark [ZOZ26] is tailored to Groth16. Both works specifically require homomorphic commitments and pairing-based assumptions, and do not provide an avenue for building dynamic SNARKs from other assumptions.

1.1 Our contributions

In this work, we aim to achieve all of these properties simultaneously. To this end, we construct a general compiler that takes as input a static SNARK, and outputs a dynamic SNARK.

Theorem 1.1 (Corollary 4.2, informal). *Let $\mathcal{R}_{\text{Plonk}}$ be the NP-complete Plonkish circuit-satisfiability relation (see Definition 3.1). Suppose instance-witness tuples $(\mathbf{x}, \mathbf{w})$ of $\mathcal{R}_{\text{Plonk}}$ have size at most n , including instances of size $n_0 \leq n$. Assume there exists a static SNARK for NP with prover time $O_\lambda(n \cdot P(n))$, verifier time $O_\lambda(n_0 + V(n))$, proof size $L(n)$, and negligible knowledge soundness error. Then, assuming the existence of collision-resistant hash functions, there exists a dynamic SNARK for the original relation with initial prover time $O_\lambda(n \cdot P(n))$, update prover time $O_\lambda(k \cdot P(n) \log^2 n)$, verifier time $O_\lambda(n_0 + V(n) \log n)$, and proof size $O(L(n) \log n)$ with negligible knowledge soundness error. Here k is the Hamming distance between the old and new instance-witness tuples of $\mathcal{R}_{\text{Plonk}}$.¹*

The above theorem can be instantiated with any static SNARK satisfying the stated properties. We discuss how to instantiate our compiler with a hash-based SNARK.

Hash-based dynamic SNARK instantiation. We instantiate the static SNARK using the polynomial interactive oracle proof (polynomial IOP) of HyperPlonk [CBBZ23], and the interactive oracle PCS of TensorSwitch [BFRW26], and compile the resulting IOP into a SNARK using the BCS transformation [BCS16]. The resulting SNARK is secure in the random oracle model (ROM) [BR93; BCS16], and in the quantum random oracle model (QROM) [BDF+11; CDHZ26]. We state its efficiency and extraction guarantees in Lemma 4.3. Using this static SNARK as the input of our compiler gives the following corollary.

Corollary 1.2 (Corollary 4.4, informal). *Assuming existence of collision-resistant hash functions against quantum adversaries, there exists a dynamic SNARK for NP secure in the QROM with $O_\lambda(n)$ initial prover time, $O_\lambda(k \log^2 n)$ update prover time, $O_\lambda(n_0 + \log^2 n)$ verifier time, and $O_\lambda(\log^2 n)$ proof size, where λ is the security parameter, n is the size of the Plonkish instance-witness tuple, $n_0 \leq n$ is the size of the instance, and k is the Hamming distance between the old and new Plonkish instance-witness tuples.*

In Section 4.5, we discuss other corollaries of our compiler, including an alternative to Dynark [ZOZ26] when using Groth16’s SNARK as the input of our compiler, and an alternative to Dynaverse/Dynalog [WPSP26b] when using a multilinear KZG polynomial commitment scheme [KZG10; PST13]. We compare our instantiations with prior dynamic SNARKs in Table 1.

Implementation and evaluation. In Section 5, we implement the dynamic SNARK obtained from our compiler using relation-specific polynomial IOPs and the Ligerito multilinear PCS [NA25] over $\mathbb{F}_{2^{128}}$.² On a 2^{23} -gate circuit, the mean update prover speedups in a 40-update sweep are $20,500\times$ for one gate and $41.8\times$ for 128 gates, excluding initialization. With the full dynamic initialization included, 10,000 successive one-gate updates and 1,000 successive 128-gate updates achieve cumulative speedups of $147\times$ and $11.7\times$, respectively, against the static HyperPlonk baseline.

1.2 Related work

Prior to the general formulation of dynamic SNARKs, Papamanthou et al. [PSG+24] used the term *updatable SNARK* for a SNARK equipped with an algorithm that transforms a proof for an instance-witness pair $(\mathbf{x}, \mathbf{w})$ into a proof for an updated pair $(\mathbf{x}', \mathbf{w}')$, without rerunning the prover from scratch. They construct Reckle Trees, which support logarithmic-time updates to succinct

¹We measure time in terms word-RAM model operations with word size $\Theta(\lambda + \log n)$, and proof size in bits. Throughout this section, $O_\lambda(\cdot)$ suppresses $\text{poly}(\lambda)$ factors, including hash evaluation and hash arithmetization costs.

²We use TensorSwitch for the asymptotic statement and Ligerito for the evaluated implementation. The implementation is available at <https://github.com/randomshuffler/dynamic-snarks-system>.

	Dynaverse [WPSP26b]	Dynalog [WPSP26b]	Dynark (fully dynamic) [ZOZ26]
Initial prover time	$n \log n$	$n \log^2 n$	$n \log n$
Update prover time	$k\sqrt{n \log n}$	$k\rho_k \log^2 n$	$k\sqrt{n \log n}$
Verifier time	$n_0 + 1$	$n_0 + \log^3 n$	$n_0 + 1$
Proof size (bits)	1	$\log^3 n$	1
Post-quantum	No	No	No
Transparent setup	No	No	No

	HyperPlonk + multilinear KZG (Corollary 4.5)	Groth16 (Corollary 4.6)	HyperPlonk + TensorSwitch (Corollary 4.4)
Initial prover time	n	$n \log n$	n
Update prover time	$k\rho_k \log n$	$k\rho_k \log^2 n$	$k\rho_k \log n$
Verifier time	$n_0 + \log^2 n$	$n_0 + \log n$	$n_0 + \log^2 n$
Proof size (bits)	$\log^2 n$	$\log n$	$\log^2 n$
Post-quantum	No	No	Yes (QROM)
Transparent setup	No	No	Yes

Table 1: Comparison of dynamic SNARKs. Here n , n_0 , and k are as above, and $\rho_k := \log(1 + n/k)$. Efficiency entries are $O_\lambda(\cdot)$ bounds, suppressing $\text{poly}(\lambda)$ factors.

batch proofs for Merkle-committed data, and Reckle+ Trees, which extend this approach to certain map-reduce computations. In this work, we consider the Plonkish circuit-satisfiability relation, which can capture arbitrary NP relations, as opposed to specific computations.

Concurrent work: Dynasaurs. In concurrent work, Batista et al. [BMPP26] construct Dynasaurs, universal pairing-based dynamic zkSNARKs for Plonk and bounded-fan-out R1CS-lite. Their main ingredients are a sparse argument for linear relations, based on witness-dependent polynomials and a rational encoding of the constraint matrices, and a compiler from sparse to dynamic linear arguments that preserves zero-knowledge. Their constructions have constant-size proofs, but the fully-dynamic update time still has a square-root dependence on circuit size, similar to Dynaverse [WPSP26b] and Dynark [ZOZ26].

Designated-verifier dynamic SNARKs. Wang et al. [WPSP26a] construct Delphus, a designated-verifier dynamic zk-SNARK with $O_\lambda(k \log n)$ update time for k changed witness values and constant-size proofs. Delphus builds on Groth16 and KZG commitments, using an auxiliary binary tree to maintain a quotient-polynomial contribution in the pairing target group. Verification requires secret state retained from a circuit-specific trusted setup.

Dynamic authenticated data structures. A data structure is said to be *authenticated* if its operations are performed by an untrusted prover, which provides proofs of said operations which can be checked by a verifier [ABC+12; MHKS14]. Dynamic variants support efficient updates to the authenticated state [PTT11; PSTY13], and they have been used to build verifiable systems for dynamic databases, including vSQL [ZGK+17]. However, these works are limited to specific data structures and operations, and do not provide a general framework for building dynamic SNARKs for arbitrary NP relations.

A close prior notion to a dynamic SNARK is a dynamic vector commitment [CF13], which allows a prover to update opening proofs for a vector commitment efficiently when the vector changes in a

few locations. [CF13] describe this property for Merkle commitments among other constructions, which was later generalized by Tas and Boneh [TB23] to give smooth update-time/opening-time tradeoffs.

Incrementally verifiable computation. Incrementally verifiable computation (IVC) [Val08] maintains a succinct proof that a current state is reachable from an initial state by repeatedly applying a prescribed transition relation. The prover’s work at each step is tied to the cost of the transition relation, but need not grow with the length of the execution. A dynamic SNARK addresses a different statement. It proves only that the current instance-witness pair satisfies a fixed relation, without asserting that the pair was reached from a valid initial state through a prescribed sequence of transitions.

IVC, whether instantiated through recursive proof composition [BCCT13; BCTV14a] or accumulation [BGH19; BCMS20; KST22; BC23], potentially offers an alternative path to construct dynamic SNARKs, by expressing witness updates as repeated applications of a small uniform transition relation, as in prior work [BCG+13; BCTV14a; BCTV14b; NDC+24]. However, obtaining IVC by recursively composing a SNARK or an accumulation scheme requires proving statements about its verification algorithm. For ROM-based ingredients, this algorithm may query the random oracle, and security of the ingredients in the ROM does not by itself justify this recursive composition. Our compiler avoids this requirement: its proved relations check local constraints and explicitly computable commitment operations, while verification of the static SNARK proofs occurs outside these relations. Consequently, the compiler can preserve ROM and QROM security without requiring a recursive instantiation of the underlying SNARK.

As part of a general IVC framework in the standard model, Aldema Tshuva and Oshman [AO26] construct updatable batch arguments using a hierarchical data structure. Their data structure resembles ours as discussed in Section 2, but their merges prove acceptance of child proof verifiers, whereas ours directly prove combined update statements.

2 Technical overview

Throughout this section, we fix $\mathbb{F}$ as a finite field of size $\Theta(2^\lambda)$ for a security parameter λ . We also fix an arbitrary NP relation $\mathcal{R}$ which is represented as a fan-in 2 arithmetic circuit $\mathcal{C}$ over $\mathbb{F}$ with n addition gates, n multiplication gates, and $n_0 \leq n$ public input occurrences.³ An instance of $\mathcal{R}$ is a public input $\mathbf{x} \in \mathbb{F}^{n_0}$, and a witness of $\mathcal{R}$ is a satisfying evaluation trace of $\mathcal{C}$.

Plonkish arithmetization. The witness of $\mathcal{R}$ can be arithmetized in many ways, including R1CS constraints [BCG+13], CCS constraints [STW23], and Plonkish constraints [GWC19; CBBZ23]. In this work, we use Plonkish constraints, although our techniques might be useful for constructing dynamic SNARKs for other arithmetizations as well. We represent the evaluation trace of $\mathcal{C}$ as a vector $\mathbf{w} \in \mathbb{F}^{6n}$ and a permutation $\sigma \in S_{6n+n_0}$, such that $(\mathbf{x}, \mathbf{w}) \in \mathcal{R}$ if and only if the following constraints are satisfied:

- **Gate constraints:** For every $i \in [n]_0$ ⁴, $z[6i] + z[6i+1] = z[6i+2]$ and $z[6i+3] \cdot z[6i+4] = z[6i+5]$ where $z := [\mathbf{w}, \mathbf{x}]$.
- **Wiring constraints:** For every $i \in [6n + n_0]_0$, $z[i] = z[\sigma(i)]$.

The gate constraints ensure that the evaluation trace satisfies the addition and multiplication gates of $\mathcal{C}$, while the wiring constraints ensure that the evaluation trace is consistent with the wiring of $\mathcal{C}$, which is captured by the permutation σ . Thus, the prover can prove knowledge of a witness for $\mathcal{R}$ by proving knowledge of a vector $z \in \mathbb{F}^m$ where $m := 6n + n_0$ which satisfies the gate and wiring constraints, and is consistent with the public input $\mathbf{x}$.

Remark 2.1 (Technical overview simplifications). In this overview, for simplicity we assume that σ is public and can be accessed by the verifier in whole, and we also omit the discussion of checking consistency with the public input. Our full construction in Section 4 describes how to avoid the σ assumption and discusses the public-input consistency checks in detail.

Let $\mathbf{P}$ and $\mathbf{V}$ be the prover and verifier of a static SNARK for NP such that, for a relation represented by an arithmetic circuit of size N , $\mathbf{P}$ runs in $O(N \cdot P(N))$ time. Using $\mathbf{P}$ and $\mathbf{V}$, we will attempt to construct a dynamic SNARK for $\mathcal{R}$ with prover $\mathcal{P}$ and verifier $\mathcal{V}$.

First attempt at a dynamic SNARK. First, $\mathcal{P}$ invokes $\mathbf{P}$ to obtain the *initial proof* π_{init} of the following *initial statement*: it knows a witness $\mathbf{w}$ such that $z := [\mathbf{w}, \mathbf{x}]$ satisfies the gate and wiring constraints. $\mathcal{P}$ sends π_{init} to $\mathcal{V}$ which then invokes $\mathbf{V}$ to verify the claim. Now, suppose $\mathcal{P}$ receives an update in the form of a set of indices $U \subseteq [6n + n_0]_0$ and corresponding new values $\{j : \beta_j\}_{j \in U}$ such that $|U| \leq k$. Let $A(U)$ and $M(U)$ denote the sets of addition and multiplication gates that are updated by U , respectively.⁵ Define

$$z'[j] := \begin{cases} \beta_j & \text{if } j \in U, \\ z[j] & \text{otherwise.} \end{cases} \quad (1)$$

Our goal is to output a new proof π' such that $\mathcal{V}$ can verify the new claim that $(\mathbf{x}', \mathbf{w}') \in \mathcal{R}$ where $\mathbf{x}' = z'[6n : 6n + n_0]$ and $\mathbf{w}' = z'[0 : 6n]$ such that $\mathcal{P}$ runs in time $O_\lambda(k \cdot P(n) \cdot \log^2 n)$. The naive

³Any NP relation can be reduced to arithmetic-circuit satisfiability by arithmetizing its polynomial-time verifier.

⁴We use $[n]_0$ to denote the set $\{0, 1, \dots, n-1\}$.

⁵For simplicity, we assume that U is closed under gate operations. That is, if $6i \in U$, then $6i+1 \in U$ and $6i+2 \in U$, and vice-versa, and similarly for multiplication gates.

approach of updating π_{init} need not achieve this goal without additional assumptions on the static SNARK. For example, if we use a hash-based SNARK based on error-correcting codes with constant relative distance, the encoding of $\mathbf{z}'$ will end up differing in $\Theta(n)$ locations from the encoding of $\mathbf{z}$ even if $k = O(1)$. Explicitly updating this encoded oracle requires $\Omega(n)$ writes, even for a single changed value. However, instead of updating π_{init} , maybe $\mathcal{P}$ can send some additional information to $\mathcal{V}$ that allows it to verify the new claim. Specifically, $\mathcal{P}$ wants to prove the following:

- For all $i \in A(U)$: $\beta_{6i} + \beta_{6i+1} = \beta_{6i+2}$, and for all $i \in M(U)$: $\beta_{6i+3} \cdot \beta_{6i+4} = \beta_{6i+5}$.
- For every $j \in U$, $\beta_j = \mathbf{z}'[\sigma(j)]$. Note that $\sigma(j)$ may or may not be in U .

However, it's unclear how to prove the wiring constraint check above. In particular, the wiring constraint check uses values of $\mathbf{z}'$ as defined in (1), and in turn potentially uses values of $\mathbf{z}$ outside of U . If $\sigma(j) \notin U$, then $\mathbf{z}'[\sigma(j)] = \mathbf{z}[\sigma(j)]$, but $\mathcal{V}$ does not know the true value of $\mathbf{z}[\sigma(j)]$. $\mathcal{P}$ can send the purported value of $\mathbf{z}[\sigma(j)]$ to $\mathcal{V}$, but then $\mathcal{V}$ has to verify that the purported value is indeed the true value of $\mathbf{z}[\sigma(j)]$. Fortunately, this can be done using vector commitments.

Using vector commitments to link old and new witnesses. A vector commitment scheme [CF13] provides algorithms VC.Commit , VC.Open , and VC.Verify . The commitment algorithm maps a vector $\mathbf{x} \in \mathbb{F}^N$ to a short digest cm . For a query set $U \subseteq [N]_0$, the opening algorithm produces a short proof authenticating $\mathbf{x}[U]$, which the verifier checks against cm . *Position binding* guarantees that an efficient prover cannot produce accepting openings assigning different values to the same position under the same commitment. For a VC algorithm $\mathcal{A}$, we use $T_{\mathcal{A}}$ for its word-RAM running time and $C_{\mathcal{A}}$ for its arithmetic-circuit size.⁶

In order to prove the wiring constraint check above, we can use a vector commitment VC as follows. Instead of the initial statement proven by $\mathbf{P}$ that $\mathbf{z}$ satisfies the gate and wiring constraints, we augment it by including the claim that $\text{cm} = \text{VC.Commit}(\mathbf{z})$. If $\mathcal{P}$ sends π_{init} and cm to $\mathcal{V}$, it is convinced that the vector “underlying” the commitment cm satisfies the gate and wiring constraints. Thus, the initial statement now has circuit size $N_{\text{init}} = O(n + C_{\text{VC.Commit}}(m))$, and the initial prover time becomes

$$T_{\text{init}} = O(n + N_{\text{init}}P(N_{\text{init}})) \quad .$$

After $\mathcal{P}$ receives the update, it invokes VC.Open on the update index set U to obtain an opening proof π_{vc} . $\mathcal{P}$ sends $(U, \{j : \alpha_j\}_{j \in U}, \{j : \beta_j\}_{j \in U}, \pi_{\text{vc}})$ to $\mathcal{V}$, which performs the following checks:

1. The new gate constraints are satisfied by the new values $\{j : \beta_j\}_{j \in U}$.
2. For all $j \in U$ such that $\sigma(j) \in U$, check that $\beta_j = \beta_{\sigma(j)}$.
3. For all $j \in U$ such that $\sigma(j) \notin U$, check that $\alpha_j = \beta_j$.
4. VC.Verify accepts the proof π_{vc} for the claimed old values $\{j : \alpha_j\}_{j \in U}$ with respect to the commitment cm .
5. $\mathbf{V}$ accepts the initial proof π_{init} .

Check 1 above explicitly checks the gate constraints within U . For the wiring constraints, consider an arbitrary cycle C of the permutation σ , where there are three types of edges:

⁶We omit the security parameter λ from these arguments and use O_{λ} as in the introduction when specializing to Merkle trees later in this section.

1. Edges within U : checked explicitly by check 2 above.
2. Edges outside U : if checks 4 and 5 above pass, $\mathcal{V}$ is convinced that the claimed old values $\{j : \alpha_j\}_{j \in U}$ are indeed the values of a vector $\mathbf{z}$ that satisfies all gate and wiring constraints. Thus, the wiring constraints for edges outside of U are satisfied for the new vector.
3. Edges crossing U : consider $j \in U$ such that $\sigma(j) \notin U$. Check 3 ensures that the values are unchanged at the “edges” of the update index set U . Since we have already certified that $\mathbf{z}[j] = \mathbf{z}[\sigma(j)]$ and $\mathbf{z}[j] = \alpha_j$ and $\mathbf{z}[\sigma(j)] = \mathbf{z}'[\sigma(j)]$ by the other checks, we get $\alpha_j = \mathbf{z}'[\sigma(j)]$. If check 3 passes, then we have $\alpha_j = \beta_j$ so we get $\beta_j = \mathbf{z}'[\sigma(j)]$ as desired. For edges in the opposite direction such that $j \notin U$ but $\sigma(j) \in U$, a similar argument applies.

While this approach is sound and the update prover time is $O(k + T_{\text{VC.Open}}(m, k))$, it has a large proof size of $\Omega(k)$. This is particularly undesirable if $k = \Omega(n^\gamma)$ for some constant $\gamma > 0$. However, instead of sending all of this information to $\mathcal{V}$, we can use a SNARK to prove that there exist values that satisfy all of the checks above. Specifically, we can define a *transition statement* with instance cm and witness $(U, \{j : \alpha_j\}_{j \in U}, \{j : \beta_j\}_{j \in U}, \pi_{\text{vc}})$, that proves that the first four checks described above pass. The circuit size $N_{\text{tr}}(k)$ of this transition statement is

$$N_{\text{tr}}(k) = O(k \log n + C_{\text{VC.Verify}}(m, k)) ,$$

where $O(k \log n)$ is the circuit size for performing the first 3 checks, and $C_{\text{VC.Verify}}(m, k)$ is the circuit size for performing the VC.Verify operation of check 4.⁷ Using $\mathbf{P}$, we generate a *transition proof* π_{tr} in $O(N_{\text{tr}}(k)P(N_{\text{tr}}(k)))$ time. Including the opening and update-witness construction, the prover’s total work at this stage is

$$O_\lambda(T_{\text{VC.Open}}(m, k) + N_{\text{tr}}(k)P(N_{\text{tr}}(k))) .$$

Now, $\mathcal{P}$ only needs to send the initial proof π_{init} , the transition proof π_{tr} , and the commitment cm to $\mathcal{V}$, and $\mathcal{V}$ can verify the new claim efficiently by invoking $\mathbf{V}$ on both proofs, as long as VC.Open and VC.Verify are efficient.

Semi-dynamic versus Fully-dynamic SNARKs. However, there is still one major issue with the above approach: what happens if there is another update? We were able to handle the first update because $\mathcal{P}$ sent cm to $\mathcal{V}$, and proved that the vector $\mathbf{z}$ underlying cm satisfies the gate and wiring constraints. Thus, we can efficiently handle as many distinct updates to the original vector $\mathbf{z}$ as we want, but we cannot yet handle updates to the updated vector $\mathbf{z}'$, because $\mathcal{V}$ does not possess a proven commitment to $\mathbf{z}'$. We have managed to construct a *semi-dynamic* SNARK as defined by [ZOZ26] with little overhead (barring the simplifications mentioned in Remark 2.1), but our goal is to construct a *fully-dynamic* SNARK that can handle an arbitrary sequence of updates. If we want to handle an update to $\mathbf{z}'$, then $\mathcal{P}$ needs to compute a commitment cm' to $\mathbf{z}'$ and prove that it is indeed the commitment to $\mathbf{z}'$. This is where we need to ask more from our vector commitment scheme VC , and see if there are any properties other than position binding that we can exploit. Fortunately, Merkle trees [Mer87] have many interesting properties that will be useful.

Achieving fully-dynamic SNARKs from Merkle trees. Merkle trees [Mer87] are a type of vector commitments whose position binding property is based on the collision resistance of a

⁷In the full construction, the permutation values at U are authenticated by a Merkle opening of the committed permutation, and wiring equalities are checked using an additional Merkle opening at $\sigma(U)$. This adds another 2 Merkle-verification circuits to the overall circuit, preserving the displayed asymptotic bound.

hash function H . Given a vector $\mathbf{x} \in \mathbb{F}^N$, the Merkle commitment cm is the hash at the root of a binary tree whose i -th leaf contains $\mathbf{x}[i]$ and $H(\mathbf{x}[i])$. Every internal node contains the hash of the concatenation of the hashes at its two children. A canonical opening proof for an index set U consists of the hashes at the roots of the maximal subtrees whose leaves are disjoint from U , as illustrated in Fig. 1. The verifier can recompute the root of the Merkle tree from the opening proof and the values of $\mathbf{x}$ at U , and check that it matches cm .

We discuss the efficiency properties of Merkle trees. We have $T_{\text{MT.Commit}}(N) = O_\lambda(N)$, $T_{\text{MT.Open}}(N, k) = O_\lambda(k \log N)$, and $T_{\text{MT.Verify}}(N, k) = O_\lambda(k \log N)$ where $k = |U|$. The corresponding circuit sizes are the same.

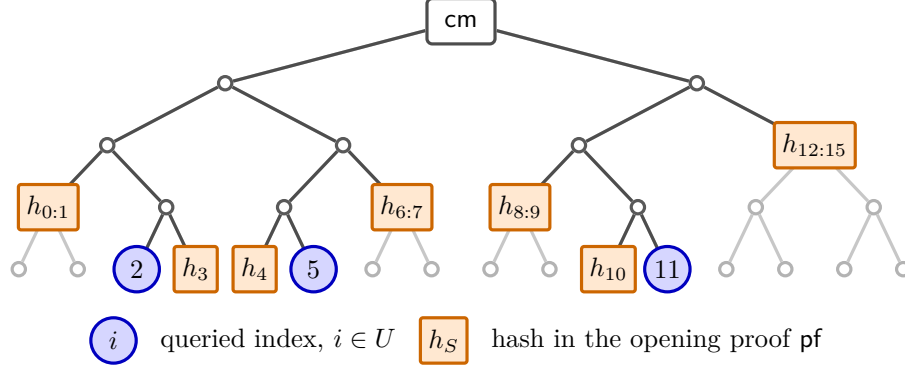


Figure 1: For query set $U = \{2, 5, 11\}$, the orange nodes form the opening proof: their descendant leaves partition $[16]_0 \setminus U$. Observe that the hashes in the opening proof do not depend on $\mathbf{x}[U]$ (opening locality). Furthermore, in order to find two distinct proofs for the same vector $\mathbf{x}$ and query set U , the opening proofs must result in a hash collision somewhere in their ancestry (proof uniqueness).

Now, recall that the goal of $\mathcal{P}$ is twofold:

1. computing a new commitment cm' to the new vector $\mathbf{z}'$ efficiently using the prover state, and
2. proving that cm' is the commitment to $\mathbf{z}'$ exactly as defined in (1) (i.e., not to an arbitrary $\mathbf{z}'$).

The first goal is easy to achieve using Merkle trees: observe that in Fig. 1, the only hashes that need to be recomputed are the $O(k \log n)$ many ancestors of the leaves in U . Thus, the time required to update the commitment is only $O_\lambda(k \log n)$, similar to the opening time and the verification time. The second goal is more challenging, since proving that cm' is the commitment to $\mathbf{z}'$ from scratch requires $\Omega(n)$ time. However, we make the following observations about Merkle trees:

1. If two vectors are equal on all indices outside of U , then the same opening proof can be used to validate both vectors. In particular, as is evident from Fig. 1, the opening proof does not depend on the values of $\mathbf{x}$ at the indices in U , but only on the values of $\mathbf{x}$ at the indices outside U . We call this property *opening locality*.
2. Assuming a canonical representation of the opening proof, finding two distinct opening proofs for a given vector, commitment and query set requires finding a collision in the hash function H . We call this property *proof uniqueness*, and this is also evident from Fig. 1.
3. Finally, a given opening proof and set of values at U can only be valid with respect to a single commitment. We call this property *commitment uniqueness*.

Together, these three properties let us prove that cm' is the commitment to $\mathbf{z}'$ efficiently. In particular, we just have to modify the transition statement to include cm' in its instance, and check that π_{vc} validates the old values $\{j : \alpha_j\}_{j \in U}$ with respect to cm and the new values $\{j : \beta_j\}_{j \in U}$ with respect to cm' . This adds $C_{\text{MT}, \text{Verify}}(m, k) = O_\lambda(k \log m)$ constraints for a second invocation of VC.Verify , preserving the bound on $N_{\text{tr}}(k)$ above. After verifying the resulting transition proof, $\mathcal{V}$ can use cm' in place of cm to process the next update. Thus, the cost of issuing this transition proof for at most k positions is

$$O_\lambda(k \log m + N_{\text{tr}}(k)P(N_{\text{tr}}(k))) = O_\lambda(k \cdot P(k \log m) \cdot \log m) = O_\lambda(k \cdot P(n) \cdot \log m) ,$$

where the last step follows from assuming $k = O(m/\log m)$ (for larger k , we can run the entire computation from scratch).

To see why this works, the security reduction first uses the static SNARK extractors to obtain $\mathbf{z}$ from π_{init} and the update values and opening proof from π_{tr} . Thus, extraction is required of the static SNARK, including against quantum adversaries for our post-quantum instantiation, but not of the auxiliary vector commitment scheme used by the compiler. We have $\text{cm} = \text{VC.Commit}(\mathbf{z})$ (as proven by π_{init}). By π_{tr} , the proof of the new transition statement, $\mathcal{V}$ is convinced that there exists some $\widehat{\pi}_{\text{vc}}$ that validates some claimed old values $\{j : \alpha_j\}_{j \in U}$ with respect to cm and some claimed new values $\{j : \beta_j\}_{j \in U}$ with respect to cm' . By position binding, the prover can only produce a valid opening proof for the true values of $\mathbf{z}$ at U , so we have $\mathbf{z}[j] = \alpha_j$ for all $j \in U$, unless a collision in H is found. Define π_{vc} to be the canonical opening proof for $\mathbf{z}$ at the indices in U . Since both π_{vc} and $\widehat{\pi}_{\text{vc}}$ validate the same claimed old values $\{j : \alpha_j\}_{j \in U}$ with respect to the same commitment cm , proof uniqueness implies that $\widehat{\pi}_{\text{vc}} = \pi_{\text{vc}}$ (unless a collision in H is found). By definition, $\mathbf{z}$ and $\mathbf{z}'$ agree outside of U , so opening locality implies that π_{vc} also validates $\{j : \beta_j\}_{j \in U}$ with respect to the honestly generated commitment $\widehat{\text{cm}}' := \text{VC.Commit}(\mathbf{z}')$. Thus, the same proof validates the values $\{j : \beta_j\}_{j \in U}$ with respect to both cm' and $\widehat{\text{cm}}'$, so commitment uniqueness implies that $\text{cm}' = \widehat{\text{cm}}'$. We defined $\widehat{\text{cm}}'$ to be the commitment to $\mathbf{z}'$, so we have shown that cm' is indeed the commitment to $\mathbf{z}'$, as desired.

Thus, the combination of the three properties of Merkle trees allows us to verifiably update the commitment efficiently, which we formalize in Definition 3.10 as *strong local updatability*. This is a strengthening of the notion of local updatability, as defined for functional commitments by [ABF24], and is also implicit in previous work on memory checking [BEG+91; BFR+13], with related SNARK-based checks of Merkle state transitions described in later work [OWWB20; SCP+22].

Wrapping up with a hierarchical data structure. However, we are not yet done. While the above argument allows $\mathcal{V}$ to obtain a verifiable commitment to the current instance-witness pair and therefore support more updates, it still leads to a $\Omega(t)$ proof size after t updates, comprising of t transition proofs and commitments in addition to the initial proof and commitment.

Fortunately, this can be resolved by compressing this chain of transition proofs by *merging* updates, computing transition proofs of the larger merged updates, and discarding the transition proofs that correspond to updates that are contained in the merged updates. We maintain a hierarchical data structure with $\ell = \lceil \log(m/\log m) \rceil = O(\log n)$ levels, where the i -th level corresponds to an update of size $k \in [2^{\ell-i}, 2^{\ell-i+1})$. For each occupied level i , a transition proof π_i certifies the transition from cm_{i-1} to cm_i , where cm_i is the Merkle commitment to the vector $\mathbf{z}_i$, the vector obtained by applying the updates stored at levels $1, \dots, i$ to the initial vector $\mathbf{z}_0 := \mathbf{z}$, in that order. The current instance-witness pair is obtained by applying the updates stored at all ℓ levels to $\mathbf{z}_0$. An empty level leaves the vector unchanged and inherits the preceding commitment.

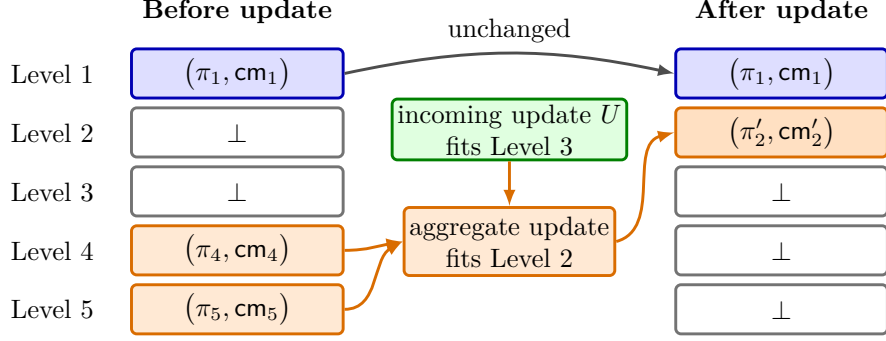


Figure 2: Example of merging process in the hierarchical data structure. Levels 1, 4, and 5 are initially occupied. The incoming update U would fit at Level 3, but it first needs to be merged with the updates at Levels 4 and 5, so that the chronological order of updates is preserved. Depending on the size of the resulting aggregate, it will be stored at Level 2 or Level 3. In this example, we assume that the aggregate fits at Level 2. Level 1 is unchanged, and the consumed lower levels are emptied.

If all levels are occupied when a new update arrives, or the incoming update has size at least 2^ℓ , we apply the update and compute a new initial proof π_{init} , emptying all levels. Otherwise, when an update of size k arrives, we begin by initializing the *aggregated update* as this incoming update and $j := \ell$, and doing the following while $j \geq 1$:

1. Take the union of the aggregated update with the update stored at level j if it exists (if there are duplicates for a position, use the latest value) to obtain a new aggregated update.
2. If the resulting update cannot be stored at level j , then empty the update at level j , decrement j , and repeat.
3. Otherwise, if it can be stored at level j (i.e., its size is in $[2^{\ell-j}, 2^{\ell-j+1})$), then store it at level j and stop.

If the aggregated update exceeds the capacity of every level, we instead compute a new initial proof for the current instance-witness pair after applying the incoming update, emptying all levels. Otherwise, once this process terminates at level s , $\mathcal{P}$ computes a new commitment cm_s to the new vector $\mathbf{z}_s$ and a new transition proof π_s for the transition statement from cm_{s-1} to cm_s . $\mathcal{V}$ can verify the new claim by invoking $\mathbf{V}$ on the initial proof π_{init} and the transition proofs $\pi_1, \dots, \pi_s$ with respect to the initial commitment cm and the subsequent commitments $\text{cm}_1, \dots, \text{cm}_s$.

We now briefly argue the amortized prover cost for this construction. Define $b_i := 2^{\ell-i}$. An update on level i involves fewer than $2b_i$ positions and therefore costs $O_\lambda(b_i P(n) \log n)$ to compute a proof for. Before the merge reaches level i , the incoming update together with the updates at levels $i+1, \dots, \ell$ must already contain at least b_i indices, otherwise the merged update can be stored at level $i+1$. These updates have accumulated since initialization or the last building of level i , both of which would have emptied those lower levels. Charging $O_\lambda(k P(n) \log n)$ to each level for an incoming update of size k therefore pays for every rebuild: the sum of their charges is $\Omega_\lambda(b_i P(n) \log n)$, and their charges have not previously been spent at that level. Across $O(\log n)$ levels, this gives a total amortized charge of $O_\lambda(k P(n) \log^2 n)$. Reinitialization requires $\Omega(m / \log m)$ accumulated update volume, so its cost is covered by an additional charge of $O((k \log m / m) \cdot T_{\text{init}})$.

per update. Since Merkle trees give $T_{\text{init}} = O_\lambda(n \cdot P(n))$, the amortized update time is

$$O_\lambda \left(\frac{k \log m}{m} \cdot T_{\text{init}} + k \cdot P(n) \cdot \log^2 n \right) = O_\lambda \left(k \cdot P(n) \cdot \log^2 n \right) .$$

Summary. This completes the compiler at a high level. The initial proof certifies the initial instance-witness pair, a shared vector-commitment opening certifies each transition between committed pairs, and the hierarchical data structure prevents the number of transition proofs from growing with the number of updates. Note that there are a few details that we have omitted in the overview for simplicity:

1. **Access to the permutation:** $\mathcal{V}$ does not have access to the permutation σ in whole since we want its work to be $O(n_0) + o(n)$. In the full construction, a publicly verifiable indexer commits to σ as well and sends cm_σ to $\mathcal{V}$. The initial and transition statements additionally prove that the claimed permutation is consistent with cm_σ .
2. **Public input consistency:** It is not sufficient for $\mathcal{P}$ to prove that it possesses some valid instance-witness pair: it needs to prove that the instance in the claimed pair is equal to the public input $\mathbf{x}$. This can also be handled with vector commitments. The prover initially opens the public-input positions in time $T_{\text{VC.Open}}(m, n_0)$. For the Merkle layout in Corollary 4.2, the update costs $O_\lambda(1)$ once the updated commitment state is available for a transition affecting k positions.
3. **Black-box use of vector commitments:** We discussed how to use Merkle trees to achieve fully-dynamic SNARKs. However, our compiler in Section 4 is more general and can be instantiated with vector commitments satisfying opening locality, commitment uniqueness, and proof uniqueness, together with the completeness, binding, and efficiency requirements stated there.
4. **Specialized PIOPs for the relations:** We used static SNARKs as black-boxes to prove an initial relation and a transition relation. To obtain a concretely efficient implementation, we construct specialized polynomial IOPs (PIOPs) for these relations in Section 5.
5. **Deamortization of updates:** With some modifications to the hierarchical data structure, the amortized update-time bound above holds in the worst-case. The details are discussed in Appendix B.
6. **Zero-knowledge:** In Appendix C, we describe a modification of our compiler that provides zero-knowledge for the final witness (but not for the entire history of witnesses), given hiding of the vector commitment scheme, and zero-knowledge of the static SNARK. We provide an informal argument for the same.

3 Preliminaries

Computation models. Unless stated otherwise, running times in this paper are measured in the word-RAM model with word length $\Theta(\lambda + \log n)$, where n denotes the size of the relevant input, and λ is the security parameter. In particular, field operations over a field of $\Theta(\lambda)$ -bit elements are assumed to take $O(1)$ time.

Indexed relations. An indexed NP relation is a family $\mathcal{R} = \{\mathcal{R}_\lambda(\mathfrak{i}) \subseteq \{0, 1\}^* \times \{0, 1\}^*\}_{\lambda, \mathfrak{i}}$, where $\mathfrak{i}$ describes the relation (for example, a circuit) and $(\mathfrak{x}, \mathfrak{w}) \in \mathcal{R}_\lambda(\mathfrak{i})$ denotes a valid instance-witness pair.

Definition 3.1 (Plonkish circuit-satisfiability relation). For each security parameter λ , fix a finite field $\mathbb{F}$ of $\Theta(\lambda)$ -bit elements. We use the following Plonkish encoding [GWC19; CBBZ23]. An index of the *Plonkish circuit-satisfiability relation* $\mathcal{R}_{\text{Plonk}}$ is a fan-in-2 arithmetic circuit $\mathcal{C}$ over $\mathbb{F}$ with n addition gates, n multiplication gates, and $n_0 \leq n$ public-input occurrences, padding with dummy gates if necessary. Set $m := 6n + n_0$. The permutation $\sigma_{\mathcal{C}} \in S_m$ encodes the wiring of $\mathcal{C}$: the cycles of $\sigma_{\mathcal{C}}$ are exactly the sets of positions that represent the same circuit wire, including the public-input positions.

For $\mathfrak{x} \in \mathbb{F}^{n_0}$ and $\mathfrak{w} \in \mathbb{F}^{6n}$, let $\mathbf{z} := [\mathfrak{w}, \mathfrak{x}] \in \mathbb{F}^m$. For each $i \in [n]_0$, positions $6i, 6i+1, 6i+2$ contain the left input, right input, and output of the i th addition gate, and positions $6i+3, 6i+4, 6i+5$ contain the corresponding values for the i th multiplication gate. We define $(\mathfrak{x}, \mathfrak{w}) \in \mathcal{R}_{\text{Plonk}}(\mathcal{C})$ if and only if

$$\begin{aligned} \mathbf{z}[6i] + \mathbf{z}[6i+1] &= \mathbf{z}[6i+2] && \text{for every } i \in [n]_0, \\ \mathbf{z}[6i+3] \cdot \mathbf{z}[6i+4] &= \mathbf{z}[6i+5] && \text{for every } i \in [n]_0, \\ \mathbf{z}[j] &= \mathbf{z}[\sigma_{\mathcal{C}}[j]] && \text{for every } j \in [m]_0. \end{aligned}$$

The distance between two Plonkish instances $\mathbf{z} = [\mathfrak{w}, \mathfrak{x}]$ and $\mathbf{z}' = [\mathfrak{w}', \mathfrak{x}']$ is the Hamming distance between their concatenated vectors.

3.1 Cryptographic primitives

Definition 3.2 (Preprocessing SNARK). Abstracting the interfaces of Marlin [CHM+20] and Fractal [COS20], a *preprocessing succinct non-interactive argument of knowledge* (SNARK) for $\mathcal{R}$ consists of four PPT algorithms $\text{SNARK} := (\mathbf{G}, \mathbf{I}, \mathbf{P}, \mathbf{V})$:

- $\mathbf{G}(1^\lambda, N)$ outputs a common reference string crs that supports indices $\mathfrak{i}$ of size at most N . We do not restrict the distribution of crs . In particular, it may be structured or uniformly random.
- $\mathbf{I}(\text{crs}, \mathfrak{i})$ is a deterministic algorithm that outputs a proving key pk and a verification key vk for the indexed relation $\mathcal{R}(\mathfrak{i})$.
- $\mathbf{P}(\text{pk}, \mathfrak{x}, \mathfrak{w})$ outputs a proof pf , and $\mathbf{V}(\text{vk}, \mathfrak{x}, \text{pf})$ outputs a bit.

We require the following properties:

- **Perfect completeness:** For every $\lambda, N \in \mathbb{N}$ and every efficient $\mathcal{A}$ that outputs an index of size at most N ,

$$\Pr \left[\begin{array}{c} (\mathbb{x}, \mathbb{w}) \notin \mathcal{R}_\lambda(\mathbb{i}) \\ \vee \quad \mathbf{V}(\mathbf{vk}, \mathbb{x}, \mathbf{pf}) = 1 \end{array} \middle| \begin{array}{c} \text{crs} \leftarrow \mathbf{G}(1^\lambda, N) \\ (\mathbb{i}, \mathbb{x}, \mathbb{w}) \leftarrow \mathcal{A}(\text{crs}) \\ (\mathbf{pk}, \mathbf{vk}) \leftarrow \mathbf{I}(\text{crs}, \mathbb{i}) \\ \mathbf{pf} \leftarrow \mathbf{P}(\mathbf{pk}, \mathbb{x}, \mathbb{w}) \end{array} \right] = 1 .$$

- **Knowledge soundness:** We say that SNARK has *knowledge soundness error* $\epsilon_{\text{ks}}(\lambda, N, t)$ and *extractor circuit size* $C_{\mathbf{E}}(\lambda, N, t)$ if, for every non-uniform adversary $\tilde{\mathbf{P}} = (\tilde{\mathbf{P}}_1, \tilde{\mathbf{P}}_2)$ of size at most $t = t(\lambda)$ whose first stage outputs an index of size at most N , there exists an extractor $\mathbf{E}^{\tilde{\mathbf{P}}}$ implementable by a circuit of size at most $C_{\mathbf{E}}(\lambda, N, t)$ such that

$$\Pr \left[\begin{array}{c} (\mathbb{x}, \mathbb{w}) \notin \mathcal{R}_\lambda(\mathbb{i}) \\ \wedge \quad \mathbf{V}(\mathbf{vk}, \mathbb{x}, \mathbf{pf}) = 1 \end{array} \middle| \begin{array}{c} \text{crs} \leftarrow \mathbf{G}(1^\lambda, N) \\ (\mathbb{i}, \mathbb{x}, \text{st}) \leftarrow \tilde{\mathbf{P}}_1(\text{crs}) \\ (\mathbf{pk}, \mathbf{vk}) \leftarrow \mathbf{I}(\text{crs}, \mathbb{i}) \\ \mathbf{pf} \leftarrow \tilde{\mathbf{P}}_2(\text{crs}, \mathbf{pk}, \mathbf{vk}, \text{st}) \\ \mathbb{w} \leftarrow \mathbf{E}^{\tilde{\mathbf{P}}}(\text{crs}, \mathbf{pk}, \mathbf{vk}, \text{st}) \end{array} \right] \leq \epsilon_{\text{ks}}(\lambda, N, t) .$$

We leave the extractor's permitted access to $\tilde{\mathbf{P}}$ implicit. The dynamic SNARK compiler in Section 4 preserves this extractor model: the dynamic SNARK extractor uses exactly the access provided to the underlying static SNARK extractors, whether that be no access, black-box rewinding access, or full tape access.

- **Succinctness:** The proof length $|\mathbf{pf}|$ is bounded by $\text{poly}(\lambda, \log |\mathbb{w}|)$.

Definition 3.3 (Preprocessing dynamic SNARK). Following the dynamic-SNARK interface of Wang et al. [WSP26b], a *preprocessing dynamic succinct non-interactive argument of knowledge* for an indexed relation $\mathcal{R}$ consists of five PPT algorithms

$$\text{DYN} := (\mathcal{G}, \mathcal{I}, \mathcal{P}_{\text{init}}, \mathcal{P}_{\text{update}}, \mathcal{V}) ,$$

where $\mathcal{G}$, $\mathcal{I}$, and $\mathcal{V}$ have the same interface as in Definition 3.2. Additionally,

- $\mathcal{P}_{\text{init}}(\mathbf{pk}, \mathbb{x}_0, \mathbb{w}_0)$ outputs an initial proof Π_0 and prover state $\mathbb{M}_0$.
- $\mathcal{P}_{\text{update}}(\mathbf{pk}, \mathbb{M}, (\mathbb{x}, \mathbb{w}), U, \{\beta_j\}_{j \in U})$ receives the instance-witness pair $(\mathbb{x}, \mathbb{w})$ before the update, an update index set consisting U and new values $\{\beta_j\}_{j \in U}$. Writing $\mathbf{z} = [\mathbb{w}, \mathbb{x}]$, the update defines the target pair $(\mathbb{x}', \mathbb{w}')$ through $\mathbf{z}' = [\mathbb{w}', \mathbb{x}']$, where

$$\mathbf{z}'[j] := \begin{cases} \beta_j & j \in U, \\ \mathbf{z}[j] & j \notin U. \end{cases}$$

The algorithm outputs a proof Π' and updated prover state $\mathbb{M}'$.

Knowledge soundness and succinctness is also the same as in Definition 3.2, but completeness needs to be defined for sequences of updates, as follows.

Perfect completeness for updates: Fix keys output by $\mathcal{G}$ and $\mathcal{I}$, and any sequence of valid pairs $((\mathbf{x}_s, \mathbf{w}_s))_{s=0}^T$ in $\mathcal{R}(\mathfrak{i})$ and sparse updates $(U_s, \{\beta_{s,i}\}_{i \in U_s})_{s=1}^T$ that transform each pair into the next one as above. If $(\Pi_0, \mathbb{M}_0)$ is generated by $\mathcal{P}_{\text{init}}$ and, for every $s \in [T]$,

$$(\Pi_s, \mathbb{M}_s) \leftarrow \mathcal{P}_{\text{update}}(\mathbf{pk}, \mathbb{M}_{s-1}, (\mathbf{x}_{s-1}, \mathbf{w}_{s-1}), U_s, \{\beta_{s,i}\}_{i \in U_s}),$$

then $\mathcal{V}(\mathbf{vk}, \mathbf{x}_s, \Pi_s) = 1$ for every $s \in \{0, \dots, T\}$ with probability one.

Definition 3.4 (Hash family). A *hash family* with input length $\nu(\lambda)$ and output length $\mu(\lambda)$ is a set $\mathcal{H} = \{h_\lambda : \{0, 1\}^\lambda \times \{0, 1\}^{\nu(\lambda)} \rightarrow \{0, 1\}^{\mu(\lambda)}\}$ of keyed hash functions, in addition to a pair of PPT algorithms:

- $\mathcal{H}.\text{Gen}(1^\lambda)$ outputs a hash key $k \xleftarrow{\$} \{0, 1\}^\lambda$ describing a hash function h .
- $\mathcal{H}.\text{Hash}(k, x)$ outputs $h_\lambda(k, x) = h(x)$.

Definition 3.5 (Collision resistance). A hash family $\mathcal{H} = \{h_\lambda : \{0, 1\}^\lambda \times \{0, 1\}^{\nu(\lambda)} \rightarrow \{0, 1\}^{\mu(\lambda)}\}$ with input length $\nu(\lambda)$ and output length $\mu(\lambda)$ has *collision resistance error* $\epsilon_{\text{cr}}(\lambda, t)$ if for any t -size adversary $\mathcal{A}_\lambda$, we have

$$\Pr \left[\begin{array}{c} h_\lambda(k, x) = h_\lambda(k, x') \\ \wedge \quad x \neq x' \end{array} \middle| \begin{array}{c} k \leftarrow \mathcal{H}.\text{Gen}(1^\lambda) \\ x, x' \leftarrow \mathcal{A}_\lambda(k) \end{array} \right] \leq \epsilon_{\text{cr}}(\lambda, t) .$$

We say that $\mathcal{H}$ is *collision resistant* if $\epsilon_{\text{cr}}(\lambda, t) = \text{negl}(\lambda)$ for every polynomially bounded $t(\lambda)$.

Definition 3.6 (Vector commitments). A *vector commitment (VC) scheme* VC is a tuple of four PPT algorithms ($\text{VC}.\text{Setup}$, $\text{VC}.\text{Commit}$, $\text{VC}.\text{Open}$, $\text{VC}.\text{Verify}$):

- The setup algorithm $\text{VC}.\text{Setup}(1^\lambda, N)$ outputs public parameters $\mathbf{pp}$ for security parameter λ and vector length upper bound N .
- The commitment algorithm $\text{VC}.\text{Commit}(\mathbf{pp}, \mathbf{x})$ takes as input a vector $\mathbf{x} \in (\{0, 1\}^{\ell(\lambda)})^n$ where $n \leq N$, and outputs $(\mathbf{cm}, \mathbf{aux})$, where $\mathbf{cm}$ is a commitment and $\mathbf{aux}$ is auxiliary information.
- The opening algorithm $\text{VC}.\text{Open}(\mathbf{pp}, \mathbf{x}, S, \mathbf{aux})$ takes as input an index set $S \subseteq [n]_0$ and the auxiliary information $\mathbf{aux}$ output by $\text{VC}.\text{Commit}$ and outputs an opening proof $\mathbf{pf}$.
- The verification algorithm $\text{VC}.\text{Verify}(\mathbf{pp}, \mathbf{cm}, S, \mathbf{x}[S], \mathbf{pf})$ outputs a bit.

In this work, we only consider deterministic commitment, open, and verify algorithms, and use $\text{Com}_{\mathbf{pp}}(\mathbf{x})$ to denote $\mathbf{cm}$.

We consider the following standard properties of VC schemes:

- **Perfect completeness:** Consider any $\mathbf{pp}$ output by $\text{VC}.\text{Setup}(1^\lambda, N)$, any $\mathbf{x} \in (\{0, 1\}^{\ell(\lambda)})^n$ where $n \leq N$, and any non-empty $S \subseteq [n]_0$. Let $(\mathbf{cm}, \mathbf{aux}) \leftarrow \text{VC}.\text{Commit}(\mathbf{pp}, \mathbf{x})$. Then

$$\text{VC}.\text{Verify}(\mathbf{pp}, \mathbf{cm}, S, \mathbf{x}[S], \text{VC}.\text{Open}(\mathbf{pp}, \mathbf{x}, S, \mathbf{aux})) = 1 .$$

- **Strong position binding:** A vector commitment scheme has strong position binding error $\epsilon_{\text{spd}}^{\text{vc}}(\lambda, n, q, t)$ if for any t -size adversary $\mathcal{A}_\lambda$, we have

$$\Pr \left[\begin{array}{l} \exists i \in Q \cap Q' : \mathbf{x}[i] \neq \mathbf{x}'[i] \\ \wedge \text{VC.Verify}(\text{pp}, \text{cm}, Q, \mathbf{x}[Q], \text{pf}) = 1 \\ \wedge \text{VC.Verify}(\text{pp}, \text{cm}, Q', \mathbf{x}'[Q'], \text{pf}') = 1 \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{VC.Setup}(1^\lambda, N) \\ (\mathbf{x}, \mathbf{x}', Q, Q', \text{pf}, \text{pf}', \text{cm}) \\ \leftarrow \mathcal{A}_\lambda(\text{pp}) \\ Q, Q' \subseteq [n]_0 \text{ and } |Q|, |Q'| \leq q \end{array} \right] \leq \epsilon_{\text{spd}}^{\text{vc}}(\lambda, n, q, t) .$$

We also consider the following non-standard properties which are satisfied by some VC schemes.

Definition 3.7 (Opening locality). A vector commitment scheme VC satisfies *opening locality* if for every pp, every $\mathbf{x}, \mathbf{x}' \in (\{0, 1\}^{\ell(\lambda)})^n$, and every non-empty $S \subseteq [n]_0$ such that $\mathbf{x}[i] = \mathbf{x}'[i]$ for all $i \in [n]_0 \setminus S$, the following holds. Let $(\text{cm}, \text{aux}) \leftarrow \text{VC.Commit}(\text{pp}, \mathbf{x})$, let $(\text{cm}', \text{aux}') \leftarrow \text{VC.Commit}(\text{pp}, \mathbf{x}')$, and let $\text{pf} \leftarrow \text{VC.Open}(\text{pp}, \mathbf{x}, S, \text{aux})$. Then

$$\text{VC.Verify}(\text{pp}, \text{cm}', S, \mathbf{x}'[S], \text{pf}) = 1 .$$

Definition 3.8 (Commitment uniqueness). A vector commitment scheme VC satisfies *commitment uniqueness* if for every pp, every non-empty $S \subseteq [n]_0$, every $\mathbf{x}[S]$, every pf, and every pair of commitments cm, cm', if

$$\text{VC.Verify}(\text{pp}, \text{cm}, S, \mathbf{x}[S], \text{pf}) = \text{VC.Verify}(\text{pp}, \text{cm}', S, \mathbf{x}[S], \text{pf}) = 1 ,$$

then $\text{cm} = \text{cm}'$.

Definition 3.9 (Proof uniqueness). A vector commitment scheme has *proof uniqueness error* $\epsilon_{\text{pu}}^{\text{vc}}(\lambda, n, q, t)$ if for any t -size adversary $\mathcal{A}_\lambda$, we have

$$\Pr \left[\begin{array}{l} \text{pf} \neq \text{pf}' \\ \wedge \text{VC.Verify}(\text{pp}, \text{cm}, Q, \mathbf{x}[Q], \text{pf}) = 1 \\ \wedge \text{VC.Verify}(\text{pp}, \text{cm}, Q, \mathbf{x}[Q], \text{pf}') = 1 \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{VC.Setup}(1^\lambda, N) \\ (\text{cm}, \mathbf{x}, Q, \text{pf}, \text{pf}') \leftarrow \mathcal{A}_\lambda(\text{pp}) \\ Q \subseteq [n]_0 \text{ and } |Q| \leq q \end{array} \right] \leq \epsilon_{\text{pu}}^{\text{vc}}(\lambda, n, q, t) . \quad (2)$$

We also introduce the following notion of *strong local updatability* which is implied by opening locality, commitment uniqueness, and proof uniqueness. A similar notion of *local updatability* is defined in the context of functional commitments by Anthoine et al. [ABF24]. We strengthen their notion, adapt it to vector commitments, and show that it is implied by the three properties above.

Definition 3.10 (Strong local updatability). A vector commitment scheme $\mathcal{V}$ has *strong local updatability error* $\epsilon_{\text{slu}}^{\text{vc}}(\lambda, n, q, t)$ if, for any t -size adversary $\mathcal{A}_\lambda$, we have

$$\Pr \left[\begin{array}{l} \left(\begin{array}{l} \text{cm} \neq \text{Com}_{\text{pp}}(\mathbf{x}) \\ \wedge \text{cm}' = \text{Com}_{\text{pp}}(\mathbf{x}') \end{array} \right) \\ \vee \left(\begin{array}{l} \text{cm} = \text{Com}_{\text{pp}}(\mathbf{x}) \\ \wedge \text{cm}' \neq \text{Com}_{\text{pp}}(\mathbf{x}') \end{array} \right) \end{array} \middle| \begin{array}{l} \text{pp} \leftarrow \text{VC.Setup}(1^\lambda, N) \\ (\mathbf{x}, \mathbf{x}', Q, \text{pf}, \text{cm}, \text{cm}') \leftarrow \mathcal{A}_\lambda(\text{pp}) \\ Q \subseteq [n]_0, \quad 1 \leq |Q| \leq q \\ \forall i \in [n]_0 \setminus Q : \mathbf{x}[i] = \mathbf{x}'[i] \\ \text{VC.Verify}(\text{pp}, \text{cm}, Q, \mathbf{x}[Q], \text{pf}) = 1 \\ \text{VC.Verify}(\text{pp}, \text{cm}', Q, \mathbf{x}'[Q], \text{pf}) = 1 \end{array} \right] \leq \epsilon_{\text{slu}}^{\text{vc}}(\lambda, n, q, t) . \quad (3)$$

The above is close to a converse of opening locality: if the same proof authenticates the entries in Q to the commitments cm and cm' for two vectors $\mathbf{x}$ and $\mathbf{x}'$ that agree outside Q , then cm is a valid commitment to $\mathbf{x}$ if and only if cm' is a valid commitment to $\mathbf{x}'$. We use the qualifier *strong* to distinguish this property from the local updatability notion of [ABF24].

Lemma 3.11 (Strong local updatability from proof uniqueness). *Let VC be a vector commitment scheme with perfect completeness, opening locality, and commitment uniqueness. Let t' upper-bound the size of a circuit that implements a t -size adversary, two invocations of the commitment circuit on length- n vectors, and one invocation of the opening circuit on a batch of size q . If VC has proof uniqueness error $\epsilon_{\text{pu}}^{\text{vc}}(\lambda, n, q, t')$, then VC has strong local updatability error*

$$\epsilon_{\text{slu}}^{\text{vc}}(\lambda, n, q, t) \leq \epsilon_{\text{pu}}^{\text{vc}}(\lambda, n, q, t') .$$

Proof. Suppose that $(\mathbf{x}, \mathbf{x}', Q, \text{pf}, \text{cm}, \text{cm}')$ satisfies the conditions in (3), including $\text{cm} = \text{Com}_{\text{pp}}(\mathbf{x})$ without loss of generality. Let $(\widehat{\text{cm}}, \text{aux}) \leftarrow \text{VC.Commit}(\text{pp}, \mathbf{x})$ and define $\widehat{\text{pf}} \leftarrow \text{VC.Open}(\text{pp}, \mathbf{x}, Q, \text{aux})$. Since VC.Commit is deterministic, $\widehat{\text{cm}} = \text{cm}$. Perfect completeness implies that pf verifies for $(\text{cm}, Q, \mathbf{x}[Q])$. Because pf verifies for the same claim, proof uniqueness implies that $\text{pf} = \widehat{\text{pf}}$, except with probability $\epsilon_{\text{pu}}^{\text{vc}}(\lambda, n, q, t')$.

Let $(\widehat{\text{cm}}', \text{aux}') \leftarrow \text{VC.Commit}(\text{pp}, \mathbf{x}')$. Since $\mathbf{x}$ and $\mathbf{x}'$ agree outside Q , opening locality implies that $\widehat{\text{pf}}$ also verifies for $(\widehat{\text{cm}}', Q, \mathbf{x}'[Q])$. Thus, conditioned on $\text{pf} = \widehat{\text{pf}}$, pf verifies for $(\widehat{\text{cm}}', Q, \mathbf{x}'[Q])$. pf also verifies for $(\text{cm}', Q, \mathbf{x}'[Q])$ by assumption, so commitment uniqueness implies that $\text{cm}' = \widehat{\text{cm}}'$. This contradicts the strong-local-updatability event. $\square$

Definition 3.12 (Updatable vector commitments). Consider a vector commitment scheme VC and let $\text{pp} \leftarrow \text{VC.Setup}(1^\lambda, N)$ where $\lambda, N \in \mathbb{N}$. Let $(\text{cm}, \text{aux}) \leftarrow \text{VC.Commit}(\text{pp}, \mathbf{x})$ for some $\mathbf{x} \in (\{0, 1\}^{\ell(\lambda)})^n$ where $n \leq N$. Given an update set $U \subseteq [n]_0$ and replacement values $\{\beta_i\}_{i \in U}$, define

$$\mathbf{x}'[i] := \begin{cases} \beta_i & \text{if } i \in U \\ \mathbf{x}[i] & \text{otherwise} \end{cases} .$$

- We say that VC is *commitment-updatable* in time $T_{\text{cupd}}^{\text{vc}}(\lambda, n, u)$ if there is an algorithm

$$\text{VC.UpdateCommit}(\text{pp}, U, \{\beta_i\}_{i \in U}, \text{cm}, \text{aux}) \longrightarrow (\text{cm}', \text{aux}')$$

that runs in time $T_{\text{cupd}}^{\text{vc}}(\lambda, n, u)$ whenever $|U| \leq u$, and its output satisfies $(\text{cm}', \text{aux}') = \text{VC.Commit}(\text{pp}, \mathbf{x}')$.

- We say that VC is *proof-updatable* in time $T_{\text{pupd}}^{\text{vc}}(\lambda, n, u, q)$ if there is an algorithm

$$\text{VC.UpdateProof}(\text{pp}, \text{pf}, \text{aux}', U, P) \longrightarrow \text{pf}'$$

with the following property. For every update set $U \subseteq [n]_0$ of size at most u , every non-empty query set $P \subseteq [n]_0$ of size at most q , and every $\text{pf} = \text{VC.Open}(\text{pp}, \mathbf{x}, P, \text{aux})$, the algorithm runs in time $T_{\text{pupd}}^{\text{vc}}(\lambda, n, u, q)$ and outputs a proof pf' such that $\text{pf}' = \text{VC.Open}(\text{pp}, \mathbf{x}', P, \text{aux}')$ where $(\text{cm}', \text{aux}') = \text{VC.Commit}(\text{pp}, \mathbf{x}')$.

3.2 Merkle commitments

We recall the Merkle-tree commitment underlying Merkle's hash-based signature scheme [Mer87].

Definition 3.13 (Merkle commitments). Let $\mathcal{H} = \{h_\lambda : \{0, 1\}^\lambda \times \{0, 1\}^{\nu(\lambda)} \rightarrow \{0, 1\}^{\mu(\lambda)}\}$ be a *collision-resistant* hash family with input length $\nu(\lambda)$ and output length $\mu(\lambda)$. A *Merkle commitment scheme* based on $\mathcal{H}$ is a vector commitment scheme $\text{MT}(\mathcal{H}) = (\text{MT.Setup}, \text{MT.Commit}, \text{MT.Open}, \text{MT.Verify})$ such that:

- $\text{MT.Setup}(1^\lambda, N)$ outputs a hash key $k \leftarrow \mathcal{H}.\text{Gen}(1^\lambda)$.
- $\text{MT.Commit}(k, \mathbf{x})$ takes as input a hash key k and a vector $\mathbf{x} \in (\{0, 1\}^{\ell(\lambda)})^n$ where $n \leq N$.⁸ It behaves as follows:
 1. For each $i \in [n]_0$, compute $\alpha_{n+i} = h_\lambda(k, \langle 0, \mathbf{x}[i] \rangle)$.⁹
 2. For $i = n-1, n-2, \dots, 1$ (in that order), compute $\alpha_i = h_\lambda(k, \langle 1, \alpha_{2i}, \alpha_{2i+1} \rangle)$.
 3. Output a commitment $\text{cm} = \alpha_1$ and auxiliary information $\text{aux} = \{\alpha_i\}_{i \in [2n-1]}$.
- $\text{MT.Open}(k, \mathbf{x}, S, \text{aux})$ takes as input a hash key k , a vector $\mathbf{x} \in (\{0, 1\}^{\ell(\lambda)})^n$, a non-empty set of indices $S \subseteq [n]_0$, and the auxiliary information aux output by MT.Commit . It invokes the method $\text{OpenHelper}(0, n, 1)$ to obtain $\text{pf} = H$, where $\text{OpenHelper}(\ell, r, j)$ is recursively defined as follows:
 1. If $\ell = r$, return $\emptyset$.
 2. If $\ell + 1 = r$, return $\{j : \alpha_j\}$ if $\ell \notin S$, otherwise return $\emptyset$.
 3. If $S \cap [\ell, r-1] = \emptyset$, return $\{j : \alpha_j\}$.
 4. Otherwise, let

$$\begin{aligned} H_L &= \text{OpenHelper}(\ell, \lfloor (\ell + r)/2 \rfloor, 2j), \\ H_R &= \text{OpenHelper}(\lfloor (\ell + r)/2 \rfloor, r, 2j + 1). \end{aligned}$$

Return $H_L \cup H_R$.

- $\text{MT.Verify}(k, \text{cm}, S, \mathbf{x}[S], \text{pf} = H)$ outputs a bit indicating whether pf is a valid opening of cm at the indices in S to the values $\mathbf{x}[S]$. It invokes $\text{VerifyHelper}(0, n, 1)$ to obtain a hash value h , where $\text{VerifyHelper}(\ell, r, j)$ is recursively defined as follows:
 1. If $\ell + 1 = r$, return $h_\lambda(k, \langle 0, \mathbf{x}[\ell] \rangle)$ if $\ell \in S$, otherwise return $H[j]$ if $j \in H$ (and otherwise MT.Verify returns 0).
 2. If $S \cap [\ell, r-1] = \emptyset$, return $H[j]$ if $j \in H$ (and otherwise MT.Verify returns 0).
 3. Otherwise, let

$$\begin{aligned} h_L &= \text{VerifyHelper}(\ell, \lfloor (\ell + r)/2 \rfloor, 2j), \\ h_R &= \text{VerifyHelper}(\lfloor (\ell + r)/2 \rfloor, r, 2j + 1). \end{aligned}$$

Return $h_\lambda(k, \langle 1, h_L, h_R \rangle)$.

⁸For simplicity, we assume that n is a power of 2. If not, we can pad $\mathbf{x}$ with dummy values.

⁹ $\langle \cdot \rangle$ denotes an injective canonical encoding of the tuple that outputs a string of length $\nu(\lambda)$; the constants 0 and 1 distinguish leaf and internal-node inputs.

Finally, MT.Verify additionally checks that H does not contain any extra keys that were not queried by VerifyHelper , and that the keys are sorted. If not, it returns 0.¹⁰ Finally, it returns 1 if $h = \text{cm}$ and 0 otherwise.

Properties of Merkle commitments. Node indices determine the verifier's tree traversal and are not hash inputs. The Merkle commitment scheme $\text{MT}(\mathcal{H})$ is an instance of Definition 3.6. Let $T_h(\lambda)$ denote the time to evaluate $h_\lambda(k, \cdot)$, and let $C_h(\lambda)$ denote the size of an arithmetic circuit that implements $h_\lambda(k, \cdot)$. Finally, let t_{MT} be an upper bound on the circuit size of the adversary. Given these parameters, $\text{MT}(\mathcal{H})$ satisfies the following properties:

- Perfect completeness by construction.
- Strong position binding: if $\mathcal{H}$ has collision resistance error as in Definition 3.5, then $\text{MT}(\mathcal{H})$ has strong position binding error $\epsilon_{\text{spd}}^{\text{vc}}(\lambda, n, q, t_{\text{MT}}) \leq \epsilon_{\text{cr}}(\lambda, t_{\text{MT}} + O(C_h(\lambda) \cdot n))$. At a differing opened position, either the two distinct leaf-hash inputs already collide, or the corresponding leaf hashes differ. In the latter case, since both proofs verify under the same commitment, climbing from that leaf to the root yields a node v whose two hash values differ but whose parent hash values are equal, again yielding a collision for $h_\lambda(k, \cdot)$.
- Opening locality: if $\mathbf{x}$ and $\mathbf{x}'$ agree outside S , then every node hash included in $\text{pf} \leftarrow \text{MT.Open}(k, \mathbf{x}, S, \text{aux})$ is the hash of a subtree whose leaves are disjoint from S . Thus, each hash in pf appears in $\text{pf}' \leftarrow \text{MT.Open}(k, \mathbf{x}', S, \text{aux})$, and we get $\text{pf} = \text{pf}'$. Opening locality then follows from perfect completeness.
- Commitment uniqueness: for fixed k , S , $\mathbf{x}[S]$, and H , the verifier either rejects or computes a single root value h . Acceptance under a commitment cm is exactly the condition $h = \text{cm}$.

We formalize the property of proof uniqueness for Merkle commitments in the following lemma.

Lemma 3.14 (Proof uniqueness of Merkle commitments). *Let $\mathcal{H} = \{h_\lambda : \{0, 1\}^\lambda \times \{0, 1\}^{\nu(\lambda)} \rightarrow \{0, 1\}^{\mu(\lambda)}\}$ be a collision-resistant hash family. Then, $\text{MT}(\mathcal{H})$ has proof uniqueness error $\epsilon_{\text{pu}}^{\text{vc}}(\lambda, n, q, t_{\text{MT}}) \leq \epsilon_{\text{cr}}(\lambda, t_{\text{MT}} + O(C_h(\lambda) \cdot n))$.*

Proof. Consider a t_{MT} -size adversary $\mathcal{A}_\lambda$ that outputs $(\text{cm}, \mathbf{x}, Q, \text{pf}, \text{pf}')$ such that $\text{pf} \neq \text{pf}'$ and both proofs verify for the same claim $(\text{cm}, Q, \mathbf{x}[Q])$. It suffices to show how to obtain a collision for $h_\lambda(k, \cdot)$ from such an output.

Since MT.Verify only accepts canonical proofs, pf and pf' have the same support S_Q determined by Q . Let γ_v and γ'_v be the values computed at node v in the verifier executions using pf and pf' , respectively. Both executions accept under the same commitment cm , so $\gamma_1 = \gamma'_1 = \text{cm}$. Since $\text{pf} \neq \text{pf}'$, there is some node $v \in S_Q$ such that $\gamma_v \neq \gamma'_v$. Consider the path from v to the root. There is a node u on this path such that $\gamma_u \neq \gamma'_u$ but $\gamma_p = \gamma'_p$ where $p = \lfloor u/2 \rfloor$ is the parent of u . Then, we have

$$h_\lambda(k, \langle 1, \gamma_{2p}, \gamma_{2p+1} \rangle) = h_\lambda(k, \langle 1, \gamma'_{2p}, \gamma'_{2p+1} \rangle) ,$$

while $\langle 1, \gamma_{2p}, \gamma_{2p+1} \rangle \neq \langle 1, \gamma'_{2p}, \gamma'_{2p+1} \rangle$ since u is either $2p$ or $2p+1$. This is a collision for $h_\lambda(k, \cdot)$. $\square$

¹⁰This check ensures that the verifier only accepts an opening proof that has a canonical representation: the dictionary's keys are sorted, and does not contain any hashes beyond those needed to verify the opening.

Lemma 3.15 (Merkle strong local updatability). *Let $\mathcal{H} = \{h_\lambda : \{0, 1\}^\lambda \times \{0, 1\}^{\nu(\lambda)} \rightarrow \{0, 1\}^{\mu(\lambda)}\}$ be a collision-resistant hash family. Then $\text{MT}(\mathcal{H})$ has strong local updatability error*

$$\epsilon_{\text{slu}}^{\text{vc}}(\lambda, n, q, t_{\text{MT}}) \leq \epsilon_{\text{cr}}(\lambda, t_{\text{MT}} + O(C_h(\lambda) \cdot n)) .$$

Proof. Merkle commitments have perfect completeness, opening locality, and commitment uniqueness, and Lemma 3.14 bounds their proof uniqueness error by $\epsilon_{\text{cr}}(\lambda, t_{\text{MT}} + O(C_h(\lambda) \cdot n))$. The result then follows from Lemma 3.11. $\square$

Efficiency of Merkle commitments. Consider a Merkle commitment scheme $\text{MT}(\mathcal{H})$ and let $k \leftarrow \text{MT.Setup}(\lambda)$. Let $(\text{cm}, \text{aux}) \leftarrow \text{MT.Commit}(k, \mathbf{x})$ for some $\mathbf{x} \in (\{0, 1\}^{\ell(\lambda)})^n$. Let $H \leftarrow \text{MT.Open}(k, \mathbf{x}, S, \text{aux})$ for some non-empty $S \subseteq [n]_0$. Then $|H| + |S| = O(|S| \cdot \log(n/|S| + 1))$. Consequently, MT.Open and MT.Verify each run in time

$$O(T_h(\lambda) \cdot |S| \log(n/|S| + 1)) .$$

The arithmetic circuits for MT.Commit and MT.Verify , which will be used inside the relations in Section 4, have sizes

$$O(C_h(\lambda) \cdot n) \quad \text{and} \quad O(C_h(\lambda) \cdot |S| \log(n/|S| + 1)),$$

respectively.

Lemma 3.16 (Updatability of Merkle commitments). *The Merkle commitment scheme $\text{MT}(\mathcal{H})$ is commitment-updatable in time*

$$T_{\text{cupd}}^{\text{vc}}(\lambda, n, u) = O(T_h(\lambda) \cdot u \log(n/u + 1)) = O(u \log(n/u + 1)) ,$$

where $T_h(\lambda)$ is the time to evaluate $h_\lambda(k, \cdot)$ and u is the size of the update set. Furthermore, $\text{MT}(\mathcal{H})$ is proof-updatable in time

$$T_{\text{pupd}}^{\text{vc}}(\lambda, n, u, q) = O(u \log(n/u + 1)) ,$$

where u is the size of the update set and q is the size of the query set. In particular, the proof-update time is independent of the q .

Proof. Let U be the update set, and let $\mathcal{A}(U)$ be the union of the paths from the leaves indexed by U to the root. Thus, $|\mathcal{A}(U)| = O(u \log(n/u + 1))$.

- **MT.UpdateCommit:** Replace the leaf hashes indexed by U and recompute the remaining nodes in $\mathcal{A}(U)$ from the bottom up. Every recomputed node requires one evaluation of $h_\lambda(k, \cdot)$.
- **MT.UpdateProof** for an arbitrary query set P : Construct H' by retaining every entry of H whose node is outside $\mathcal{A}(U)$ and, for every node $v \in \mathcal{A}(U)$ that is a key of H , replacing $H[v]$ by the updated hash $\text{aux}'[v]$. Values at queried leaves are separate inputs to MT.Verify and are not part of H , so the same procedure applies even when $U \cap P \neq \emptyset$. A proof node outside $\mathcal{A}(U)$ roots a subtree containing no updated leaf, so its hash is unchanged. Every proof node in $\mathcal{A}(U)$ receives its new hash from aux' . Consequently, $H' = \text{MT.Open}(k, \mathbf{x}', P, \text{aux}')$ as desired. Enumerating $\mathcal{A}(U)$ and performing the corresponding dictionary lookups and persistent writes takes $O(u \log(n/u + 1))$ time. No scan of P or of the unchanged portion of H is required.

$\square$

4 Dynamic SNARK compiler

We first state the main compiler theorem. The remainder of this section describes the construction and establishes the claimed bounds.

Theorem 4.1 (Compiler for dynamic SNARKs). *Fix a security parameter λ and an index $\mathcal{C}$ of $\mathcal{R}_{\text{Plonk}}$ with size parameters n, n_0, m as in Definition 3.1, and define $\ell := \lceil \log(m+1) \rceil$. For every level $j \in [\ell]$, define the capacity bound¹¹*

$$q_j := \min\{m, 2^{\ell-j+1}\} . \quad (4)$$

An update has size k if the old and new pairs have Plonkish distance k . Writing their concatenated vectors as $\mathbf{z}$ and $\mathbf{z}'$, the update is defined by the exact difference set $U := \{r \in [m]_0 : \mathbf{z}[r] \neq \mathbf{z}'[r]\}$ and the new values $\{\mathbf{z}'[r]\}_{r \in U}$, together with random access to the old pair. Consider the following two ingredients:

- *A preprocessing SNARK $\text{SNARK} := (\mathbf{G}, \mathbf{I}, \mathbf{P}, \mathbf{V})$ for NP relations. Let*

$$\begin{aligned} \kappa^{\text{arg}}(\lambda, N, t), \quad C_{\mathbf{E}}^{\text{arg}}(\lambda, N, t), \quad T_{\mathbf{G}}^{\text{arg}}(\lambda, N), \quad T_{\mathbf{I}}^{\text{arg}}(\lambda, N), \\ T_{\mathbf{P}}^{\text{arg}}(\lambda, N), \quad T_{\mathbf{V}}^{\text{arg}}(\lambda, N), \quad L^{\text{arg}}(\lambda, N) \end{aligned}$$

denote, respectively, its knowledge error against t -size adversaries, extractor circuit size, setup time, indexer time, prover time, verifier time, and proof length. Here N denotes the CRS capacity in the first three bounds and the indexed circuit size in the remaining bounds.

- *A vector commitment scheme VC with perfect completeness, opening locality, and commitment uniqueness. Let its strong position-binding and proof-uniqueness errors be $\epsilon_{\text{spd}}^{\text{vc}}(\lambda, m, q, t)$ and $\epsilon_{\text{pu}}^{\text{vc}}(\lambda, m, q, t)$ respectively, for vectors of length m , batches of size at most $q \leq m$, and t -size adversaries. Let*

$$T_{\text{setup}}^{\text{vc}}(\lambda, m), \quad T_{\text{com}}^{\text{vc}}(\lambda, m), \quad T_{\text{open}}^{\text{vc}}(\lambda, m, q), \quad T_{\text{ver}}^{\text{vc}}(\lambda, m, q)$$

denote the running time of VC.Setup , VC.Commit , VC.Open , and VC.Verify , respectively, for vectors of length m and query sets of size at most q . Let $T_{\text{cupd}}^{\text{vc}}(\lambda, m, u)$ and $T_{\text{pupd}}^{\text{vc}}(\lambda, m, u, q)$ be the commitment-update time and the proof-update time for updates of size at most u and query sets of size at most q . Let $C_{\text{com}}^{\text{vc}}(\lambda, m)$ and $C_{\text{ver}}^{\text{vc}}(\lambda, m, q)$ denote the arithmetic-circuit sizes of the VC.Commit and VC.Verify algorithms, respectively. Finally, let $L_{\text{cm}}^{\text{vc}}(\lambda, m)$ and $L_{\text{open}}^{\text{vc}}(\lambda, m, q)$ denote the lengths of a commitment and a batch-opening proof, respectively.

The initial relation InitStmt (see (16)) has $\sigma_{\mathcal{C}}$ hardwired into its circuit during public indexing, so each wiring equality compares two fixed witness coordinates and uses $O(1)$ arithmetic constraints. The arithmetic-circuit sizes of InitStmt and the update relation UpdateStmt (see (17)) therefore satisfy

$$N_{\text{init}} := O(n + C_{\text{com}}^{\text{vc}}(\lambda, m)) \quad \text{and} \quad N_{\text{upd}}(q) := O(q \log m + C_{\text{ver}}^{\text{vc}}(\lambda, m, q))$$

¹¹This is a bound on the update size contained on the j -th level of the hierarchical data structure discussed in Fig. 4. However, for the deamortized construction and analysis in Appendix B, this bound is $q_j := \min\{m, 3 \cdot 2^{\ell-j}\}$.

Let $N_* := \max\{N_{\text{init}}, N_{\text{upd}}(q_1)\}$ be the static CRS capacity. There exists a compiler that, given SNARK and VC, outputs a preprocessing dynamic SNARK for $\mathcal{R}_{\text{Plonk}}$:

$$\text{DYN}[\text{SNARK}, \text{VC}] := (\mathcal{G}, \mathcal{I}, \mathcal{P}_{\text{init}}, \mathcal{P}_{\text{update}}, \mathcal{V})$$

with the following properties:

- **Setup time.** $\mathcal{G}$ runs in time

$$T_{\mathcal{G}}(\lambda, n, n_0) \leq T_{\text{setup}}^{\text{vc}}(\lambda, m) + T_{\mathbf{G}}^{\text{arg}}(\lambda, N_*) . \quad (5)$$

- **Indexer time.** $\mathcal{I}$ runs in time

$$T_{\mathcal{I}}(\lambda, n, n_0) \leq O(n) + T_{\text{com}}^{\text{vc}}(\lambda, m) + T_{\mathbf{I}}^{\text{arg}}(\lambda, N_{\text{init}}) + \sum_{j=1}^{\ell} T_{\mathbf{I}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_j)) . \quad (6)$$

- **Initial prover time.** For the initial computation, $\mathcal{P}_{\text{init}}$ runs in time

$$T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0) \leq T_{\mathbf{P}}^{\text{arg}}(\lambda, N_{\text{init}}) + T_{\text{com}}^{\text{vc}}(\lambda, m) + T_{\text{open}}^{\text{vc}}(\lambda, m, n_0) . \quad (7)$$

- **Update prover time.** Define

$$\tau_{\text{upd}}(\lambda, m, n_0) := \max_{j \in [\ell]} \frac{T_{\text{pupd}}^{\text{vc}}(\lambda, m, q_j, n_0) + T_{\mathbf{P}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_j)) + T_{\text{cupd}}^{\text{vc}}(\lambda, m, q_j) + 3T_{\text{open}}^{\text{vc}}(\lambda, m, q_j) + O(q_j)}{2^{\ell-j}} . \quad (8)$$

An update of size $k \in [m]$ has update-prover time

$$\begin{aligned} T_{\mathcal{P}_{\text{update}}}(\lambda, n, n_0, k) &= O((k/m) \cdot T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0)) \\ &\quad + O(k \cdot \tau_{\text{upd}}(\lambda, m, n_0) \cdot \log(m/k + 1)) . \end{aligned} \quad (9)$$

In particular, if $\tau_{\text{upd}}(\lambda, m, n_0) = O(\log m)$ and $T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0) = O(m \log m)$, then the update-prover time is

$$O(k \cdot \log m \cdot \log(m/k + 1)) .$$

Up to factors depending only on λ , this condition holds for Merkle commitments when the static SNARK prover runs in time linear in the size of the corresponding relation.

- **Verifier time.** The verifier runs in time

$$T_{\mathcal{V}}(\lambda, n, n_0) \leq T_{\mathbf{V}}^{\text{arg}}(\lambda, N_{\text{init}}) + \sum_{j=1}^{\ell} T_{\mathbf{V}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_j)) + T_{\text{ver}}^{\text{vc}}(\lambda, m, n_0) . \quad (10)$$

- **Proof length.** The proof length is bounded by

$$\begin{aligned} L(\lambda, n, n_0) &\leq L^{\text{arg}}(\lambda, N_{\text{init}}) + \sum_{j=1}^{\ell} L^{\text{arg}}(\lambda, N_{\text{upd}}(q_j)) \\ &\quad + (\ell + 1)L_{\text{cm}}^{\text{vc}}(\lambda, m) + L_{\text{open}}^{\text{vc}}(\lambda, m, n_0) . \end{aligned} \quad (11)$$

- **Knowledge error and extractor size.** For every classical t -size dynamic adversary, let $t_0, \dots, t_\ell$ denote the sizes of the adversaries induced against the static SNARK by the dynamic extractor. Let $\epsilon_{\text{spd}}^{\text{vc}}$ upper-bound the strong position-binding error for all batch sizes and induced adversary sizes used by the extractor, and let $\epsilon_{\text{pu}}^{\text{vc}}$ upper-bound the proof-uniqueness error at the enlarged adversary sizes required by Lemma 3.11. The compiled dynamic SNARK has knowledge error at most

$$\kappa^{\text{arg}}(\lambda, N_*, t_0) + \sum_{j=1}^{\ell} \kappa^{\text{arg}}(\lambda, N_*, t_j) + (3\ell + 2)\epsilon_{\text{spd}}^{\text{vc}} + \ell\epsilon_{\text{pu}}^{\text{vc}} . \quad (12)$$

Its extractor has circuit size at most

$$O \left(C_{\mathbf{E}}^{\text{arg}}(\lambda, N_*, t_0) + \sum_{j=1}^{\ell} C_{\mathbf{E}}^{\text{arg}}(\lambda, N_*, t_j) \right) . \quad (13)$$

Suppose further that $m, N_* = \text{poly}(\lambda)$ and the static SNARKs for the initial and update relations satisfy post-quantum straightline knowledge soundness in the sense of [CDHZ26, Definition 10.3], with quantum polynomial-time extractors, a common efficiently implementable emulated oracle, and negligible simulation, commutativity, and extraction errors. If the vector commitment is strongly position binding and proof unique against quantum polynomial-time adversaries with negligible errors, then the compiled dynamic SNARK has negligible knowledge error in the QROM, with a quantum polynomial-time straightline extractor.

The following subsections are organized as follows. In Section 4.1, we describe setup and indexing and prove Eq. (6). In Section 4.2, we give an overview of the dynamic SNARK construction and motivate it from scratch. In Section 4.3, we give the full construction of the dynamic SNARK. In Section 4.4, we sketch its efficiency analysis; the full amortized efficiency and knowledge soundness proofs appear in Appendix A. Finally, Section 4.5, we instantiate the compiler with a Merkle commitments, and prove a special corollary of Theorem 4.1.

4.1 Setup and indexing

Fix the index $\mathcal{C}$ and its wiring permutation $\sigma_{\mathcal{C}}$ as in Definition 3.1. We use the parameters m, ℓ , and $(q_j)_{j \in [\ell]}$ fixed in Theorem 4.1.

Setup and indexer computation. The setup algorithm $\mathcal{G}$ computes

$$\text{pp} \leftarrow \text{VC.Setup}(1^\lambda, m) \quad \text{and} \quad \text{crs} \leftarrow \mathbf{G}(1^\lambda, N_*)$$

and outputs the public parameters (crs, pp) . The setup-time bound in (5) follows immediately.

The index of the compiled preprocessing SNARK is $\mathfrak{i} = \mathcal{C}$. On input $\mathcal{C}$ and the public parameters (crs, pp) , the indexer $\mathcal{I}$ computes the wiring permutation $\sigma_{\mathcal{C}}$. It then commits to the permutation:

$$(\text{cm}_{\text{perm}}, \text{aux}_{\text{perm}}) \leftarrow \text{VC.Commit}(\text{pp}, \sigma_{\mathcal{C}}) .$$

It invokes the underlying static indexer on the arithmetic circuit for the initial relation in (16) and on the circuit for each update relation in (17), padding the level- j update relation to capacity q_j , and obtaining

$$\begin{aligned} (\text{pk}_0, \text{vk}_0) &\leftarrow \mathbf{I}(\text{crs}, \text{InitStmt}), \\ (\text{pk}_j, \text{vk}_j) &\leftarrow \mathbf{I}(\text{crs}, \text{UpdateStmt}_j) \quad \text{for each } j \in [\ell]. \end{aligned}$$

Finally, $\mathcal{I}$ outputs

$$\text{pk} := ((\text{pk}_j)_{j \in [0, \ell]}, \sigma_{\mathcal{C}}, \text{pp}, \text{aux}_{\text{perm}}) \quad \text{and} \quad \text{vk} := ((\text{vk}_j)_{j \in [0, \ell]}, \text{pp}, \text{cm}_{\text{perm}}) .$$

The indexer-time bound in (6) follows immediately by summing the time to compute $\sigma_{\mathcal{C}}$, commit to it, and index these $\ell + 1$ relations.

4.2 Dynamic SNARK construction overview

Let $\mathcal{P}_{\text{init}}$, $\mathcal{P}_{\text{update}}$, and $\mathcal{V}$ be the initial prover, update prover, and verifier, respectively, and fix the index $i = \mathcal{C}$ and the proving and verification keys output by $\mathcal{I}$. We proceed in three stages:

1. We describe $\mathcal{P}_{\text{init}}$ for the initial computation.
2. We describe the first update, specified by an index set $U \subseteq [m]_0$ and replacement values $\{\beta_i\}_{i \in U}$.
3. We extend the update algorithm to multiple updates by maintaining a hierarchical data structure in the prover's auxiliary state.

Initial computation. Let $(\mathbf{x}_0, \mathbf{w}_0)$ be the initial instance-witness pair in $\mathcal{R}_{\text{Plonk}}(\mathcal{C})$. Define $\mathbf{z}_0 := [\mathbf{w}_0, \mathbf{x}_0]$. $\mathcal{P}_{\text{init}}$ computes

$$(\text{cm}_0, \text{aux}_0) \leftarrow \text{VC.Commit}(\text{pp}, \mathbf{z}_0) .$$

$\mathcal{P}_{\text{init}}$ then invokes the static SNARK prover $\mathbf{P}_0$ with respect to the instance $(\text{cm}_{\text{perm}}, \text{cm}_0)$ and witness $\mathbf{z}_0$ for the following constraints:

1. $\mathbf{z}_0$ satisfies the gate constraints of $\mathcal{C}$.
2. $\mathbf{z}_0$ satisfies the wiring constraints of $\mathcal{C}$ with respect to the fixed permutation $\sigma_{\mathcal{C}}$.
3. The commitment of $\mathbf{z}_0$ is cm_0 and the commitment of $\sigma_{\mathcal{C}}$ is cm_{perm} .

The precise statement proven by $\mathbf{P}_0$ is given in (16), and it returns a proof π_0 . For the public-input positions $P := \{6n, \dots, 6n + n_0 - 1\}$, the prover computes pf_{pub} , the opening of $\mathbf{z}_0$ at P .¹²

Now, suppose $\mathcal{V}$ receives $\mathbf{x}_0$ and the proof object $(\text{cm}_0, \pi_0, \text{pf}_{\text{pub}})$. $\mathcal{V}$ invokes the static SNARK verifier $\mathbf{V}_0$ on $((\text{cm}_{\text{perm}}, \text{cm}_0), \pi_0)$ to verify that the vector underlying cm_0 satisfies the gate and wiring constraints of $\mathcal{C}$. While this is sufficient to ensure that $\mathcal{P}_{\text{init}}$ possesses some valid witness, it does not guarantee that the witness is consistent with the public input $\mathbf{x}_0$. To ensure this consistency, $\mathcal{V}$ also checks that pf_{pub} is a valid opening of $\mathbf{x}_0$ at P with respect to the commitment cm_0 .

First update. The update prover receives the old pair $(\mathbf{x}_0, \mathbf{w}_0)$ as a random-access object, together with a sparse update $(U, \{\beta_i\}_{i \in U})$ consisting of an index set $U \subseteq [m]_0$ and new values $\{\beta_i\}_{i \in U}$. For the construction, it closes U under gate tuples: if U contains any one of $6i, 6i + 1, 6i + 2$ (respectively, $6i + 3, 6i + 4, 6i + 5$), it adds all three indices, using the old value $\mathbf{z}_0[j]$ as β_j at every newly added position j . We call the resulting set a *valid update index set* and continue to denote it by U . This closure preserves the target pair and increases the update size, and hence $\mathcal{P}_{\text{update}}$'s work, by at most a factor of three. We define $\mathbf{z}_1$ as follows:

$$\mathbf{z}_1[i] := \begin{cases} \beta_i & \text{if } i \in U, \\ \mathbf{z}_0[i] & \text{if } i \notin U \end{cases} ,$$

¹²For readability, we assume throughout that $n_0 > 0$. If $n_0 = 0$, all public-input openings and checks are omitted.

and define $\mathbf{w}_1 := \mathbf{z}_1[0, \dots, 6n - 1]$ and $\mathbf{x}_1 := \mathbf{z}_1[6n, \dots, 6n + n_0 - 1]$. Additionally, for each $i \in U$, let $\alpha_i := \mathbf{z}_0[i]$. To prove that $(\mathbf{x}_1, \mathbf{w}_1) \in \mathcal{R}_{\text{Plonk}}(\mathcal{C})$, $\mathcal{P}_{\text{update}}$ invokes the static SNARK prover $\mathbf{P}_1$ for the instance $(\mathbf{cm}_{\text{perm}}, \mathbf{cm}_0, \mathbf{cm}_1)$, which produces a proof π_1 for the following constraints:

1. The gate constraints of $\mathcal{C}$ are satisfied at all gates touched by the claimed update index set U with respect to the claimed new values $\{\beta_i\}_{i \in U}$.
2. The wiring constraints of $\mathcal{C}$ are satisfied at all positions in U with respect to the claimed new values $\{\beta_i\}_{i \in U}$ and some claimed wiring $\{\sigma_i\}_{i \in U}$.
3. There exists an opening proof that validates the claimed new values $\{\beta_i\}_{i \in U}$ at the indices U with respect to the new commitment $\mathbf{cm}_1$, and also validates the claimed old values $\{\alpha_i\}_{i \in U}$ at the indices U with respect to the old commitment $\mathbf{cm}_0$.
4. There exists an opening proof that validates the claimed wiring $\{\sigma_i\}_{i \in U}$ at the indices U with respect to the commitment $\mathbf{cm}_{\text{perm}}$.

The first two constraints prove that $\mathbf{P}_1$ possesses some $(U, \{\beta_i\}_{i \in U}, \{\sigma_i\}_{i \in U})$ that satisfies the gate and wiring constraints of $\mathcal{C}$ within U . Together with validity of $\mathbf{cm}_0$ and strong local updatability of $\mathbf{VC}$, the third constraint ensures that the vector underlying $\mathbf{cm}_1$ is obtained from the vector underlying $\mathbf{cm}_0$ by changing only the entries in U , and nowhere else. The fourth constraint ensures that the claimed wiring $\{\sigma_i\}_{i \in U}$ is consistent with the true permutation $\sigma_{\mathcal{C}}$. Note that the size of this statement is only a function of $|U|$, and the size of the arithmetic circuit that implements $\mathbf{VC.Verify}$ for a query set of size $|U|$. The precise statement proven by $\mathbf{P}_1$ is given in (17). Finally, $\mathcal{P}_{\text{update}}$ computes $\mathbf{pf}'_{\text{pub}}$, the opening of $\mathbf{z}_1$ at P . This opening can be updated efficiently using $\mathcal{V}.\text{UpdateProof}$.

Now, suppose $\mathcal{V}$ receives $\mathbf{x}_1$ and the proof object

$$(\mathbf{cm}_0, \mathbf{cm}_1, \pi_0, \pi_1, \mathbf{pf}'_{\text{pub}}) .$$

$\mathcal{V}$ invokes $\mathbf{V}_0$ on $((\mathbf{cm}_{\text{perm}}, \mathbf{cm}_0), \pi_0)$ to ensure that there exists a valid witness $\mathbf{z}_0$ underlying $\mathbf{cm}_0$. Then, it invokes $\mathbf{V}_1$ on $((\mathbf{cm}_{\text{perm}}, \mathbf{cm}_0, \mathbf{cm}_1), \pi_1)$ to ensure that there exists a valid update $(U, \{\beta_i\}_{i \in U})$ that transforms $\mathbf{z}_0$ into a new valid witness $\mathbf{z}_1$ underlying $\mathbf{cm}_1$. Finally, $\mathcal{V}$ checks that $\mathbf{pf}'_{\text{pub}}$ is a valid opening of $\mathbf{x}_1$ at P with respect to the commitment $\mathbf{cm}_1$.

Auxiliary state for multiple updates. To handle multiple updates, $\mathcal{P}_{\text{update}}$ maintains an auxiliary state $\mathbb{M}$ containing $\ell = \lceil \log(m + 1) \rceil$ update witness tables $\mathbf{u}_1, \dots, \mathbf{u}_\ell$. For each $j \in [\ell]$, $\mathbf{u}_j$ is either $\perp$ or corresponds to an update index set U_j of size $2^{\ell-j} \leq |U_j| < 2^{\ell-j+1}$ and hence $|U_j| \leq q_j$, and consists of the following information:

$$\mathbf{u}_j = \{(i, \alpha_i, \beta_i, \sigma_i)\}_{i \in U_j} , \tag{14}$$

where α_i is the purported old value at index i , β_i is the purported new value at index i , and σ_i is the purported permutation value at index i . For each $j \in [\ell]$, we define

$$\mathbf{z}_j[i] := \begin{cases} \beta_i & \text{if } \mathbf{u}_j \neq \perp \text{ and } i \in U_j, \\ \mathbf{z}_{j-1}[i] & \text{otherwise} \end{cases} . \tag{15}$$

Thus, the application of $\mathbf{u}_j$ to $\mathbf{z}_{j-1}$ yields $\mathbf{z}_j$.

In the discussion for the first update, note that the proof object expanded to include a commitment cm_1 to the updated vector $\mathbf{z}_1$ and a transition proof π_1 from cm_0 to cm_1 . Generalizing this approach, we can handle multiple updates by maintaining a sequence of commitments $\text{cm}_0, \text{cm}_1, \dots, \text{cm}_\ell$ to the vectors $\mathbf{z}_0, \mathbf{z}_1, \dots, \mathbf{z}_\ell$, respectively. The prover retains the corresponding commitment auxiliary information $\text{aux}_0, \text{aux}_1, \dots, \text{aux}_\ell$ in $\mathbb{M}$. If $\mathbf{u}_j = \perp$, then $(\text{cm}_j, \text{aux}_j) = (\text{cm}_{j-1}, \text{aux}_{j-1})$. Additionally, for each $j \in [\ell]$, $\mathcal{P}_{\text{update}}$ invokes $\mathbf{P}_j$ to generate a transition proof π_j from cm_{j-1} to cm_j . More specifically, π_j is generated for the capacity- q_j update relation UpdateStmt_j defined in (17), which captures the aforementioned four constraints.

Handling multiple updates. When $\mathcal{P}_{\text{update}}$ receives the old pair and a sparse update $(U, \{\beta_i\}_{i \in U})$, it first performs the gate-tuple closure above and then merges the affected update witness tables into a new table $\mathbf{u}_t$ for some $t \in [\ell]$. If all update tables are nonempty, the prover applies the update to the current vector, reruns the initialization algorithm on the resulting instance-witness pair, and empties all update tables. The table merging itself takes amortized time $O(|U| \cdot \log n)$. The prover then invokes $\mathbf{P}_t$ for UpdateStmt_t to generate a new transition proof π_t and replaces the affected tables and transition proof components. The detailed update rule appears in Fig. 4.

4.3 Full construction

We present the full construction of the dynamic SNARK $\text{DYN}[\text{SNARK}, \text{VC}]$ in this subsection. Fig. 3 describes the prover algorithm for generating a proof for the initial instance-witness pair. Fig. 4 describes the prover algorithm for generating a proof for an arbitrary update. Fig. 5 describes the verifier algorithm for verifying a proof after an arbitrary (possibly empty) sequence of updates. All algorithms use random-access dictionaries with constant-time access. This includes the auxiliary state and proof objects, as well as in the working memory of the algorithms.

Input: $(\text{pk}, (\mathbf{x}_0, \mathbf{w}_0))$, where pk is the proving key:

$$\text{pk} = ((\text{pk}_j)_{j \in [0, \ell]}, \sigma_{\mathcal{C}}, \text{pp}, \text{aux}_{\text{perm}});$$

and $(\mathbf{x}_0, \mathbf{w}_0)$ is an instance-witness pair such that $(\mathbf{x}_0, \mathbf{w}_0) \in \mathcal{R}_{\text{Plonk}}(\mathcal{C})$.

1. $\mathcal{P}_{\text{init}}$ computes

$$\begin{aligned} \mathbf{z}_0 &:= [\mathbf{w}_0, \mathbf{x}_0], \\ (\text{cm}_0, \text{aux}_0) &:= \text{VC.Commit}(\text{pp}, \mathbf{z}_0). \end{aligned}$$

2. Define the *initial relation* InitStmt , with $\sigma_{\mathcal{C}}$ hardwired into its circuit during indexing:

$$\text{InitStmt} := \left\{ \begin{array}{l} ((\text{cm}_{\text{perm}}, \text{cm}_0), (\mathbf{x}_0, \mathbf{w}_0)) : \\ \mathbf{z}_0 = [\mathbf{w}_0, \mathbf{x}_0] \\ \text{and } \forall i \in [n]_0 : \mathbf{z}_0[6i] + \mathbf{z}_0[6i+1] = \mathbf{z}_0[6i+2] \\ \text{and } \forall i \in [n]_0 : \mathbf{z}_0[6i+3] \cdot \mathbf{z}_0[6i+4] = \mathbf{z}_0[6i+5] \\ \text{and } \forall i \in [m]_0 : \mathbf{z}_0[i] = \mathbf{z}_0[\sigma_{\mathcal{C}}[i]] \\ \text{and } \text{cm}_0 = \text{Com}_{\text{pp}}(\mathbf{z}_0) \\ \text{and } \text{cm}_{\text{perm}} = \text{Com}_{\text{pp}}(\sigma_{\mathcal{C}}) \end{array} \right\}. \quad (16)$$

3. Using witness $(\mathbf{x}_0, \mathbf{w}_0)$, $\mathcal{P}_{\text{init}}$ invokes $\mathbf{P}_0$ to prove the public instance $(\mathbf{cm}_{\text{perm}}, \mathbf{cm}_0)$ for the relation InitStmt . Denote the resulting initial proof by π_0 .

4. For each $s \in [\ell]$, initialize

$$\mathbf{cm}_s := \mathbf{cm}_0, \quad \mathbf{aux}_s := \mathbf{aux}_0, \quad \pi_s := \perp.$$

Let $P := \{6n, \dots, 6n + n_0 - 1\}$ denote the indices for the public input. Compute

$$\mathbf{pf}_{\text{pub}} := \text{VC.Open}(\mathbf{pp}, \mathbf{z}_0, P, \mathbf{aux}_0) .$$

5. $\mathcal{P}_{\text{init}}$ outputs:

- the current proof object Π_0 ,

$$\Pi_0 := \left((\mathbf{cm}_s, \pi_s)_{s \in [0, \ell]}, \mathbf{pf}_{\text{pub}} \right) ,$$

- the auxiliary state

$$\mathbb{M} := \left(\mathbf{x}_0, \mathbf{w}_0, (\mathbf{aux}_s)_{s \in [0, \ell]}, (\mathbf{u}_s)_{s \in [\ell]}, \Pi_0 \right),$$

where $\mathbf{u}_s = \perp$ for each $s \in [\ell]$.

Figure 3: Prover algorithm for initial computation.

Input: $(\mathbf{pk}, \mathbb{M}, (\mathbf{x}, \mathbf{w}), U^*, \{\beta_i\}_{i \in U^*})$, where $\mathbf{pk}$ is the proving key:

$$\mathbf{pk} = ((\mathbf{pk}_j)_{j \in [0, \ell]}, \sigma_C, \mathbf{pp}, \mathbf{aux}_{\text{perm}}) ,$$

$\mathbb{M}$ is the current auxiliary state:

$$\mathbb{M} := \left(\mathbf{x}_0, \mathbf{w}_0, (\mathbf{aux}_j)_{j \in [0, \ell]}, (\mathbf{u}_j)_{j \in [\ell]}, \Pi := \left((\mathbf{cm}_j, \pi_j)_{j \in [0, \ell]}, \mathbf{pf}_{\text{pub}} \right) \right) ,$$

$(\mathbf{x}, \mathbf{w})$ is the current old instance-witness pair, available as a random-access object and satisfying $[\mathbf{w}, \mathbf{x}] = \mathbf{z}_\ell$, and $(U^*, \{\beta_i\}_{i \in U^*})$ is an update such that the resulting instance-witness pair is in $\mathcal{R}_{\text{Plonk}}(\mathcal{C})$.

1. First, if $U^* = \emptyset$ then return the existing proof object Π and the existing auxiliary state $\mathbb{M}$. Otherwise, let U be the gate-tuple closure of U^* , and for every $i \in U \setminus U^*$, set $\beta_i := \mathbf{z}_\ell[i]$.
2. If every update table is non- $\perp$, $\mathcal{P}_{\text{update}}$ applies the incoming update to $\mathbf{z}_\ell$, invokes $\mathcal{P}_{\text{init}}$ in Fig. 3 on the resulting instance-witness pair, outputs the resulting proof object and auxiliary state, and halts. Otherwise, initialize $\tilde{U} := U$ and $\tilde{\mathbf{u}} := \{(i, \alpha_i, \beta_i, \sigma_C[i]) : i \in U\}$ where $\alpha_i = \mathbf{z}_\ell[i]$ and β_i is the new value at position i . Loop until termination:

(a) Let $t \in [\ell]$ be such that $2^{\ell-t} \leq |\tilde{U}| < 2^{\ell-t+1}$, and let

$$J := \{j \in [t, \ell] : \mathbf{u}_j \neq \perp\} .$$

- (b) If $J = \emptyset$, then set $U_t \leftarrow \tilde{U}$, $\mathfrak{u}_t \leftarrow \tilde{\mathfrak{u}}$, and terminate the loop.
- (c) Otherwise, compose, in increasing order, the updates in $(\mathfrak{u}_j)_{j \in J}$ followed by $\tilde{\mathfrak{u}}$. Let $\tilde{\mathfrak{u}}$ be the resulting update, let $\tilde{U}$ be its update index set, set $\mathfrak{u}_j := \perp$ for every $j \in J$ and continue the loop.

Define $\sigma_i := \sigma_C[i]$ for each $i \in U_t$.

3. $\mathcal{P}_{\text{update}}$ updates the commitment by invoking

$$(\text{cm}_t, \text{aux}_t) \leftarrow \text{VC.UpdateCommit}(\text{pp}, U_t, \{\beta_i\}_{i \in U_t}, \text{cm}_{t-1}, \text{aux}_{t-1}) .$$

4. Compute the shared update opening

$$\text{pf}_{\text{upd},t} := \text{VC.Open}(\text{pp}, \mathbf{z}_{t-1}, U_t, \text{aux}_{t-1}) ,$$

which certifies $\{\alpha_i\}_{i \in U_t}$ under cm_{t-1} and $\{\beta_i\}_{i \in U_t}$ under cm_t .

5. Compute the permutation opening

$$\text{pf}_{\text{perm},t} := \text{VC.Open}(\text{pp}, \sigma_C, U_t, \text{aux}_{\text{perm}}) .$$

6. Let $V_t := \{\sigma_i : i \in U_t\}$. (The values σ_i are distinct because σ_C is a permutation.) Compute the wiring-neighbor opening

$$\text{pf}_{\text{wire},t} := \text{VC.Open}(\text{pp}, \mathbf{z}_t, V_t, \text{aux}_t) ,$$

which certifies the index-value pairs $\{\sigma_i : \beta_i\}_{i \in U_t}$ under cm_t .

7. $\mathcal{P}_{\text{update}}$ invokes $\mathbf{P}_t$ to prove UpdateStmt_t , defined as follows:

$$\text{UpdateStmt}_t := \left\{ \begin{array}{l} (\xi_t, \omega_t) : \\ U_t \text{ is a valid update index set} \\ \text{and } \forall i \in A(U_t) : \beta_{6i} + \beta_{6i+1} = \beta_{6i+2} \\ \text{and } \forall i \in M(U_t) : \beta_{6i+3} \cdot \beta_{6i+4} = \beta_{6i+5} \\ \text{and } \text{VC.Verify}(\text{pp}, \text{cm}_{t-1}, U_t, \{\alpha_i\}_{i \in U_t}, \text{pf}_{\text{upd},t}) = 1 \\ \text{and } \text{VC.Verify}(\text{pp}, \text{cm}_t, U_t, \{\beta_i\}_{i \in U_t}, \text{pf}_{\text{upd},t}) = 1 \\ \text{and } \text{VC.Verify}(\text{pp}, \text{cm}_t, V_t, \{\sigma_i : \beta_i\}_{i \in U_t}, \text{pf}_{\text{wire},t}) = 1 \\ \text{and } \text{VC.Verify}(\text{pp}, \text{cm}_{\text{perm}}, U_t, \{\sigma_i\}_{i \in U_t}, \text{pf}_{\text{perm},t}) = 1 \end{array} \right\} , \quad (17)$$

where $1 \leq |U_t| \leq q_t$ by (4), and

$$\begin{aligned} \xi_t &:= (\text{cm}_{\text{perm}}, \text{cm}_{t-1}, \text{cm}_t) , \\ \omega_t &:= (\text{pf}_{\text{upd},t}, \text{pf}_{\text{perm},t}, \text{pf}_{\text{wire},t}, \mathfrak{u}_t = \{(i, \alpha_i, \beta_i, \sigma_i)\}_{i \in U_t}) , \\ A(U) &:= \{\lfloor i/6 \rfloor : i \in U \cap [6n]_0 \text{ and } i \bmod 6 \in \{0, 1, 2\}\} , \\ M(U) &:= \{\lfloor i/6 \rfloor : i \in U \cap [6n]_0 \text{ and } i \bmod 6 \in \{3, 4, 5\}\} , \\ V_t &:= \{\sigma_i : i \in U_t\} . \end{aligned}$$

Denote the resulting transition proof by π_t . For every $j > t$, set

$$\text{cm}_j := \text{cm}_t, \quad \text{aux}_j := \text{aux}_t, \quad \pi_j := \perp .$$

8. Let $P := \{6n, \dots, 6n + n_0 - 1\}$. Update the public-input opening by computing

$$\text{pf}'_{\text{pub}} := \text{VC.UpdateProof}(\text{pp}, \text{pf}_{\text{pub}}, \text{aux}_\ell, U, P) \ .$$

9. $\mathcal{P}_{\text{update}}$ outputs:

- the current proof object Π_{new} :

$$\Pi_{\text{new}} := \left((\text{cm}_j, \pi_j)_{j \in [0, \ell]}, \text{pf}'_{\text{pub}} \right) \ ,$$

- the updated auxiliary state

$$\mathbb{M}_{\text{new}} := \left(\mathbb{x}_0, \mathbb{w}_0, (\text{aux}_j)_{j \in [0, \ell]}, (\mathbb{u}_j)_{j \in [\ell]}, \Pi_{\text{new}} \right).$$

Figure 4: Prover algorithm for update computation.

Input: $(\text{vk}, \Pi, \mathbb{x})$, where vk is the verification key:

$$\text{vk} = ((\text{vk}_j)_{j \in [0, \ell]}, \text{pp}, \text{cm}_{\text{perm}}) \ ,$$

a proof object received from the dynamic prover,

$$\Pi = \left((\text{cm}_j, \pi_j)_{j \in [0, \ell]}, \text{pf}_{\text{pub}} \right) \ ,$$

as well as the current instance $\mathbb{x}$.

1. $\mathcal{V}$ invokes $\mathbf{V}_0$ for the relation `InitStmt` and checks that

$$\mathbf{V}_0((\text{cm}_{\text{perm}}, \text{cm}_0), \pi_0) = 1 \ .$$

2. For each $j = 1, \dots, \ell$ in increasing order:

- If $\pi_j = \perp$, $\mathcal{V}$ checks that $\text{cm}_j = \text{cm}_{j-1}$.
- Otherwise, $\mathcal{V}$ invokes $\mathbf{V}_j$ for the relation `UpdateStmtj` and checks that

$$\mathbf{V}_j((\text{cm}_{\text{perm}}, \text{cm}_{j-1}, \text{cm}_j), \pi_j) = 1 \ .$$

3. Let $P := \{6n, \dots, 6n + n_0 - 1\}$. $\mathcal{V}$ checks that

$$\text{VC.Verify}(\text{pp}, \text{cm}_\ell, P, \mathbb{x}, \text{pf}_{\text{pub}}) = 1 \ .$$

4. Accept if and only if all of the above checks pass.

Figure 5: Verifier algorithm for dynamic SNARK construction.

4.4 Proof sketch of Theorem 4.1

The initial-prover time, verifier time, and proof length follow directly from the construction. For the update time, rebuilding a table at level t costs $O(2^{\ell-t} \cdot \tau_{\text{upd}})$ and occurs only after enough updates have accumulated to reach that level's size threshold. Charging each update of size k at the $O(\log(m/k + 1))$ relevant levels covers these rebuilds. Reinitialization occurs only after at least m total update volume has accumulated, so charging each update a k/m fraction of the initialization cost covers this work. The full amortized efficiency and knowledge soundness proofs appear in Appendices A.1 and A.2, respectively.

4.5 Instantiation with Merkle commitments

Corollary 4.2 (Dynamic SNARKs with Merkle commitments). *Define SNARK, $n, n_0, \lambda, m, \ell, N_*$, and $(q_j)_{j \in [\ell]}$ as in Theorem 4.1, and let $\text{MT}(\mathcal{H})$ be a Merkle commitment scheme based on a collision-resistant hash function $\mathcal{H}$ with output length $\mu(\lambda)$. Then, the dynamic SNARK $\text{DYN}[\text{SNARK}, \text{MT}(\mathcal{H})]$ has the following properties:*

- **Indexer time.** $O(n) + T_{\mathbf{I}}^{\text{arg}}(\lambda, N_{\text{init}}) + \sum_{j=1}^{\ell} T_{\mathbf{I}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_j))$.
- **Initial prover time.** $O(n) + T_{\mathbf{P}}^{\text{arg}}(\lambda, N_{\text{init}})$.
- **Update prover time.** *For an update whose old and new Plonkish pairs are at distance k , the time is*

$$O\left(\frac{k}{m} T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0)\right) + O\left(k \log(n/k + 1) \left(\log n + \max_{j \in [\ell]} \frac{T_{\mathbf{P}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_j))}{2^{\ell-j}}\right)\right).$$

- **Verifier time.** $O(n_0) + T_{\mathbf{V}}^{\text{arg}}(\lambda, N_{\text{init}}) + \sum_{j=1}^{\ell} T_{\mathbf{V}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_j))$.
- **Proof length.** $L^{\text{arg}}(\lambda, N_{\text{init}}) + \sum_{j=1}^{\ell} L^{\text{arg}}(\lambda, N_{\text{upd}}(q_j)) + (\ell + 2)\mu(\lambda)$ bits.
- **Knowledge error.** $\kappa^{\text{arg}}(\lambda, N_*, t_0) + \sum_{j=1}^{\ell} \kappa^{\text{arg}}(\lambda, N_*, t_j) + (4\ell + 2)\epsilon_{\text{cr}}(\lambda, \tilde{t} + O(C_h(\lambda) \cdot n))$ against t -size adversaries, where ϵ_{cr} is the collision-resistance error of $\mathcal{H}$ and $C_h(\lambda)$ is the circuit size of $\mathcal{H}$. Here $\tilde{t}$ bounds the full circuit sizes of the induced VC adversaries, including extraction and the enlargement required by Lemma 3.11. The extractor size is same as (13).

Proof. First, consider the following unbalanced Merkle tree: the overall scheme consists of two Merkle trees, one for the first $6n$ entries of the vector, and one for the last n_0 entries of the vector, corresponding to $\mathbf{w}$ and $\mathbf{x}$ respectively. The commitment is the hash of the concatenation of the roots of these two trees.

The indexer time and initial prover time follow immediately from the fact that a Merkle commitment can be computed in linear time.

For the update time, first consider the value of $\tau_{\text{upd}}(\lambda, m, n_0)$ defined in (8). For Merkle commitments at level j , we have $T_{\text{cupd}}^{\text{vc}}(\lambda, m, q_j) = O(q_j \cdot \log(m/q_j + 1))$ and $T_{\text{open}}^{\text{vc}}(\lambda, m, q_j) = O(q_j \cdot \log(m/q_j + 1))$. Furthermore, given the unbalanced tree structure discussed above, we have

$T_{\text{pupd}}^{\text{vc}}(\lambda, m, q_j, n_0) = O(1)$ since the public-input opening just consists of the root of the Merkle subtree over the first $6n$ entries. Therefore,

$$\tau_{\text{upd}}(\lambda, m, n_0) = O\left(\log m + \max_{j \in [\ell]} \frac{T_{\mathbf{P}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_j))}{2^{\ell-j}}\right),$$

and the update time, including reinitialization, follows from (9).

The verifier time follows from the fact that the Merkle opening for the public input can be verified in time $O(n_0)$, and the proof length follows from the fact that the Merkle opening of the public input is just one hash element of length $\mu(\lambda)$, and each Merkle commitment is also one hash element of length $\mu(\lambda)$.

Finally, for the knowledge error, note that the strong position-binding error $\epsilon_{\text{spd}}^{\text{vc}}$ and proof-uniqueness error $\epsilon_{\text{pu}}^{\text{vc}}$ for Merkle commitments are both bounded by the collision-resistance error $\epsilon_{\text{cr}}(\lambda, \tilde{t} + O(\mathcal{C}_h(\lambda) \cdot n))$ of the hash function $\mathcal{H}$, where $\mathcal{C}_h(\lambda)$ is the circuit size of $\mathcal{H}$ (see Lemma 3.14). $\square$

Corollaries from different choices of the static SNARK. The following corollaries use Merkle commitments and the parameters of Corollary 4.2, with $n = \text{poly}(\lambda)$ and $\rho_k := \log(1 + m/k)$.

	HyperPlonk + TensorSwitch	HyperPlonk + multilinear KZG	Groth16
Indexer time	$n \log n$	$n \log n$	$n \log n$
Initial prover time	n	n	$n \log n$
Update prover time	$k\rho_k \log n$	$k\rho_k \log n$	$k\rho_k \log^2 n$
Verifier time	$n_0 + \log^2 n$	$n_0 + \log^2 n$	$n_0 + \log n$
Proof length (bits)	$\log^2 n$	$\log^2 n$	$\log n$
Knowledge-soundness model	ROM / QROM	AGM+ROM	GGM / AGM

Table 2: Dynamic SNARK instantiations under the assumptions below. Efficiency entries are $O_\lambda(\cdot)$ bounds.

A hash-based static SNARK. The BCS transformation [BCS16] compiles interactive oracle proofs (IOPs) into hash-based SNARKs in the ROM, with security analyses in the ROM [CCH+18] and QROM [CMS19; CDHZ26]. We use the following instantiation as the static SNARK in our compiler.

Lemma 4.3 (Hash-based static SNARK). *For arithmetic circuits of size $N = \text{poly}(\lambda)$ with $n_0 \leq N$ public inputs, there exists a transparent preprocessing SNARK for NP with $O_\lambda(N)$ indexer and prover time, $O_\lambda(n_0 + \log N)$ verifier time, and $O_\lambda(\log N)$ proof size in bits. It has negligible knowledge-soundness error in the ROM and QROM.*

Sketch. We combine the HyperPlonk PIOP [CBBZ23], using bounded-degree gates, with an interactive oracle polynomial commitment scheme (IOPCS) or an IOP of proximity [RRR16; BCG+17]. There are IOPPs and IOPCSs with linear prover time and $O(\lambda)$ query complexity [BMMS26; BFRW26]. We choose the TensorSwitch IOPCS [BFRW26], instantiated with constant-rate, constant-distance, linear-time encodable codes admitting efficient decoding from a constant fraction of errors, and with proximity parameters within that decoding radius.¹³ This decoding condition gives efficient extraction in TensorSwitch’s round-by-round knowledge-soundness analysis.

¹³Spielman’s codes [Spi96] satisfy these properties, and the construction extends to arbitrary fields [Vid15].

The HyperPlonk PIOP is round-by-round knowledge sound, since it reduces the input claim to a constant number of sumcheck claims in constant rounds, and then sumcheck itself is round-by-round knowledge sound [ACFY25]. The proof-composition result of [BFRW26, Lemma A.1] combines this guarantee with TensorSwitch’s round-by-round binding and knowledge soundness. Choosing $|\mathbb{F}| = 2^{\Theta(\lambda)}$ gives an IOP with negligible round-by-round knowledge-soundness error, $O_\lambda(N)$ prover and indexer time, $O_\lambda(n_0 + \log N)$ verifier time, and $O(\lambda)$ queries. By [CDHZ26, Theorems 6.10 and 11.3], round-by-round knowledge soundness of the IOP implies adaptive quantum extraction for the BCS-compiled SNARK. The IOP has $O_\lambda(N)$ total oracle length and $O_\lambda(\log N)$ non-oracle communication. Merkle commitments therefore preserve $O_\lambda(N)$ indexer and prover time, while authenticating the queries adds $O_\lambda(\log N)$ bits and verification work. $\square$

Corollary 4.4 (HyperPlonk with a hash-based PCS). *Instantiate the compiler with the static SNARK of Lemma 4.3. Then the dynamic SNARK achieves the first column of Table 2 and knowledge-soundness error $\text{negl}(\lambda)$ in the ROM. If $\mathcal{H}$ is collision resistant against quantum adversaries, it also achieves knowledge soundness in the QROM with the adaptive quantum extraction guarantee of [CDHZ26].*

Corollary 4.5 (HyperPlonk with multilinear KZG). *Instantiate the static SNARK with the HyperPlonk PIOP [CBBZ23] and the multilinear KZG [KZG10; PST13] PCS analyzed in [Lib24], using Fiat–Shamir in the ROM. Under the powers-of-discrete-logarithm assumption of [Lib24], the dynamic SNARK achieves the second column of Table 2 and knowledge-soundness error $\text{negl}(\lambda)$ in the combined algebraic group model (AGM) and ROM.*

Corollary 4.6 (Groth16). *Replace deterministic indexing in Definition 3.2 by a trusted setup for each fixed compiled relation, and instantiate the static SNARKs with Groth16 [Gro16] over a pairing-friendly field. Then the dynamic SNARK achieves the third column of Table 2 and knowledge-soundness error $\text{negl}(\lambda)$ in the generic bilinear group model, or in the AGM under the q -DLOG assumption [FKL18].*

Proof of the three corollaries. Use Corollary 4.2 and Appendix B with $N_{\text{init}} = O_\lambda(n)$, $N_{\text{upd}}(q_j) = O_\lambda(q_j \log n)$, and $\sum_j q_j = O(n)$. Static preprocessing takes $O_\lambda(N)$ time in all three cases, and proving takes $O_\lambda(N)$ for HyperPlonk and $O_\lambda(N \log N)$ for Groth16. For constant-size root statements, static verification and proof length are respectively $O_\lambda(\log N)$, $O_\lambda(\log N)$, and $O_\lambda(1)$. The cited extraction guarantees and collision resistance of $\mathcal{H}$ give knowledge soundness. $\square$

Remark 4.7 (Linear-time indexing). The indexer time in Table 2 can be reduced to $O_\lambda(n)$ for all three instantiations by omitting the $\lceil \log_2 \log_2 m \rceil$ largest-capacity levels and reinitializing after $\Theta(m/\log m)$ total update volume. An individual update of at size at least $\Theta(m/\log m)$ is handled by immediate reinitialization. This preserves all other stated asymptotic bounds.

5 Concrete instantiation of Dynamic SNARKs

While the Dynamic SNARK construction in Section 4 achieves our headline asymptotic results, it has several limitations in terms of concrete efficiency. Perhaps the most significant concrete overhead is due to *arithmetization*: if we want to use static SNARKs for NP as black-boxes to prove the initial statement `InitStmt` (16) or the update statement `UpdateStmt` (17), then we have to first arithmetize them as R1CS or Plonkish instances (or as an instance of some other NP-complete relation).

In this section, we attempt to minimize this overhead by building a polynomial interactive oracle proof (PIOP) [BFS20; CHM+20] for `InitStmt` and `UpdateStmt`. We then instantiate the polynomial oracles of the PIOP using a polynomial commitment scheme (PCS) for multilinear polynomials. Applying the BCS transform [BCS16] on the resulting interactive argument yields a SNARK for the specific relations.

We use Merkle trees instantiated with the Poseidon2 hash function [GKR+21; GKS23] as the vector commitment in our SNARK compiler. Our evaluated implementation uses the binary-field variant Poseidon2b [GKK+26] and the Ligerito PCS for multilinear polynomials [NA25]. We describe the implementation and concrete parameters in Section 5.4.

5.1 Preliminaries for PIOP

A (public-coin) PIOP is an interactive proof between a prover and a verifier where the prover is allowed to send oracles of polynomials to the verifier in each round. In the i -th round of a PIOP, the prover may send a polynomial oracle and/or a plaintext message to the verifier, which responds with a random challenge $\rho_i \xleftarrow{\$} \{0, 1\}^{r_i}$. In this work, we only consider oracles of multilinear polynomials.

For a v -variate multilinear polynomial $\hat{m} : \mathbb{F}^v \rightarrow \mathbb{F}$, we use $\llbracket \hat{m} \rrbracket$ to denote a polynomial oracle to $\hat{m}$. Given a matrix $\mathbf{M} \in \mathbb{F}^{2^u \times 2^v}$, we define $\hat{M} : \mathbb{F}^{u+v} \rightarrow \mathbb{F}$ as the multilinear extension of $\mathbf{M}$, which is the unique multilinear polynomial such that $\hat{M}(\langle x \rangle, \langle y \rangle) = \mathbf{M}[x][y]$ for each $x \in [2^u]_0$ and $y \in [2^v]_0$, where $\langle x \rangle$ represents the binary representation of x . Throughout this section, binary representations use the least significant bit first, and all logarithms are base two. That is,

$$\hat{M}(\mathbf{r}_1, \mathbf{r}_2) := \sum_{x=0}^{2^u-1} \sum_{y=0}^{2^v-1} \mathbf{M}[x][y] \cdot \text{eq}_u(\langle x \rangle, \mathbf{r}_1) \cdot \text{eq}_v(\langle y \rangle, \mathbf{r}_2) ,$$

where $\mathbf{r}_1 \in \mathbb{F}^u$ and $\mathbf{r}_2 \in \mathbb{F}^v$, and $\text{eq}_j(\mathbf{x}, \cdot)$ is the $\mathbf{x}$ -th multilinear Lagrange basis polynomial for each $\mathbf{x} \in \{0, 1\}^j$.

$$\text{eq}_j(\mathbf{r}, \mathbf{x}) := \prod_{k=0}^{j-1} (r_k x_k + (1 - r_k)(1 - x_k)) \quad (\mathbf{r}, \mathbf{x} \in \mathbb{F}^j) .$$

We will use sumcheck [LFKN92] as a sub-protocol in our PIOPs, which is an interactive protocol that reduces sumcheck claims like “ $\sum_{\mathbf{x} \in \{0,1\}^j} f(\mathbf{x}) = \sigma$ ” to polynomial evaluation claims like “ $f(\mathbf{r}) = y$ ”, for any polynomial f .

Indexed polynomial oracle relations. An indexed polynomial oracle relation $\mathcal{R}(\mathbf{i})$ consists of tuples of the form

$$\left((\xi, \llbracket \hat{f}_1 \rrbracket, \dots, \llbracket \hat{f}_s \rrbracket); (\hat{f}_1, \dots, \hat{f}_s) \right) ,$$

where ξ is a tuple of public field elements, $\mathbf{i}$ is an index specifying the public parameters, and $\hat{f}_i$ are multilinear polynomials known to the prover in full, and available to the verifier in the form of

polynomial oracles. An index may also contain polynomials computed by an honest indexer, with oracle access provided to the verifier. We will abbreviate such a pair as $(\xi; \hat{\mathbf{f}}) \in \mathcal{R}(\mathfrak{i})$. Note that PIOPs may use bounded-degree functions of these multilinear inputs, evaluated through their input oracles without sending additional oracles. We discuss some indexed oracle relations defined in HyperPlonk [CBBZ23].

Indexed oracle relation 5.1 (Zerocheck). Consider an index that specifies v , the arity of the multilinear inputs $\hat{\mathbf{f}}$, and polynomials $g_1, \dots, g_t$ of bounded individual degree. The relation $\mathcal{R}_{\text{zero}}(\mathfrak{i})$ consists of polynomials $\hat{\mathbf{f}}$ such that

$$g_j(\mathbf{x}, \hat{\mathbf{f}}(\mathbf{x})) = 0 \quad \text{for all } \mathbf{x} \in \{0, 1\}^v \text{ and } j \in [t] .$$

Indexed oracle relation 5.2 (Product check). Consider an index that specifies v , the arity of the multilinear inputs $\hat{\mathbf{f}}$, and a polynomial g of bounded individual degree. The relation $\mathcal{R}_{\text{prod}}(\mathfrak{i})$ consists of a scalar $\pi \in \mathbb{F}$ and polynomials $\hat{\mathbf{f}}$ such that

$$\prod_{\mathbf{x} \in \{0, 1\}^v} g(\mathbf{x}, \hat{\mathbf{f}}(\mathbf{x})) = \pi .$$

Indexed oracle relation 5.3 (Multiset equality check). For a selector polynomial s and a tuple of polynomials $\mathbf{f}$ defined over $\{0, 1\}^v$, define

$$\mathbf{M}(s; \mathbf{f}) := \{\{\mathbf{f}(\mathbf{x}) : \mathbf{x} \in \{0, 1\}^v, s(\mathbf{x}) = 1\}\} .$$

Consider an index that specifies two arities v_A, v_B , a tuple width t , and polynomial maps g_A and g_B , each with t coordinates of bounded individual degree. The relation $\mathcal{R}_{\text{mset}}(\mathfrak{i})$ consists of multilinear polynomial tuples $\mathbf{f}_A$ and $\mathbf{f}_B$ and selector polynomials s_A and s_B such that the selectors are binary on their respective domains and

$$\mathbf{M}(s_A; g_A(\mathbf{f}_A)) = \mathbf{M}(s_B; g_B(\mathbf{f}_B)) ,$$

where the two multisets use domains $\{0, 1\}^{v_A}$ and $\{0, 1\}^{v_B}$, respectively, and equality preserves multiplicities. A selector may be supplied as a multilinear input polynomial or as a fixed bounded-degree expression in the supplied multilinear input polynomials, evaluated through their oracles.

We use the zerocheck and product-check PIOPs of HyperPlonk [CBBZ23] to decide Indexed oracle relations 5.1 and 5.2 respectively. The selector polynomials being binary-valued is enforced by zerocheck unless it follows from prior checks. For multiset equality, the verifier samples $\gamma, \eta \xleftarrow{\$} \mathbb{F}$ and compresses a tuple $\mathbf{a} \in \mathbb{F}^t$ as $\text{cmp}_\gamma(\mathbf{a}) = \sum_{j=0}^{t-1} \gamma^j a_j$. Define the fingerprint of $\mathbf{x}$ with respect to $(s, \mathbf{f}, g)$ as

$$e_{s, \mathbf{f}, g}(\mathbf{x}) := 1 - s(\mathbf{x}) + s(\mathbf{x}) \left(\eta - \text{cmp}_\gamma(g(\mathbf{f}(\mathbf{x}))) \right) .$$

By Schwartz-Zippel, multiset equality can be reduced to checking

$$\prod_{\mathbf{x} \in \{0, 1\}^{v_A}} e_{s_A, \mathbf{f}_A, g_A}(\mathbf{x}) = \prod_{\mathbf{x} \in \{0, 1\}^{v_B}} e_{s_B, \mathbf{f}_B, g_B}(\mathbf{x}) ,$$

which incurs error at most $D \cdot \max\{1, t-1\}/|\mathbb{F}|$, where D bounds the number of selected records on both sides. Two product-check PIOPs prove that both products equal the same scalar π .

Poseidon2 hash function. Given a prime field $\mathbb{F}$, the Poseidon2 hash function is $\text{Pos2} : \mathbb{F}^3 \rightarrow \mathbb{F}$, which transforms a 3-input field element tuple via multiple non-linear rounds of transformation. Each round of transformation takes as input 3 $\mathbb{F}$ elements and outputs 3 $\mathbb{F}$ elements. One round either looks like the following:

$$F(a, b, c) := \mathbf{M}_F \cdot \begin{pmatrix} (a + \kappa_0)^5 \\ (b + \kappa_1)^5 \\ (c + \kappa_2)^5 \end{pmatrix} \text{ where } \mathbf{M}_F := \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix},$$

which we refer to as a “full round”, or it looks like the following:

$$P(a, b, c) := \mathbf{M}_P \cdot \begin{pmatrix} (a + \kappa_0)^5 \\ b \\ c \end{pmatrix} \text{ where } \mathbf{M}_P := \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 3 \end{pmatrix},$$

which we refer to as a “partial round”. In the above maps $\kappa_0, \kappa_1, \kappa_2$ are some fixed publicly-known constants, and are different across rounds. The overall Poseidon2 construction $\text{Pos2} : \mathbb{F}^3 \rightarrow \mathbb{F}$ is as follows: (a) apply $\mathbf{M}_F$ to the input, (b) apply 4 full transformation rounds, (c) apply 56 partial transformation rounds, (d) apply 4 full transformation rounds, (e) output the first field element in the resulting 3-tuple.

For Merkle trees instantiated with Poseidon2, we use $\text{Pos2}(v, 0, 1)$ for a leaf containing the value $v \in \mathbb{F}$, and $\text{Pos2}(d_L, d_R, 2)$ for an internal node with child hashes d_L, d_R .

Indexed oracle relation 5.4 (Batch Poseidon2). For an index $(\mathbb{F}, v, \text{Pos2})$, where $v \geq 1$, the relation $\mathcal{R}_{\text{Pos2}}$ contains tuples $(\emptyset; \hat{A}, \hat{B}, \hat{C}, \hat{O})$ of v -variate multilinear polynomials satisfying

$$\hat{O}(\mathbf{x}) = \text{Pos2}(\hat{A}(\mathbf{x}), \hat{B}(\mathbf{x}), \hat{C}(\mathbf{x})) \text{ for every } \mathbf{x} \in \{0, 1\}^v.$$

Looking ahead, we will express the initial (16) and update (17) relations of our dynamic SNARK compiler as indexed oracle relations. Since these relations involve checking `MT.Commit` and `MT.Verify` computations respectively, we first define oracle relations specifically for these computations.

Indexed oracle relation 5.5 (Full Merkle tree). Consider an index $(\mathbb{F}, h, \text{Pos2})$, where $h \geq 1$ is the height of the Merkle tree. The relation $\mathcal{R}_{\text{full}}$ contains tuples of the form $(\text{cm}; \hat{w})$, where $\hat{w}$ is the h -variate multilinear extension of $\mathbf{w} \in \mathbb{F}^{2^h}$, such that $\text{cm} = \text{MT.Commit}(\mathbf{w})$.

Indexed oracle relation 5.6 (Partial Merkle tree). Consider an index $(\mathbb{F}, h, \text{Pos2}, v_L, v_B, v_R, z)$, where $h \geq 1$ is the height of the full Merkle tree, $L_{\max} := 2^{v_L}$, $B := 2^{v_B}$ and $R := 2^{v_R}$ are upper bounds on the number of selected leaves, boundary nodes (defined below), and total nodes in the partial Merkle tree respectively, and $z \in \mathbb{F}^\times$ is a “basis element” that satisfies conditions stated later in Section 5.3.

Let $\mathbf{L} \in \mathbb{F}^{L_{\max} \times (h+2)}$ be a *leaf table* consisting of a selector column s_L , h position-bit columns $p_0, \dots, p_{h-1}$, and a value column ν . This table captures the values and positions of the leaves of the corresponding `MT.Verify` check. Let $\mathbf{B} \in \mathbb{F}^{B \times 4}$ be a *boundary table* consisting of columns (s_B, A_B, L_B, d_B) , representing a selector, address label, level label, and hash respectively. This table captures the hashes and positions of the nodes of the Merkle opening proof of the corresponding `MT.Verify` check. We will refer to these nodes as *boundary nodes*. Write $\hat{L} = (\hat{s}_L, \hat{p}_0, \dots, \hat{p}_{h-1}, \hat{\nu})$ and $\hat{B} = (\hat{s}_B, \hat{A}_B, \hat{L}_B, \hat{d}_B)$ for the tuples of column multilinear extensions, of arities v_L and v_B , respectively. A row is active when its selector is one; padding rows have selector zero.

The relation $\mathcal{R}_{\text{partial}}$ contains tuples of the form $(\text{cm}; \widehat{L}, \widehat{B})$ where selector columns are binary, at least one leaf row is active, active leaf rows have binary position bits and distinct decoded positions, and the following condition holds. There exists a binary tree (not necessarily complete) of at most R nodes rooted at position 1, with every node at depth at most h , whose computed internal nodes have two children, and whose terminal nodes are precisely the leaves and boundary nodes specified by active rows of $\mathbf{L}$ and $\mathbf{B}$. An active leaf record at position $p = \sum_{j=0}^{h-1} 2^j p_j$ occurs at depth h and has hash $\text{Pos2}(\nu, 0, 1)$. An active boundary record at depth d and path $b_1 \cdots b_d$ has labels

$$A_B = z^d + \sum_{j=1}^d b_j z^{d-j}, \quad L_B = z^{h-d}.$$

The hash of every computed internal node must be equal to $\text{Pos2}(d_L, d_R, 2)$, and the computed hash of the root must be equal to cm .

Assignment encoding. Let the circuit have n addition gates and n multiplication gates, let $n_0 \leq n$ be the number of public inputs, and define

$$m := 6n + n_0, \quad N = 2^{\lceil \log(2n) \rceil}, \quad h = \log(4N).$$

Also define the following map:

$$p(i) = \begin{cases} 4\lfloor i/3 \rfloor + (i \bmod 3), & 0 \leq i < 6n, \\ 4(i - 6n) + 3, & 6n \leq i < m. \end{cases} \quad (18)$$

Going forward we use $E(\mathbf{z}) \in \mathbb{F}^{4N}$ to denote the vector such that $E(\mathbf{z})[p(i)] = \mathbf{z}[i]$ for all $i \in [m]_0$, and is equal to zero in other positions. This mapping pads $\mathbf{z}$ to a power of two while adding zeros in specific locations rather than at the end. For integer wire indices, use the same public z as in the partial-tree relation, satisfying its routing conditions for the indexed capacities and such that

$$\iota_{\mathbb{F}}(i) := \sum_{j=0}^{\lceil \log m \rceil - 1} z^j i_j \quad \text{is injective on } [m]_0,$$

where i_j is the j -th digit in the binary representation of i . For vectors of integer indices, E first applies $\iota_{\mathbb{F}}$ to each coordinate. The indexed permutation vector $\mathbf{S} \in \mathbb{F}^{4N}$ stores $\iota_{\mathbb{F}}(\sigma_{\mathcal{C}}[i])$ at position $p(i)$ and zero at unused positions. The honest indexer computes and sends $\text{cm}_{\text{perm}} = \text{MT.Commit}(E(\sigma_{\mathcal{C}}))$.

5.2 PIOPs for initial and update statements

Indexed oracle relation 5.7 (Initial statement). The index of the initial statement consists of the circuit parameters, hash parameters, and the permutation oracle $[\hat{S}]$, where $\hat{S}$ is the multilinear extension of $\mathbf{S}$ as defined above. The relation $\mathcal{R}_{\text{init}}$ consists of tuples of the form $((\text{cm}_{\text{perm}}, \text{cm}_0); \hat{w})$ where $\mathbf{w} := E(\mathbf{z}_0)$ and $\text{cm}_{\text{perm}} = \text{MT.Commit}(E(\sigma_{\mathcal{C}}))$, and the following conditions are satisfied:

$$\begin{aligned} z_0[6i] + z_0[6i+1] &= z_0[6i+2] & (i \in [n]_0), \\ z_0[6i+3] \cdot z_0[6i+4] &= z_0[6i+5] & (i \in [n]_0), \\ z_0[i] &= z_0[\sigma_{\mathcal{C}}[i]] & (i \in [m]_0), \\ (\text{cm}_0; \hat{w}) &\in \mathcal{R}_{\text{full}}(\mathbb{F}, h, \text{Pos2}), \\ (\text{cm}_{\text{perm}}; \hat{S}) &\in \mathcal{R}_{\text{full}}(\mathbb{F}, h, \text{Pos2}). \end{aligned}$$

Note that $(\text{cm}_{\text{perm}}; \hat{S}) \in \mathcal{R}_{\text{full}}(\mathbb{F}, h, \text{Pos2})$ is enforced by the honest indexer, so does not need to be checked by the interactive phase of the PIOP for $\mathcal{R}_{\text{init}}$.

The initial zerocheck includes the padding constraint $\hat{s}_{\text{pad}}(\mathbf{x})\hat{w}(\mathbf{x}) = 0$ on $\{0, 1\}^h$, where the public multilinear selector $\hat{s}_{\text{pad}}$ identifies positions outside $p([m]_0)$, in addition to the gate identities. For the wiring check, let $\hat{\text{id}}$ store $\iota_{\mathbb{F}}(i)$ at position $p(i)$ and zero on padding. The wiring clause is equivalently the multiset relation

$$M(1; (\hat{\text{id}}, \hat{w})) = M(1; (\hat{S}, \hat{w})) .$$

Thus, the only remaining check is $(\text{cm}_0; \hat{w}) \in \mathcal{R}_{\text{full}}(\mathbb{F}, h, \text{Pos2})$, which we discuss in detail in Section 5.3.

Indexed oracle relation 5.8 (Update statement). The index contains the circuit and hash parameters, the position map p , and the update-record and Merkle-table capacities. The relation $\mathcal{R}_{\text{upd}}$ consists of tuples of the form $((\text{cm}_{\text{perm}}, \text{cm}_0, \text{cm}_1); (\hat{U}, \hat{B}_{\text{upd}}, \hat{B}_{\text{perm}}, \hat{B}_{\text{wire}}))$. The update table underlying $\hat{U}$ has a binary selector s_U , zero on padding rows, and its active rows encode $(i, \alpha_i, \beta_i, \sigma_i)$. Across active rows, the indices i are pairwise distinct integers in $[m]_0$, and so are σ_i . Their field-valued copies in $\hat{U}$ equal $\iota_{\mathbb{F}}(i)$ and $\iota_{\mathbb{F}}(\sigma_i)$, enforced by consistency constraints with the bit representations of i and σ_i ; coordinate Booleanity, range, and record constraints apply only to active rows. Let U be the set of active wire indices, and define

$$\begin{aligned} A(U) &= \{ \lfloor i/6 \rfloor : i \in U \cap [6n]_0, i \bmod 6 \in \{0, 1, 2\} \} , \\ M(U) &= \{ \lfloor i/6 \rfloor : i \in U \cap [6n]_0, i \bmod 6 \in \{3, 4, 5\} \} . \end{aligned}$$

Membership in the relation requires that U is non-empty, has at most the indexed capacity, and is closed under the gate tuples, as in Fig. 4, and that $\beta_{6j} + \beta_{6j+1} = \beta_{6j+2}$ for all $j \in A(U)$, and $\beta_{6j+3} \cdot \beta_{6j+4} = \beta_{6j+5}$ for all $j \in M(U)$. Define leaf tables $\hat{L}_\alpha, \hat{L}_\beta, \hat{L}_{\text{perm}}, \hat{L}_{\text{wire}}$ from the same update rows, using s_U as their common selector, with position-value pairs

$$(p(i), \alpha_i) , (p(i), \beta_i) , (p(i), \iota_{\mathbb{F}}(\sigma_i)) , (p(\sigma_i), \beta_i) ,$$

respectively, and the position-bit columns used by $\mathcal{R}_{\text{partial}}$. All leaf-table values and position bits are bound to their update-record coordinates by the specified column restrictions and consistency constraints. Finally, membership requires

$$\begin{aligned} (\text{cm}_0; \hat{L}_\alpha, \hat{B}_{\text{upd}}) &\in \mathcal{R}_{\text{partial}} , \\ (\text{cm}_1; \hat{L}_\beta, \hat{B}_{\text{upd}}) &\in \mathcal{R}_{\text{partial}} , \\ (\text{cm}_1; \hat{L}_{\text{wire}}, \hat{B}_{\text{wire}}) &\in \mathcal{R}_{\text{partial}} , \\ (\text{cm}_{\text{perm}}; \hat{L}_{\text{perm}}, \hat{B}_{\text{perm}}) &\in \mathcal{R}_{\text{partial}} , \end{aligned}$$

with the appropriate indices for each check above.

The prover sends the column oracles of all three boundary tuples before the partial-tree invocations. The first two $\mathcal{R}_{\text{partial}}$ membership checks use the same boundary polynomial, encoding the Merkle verification of the shared opening pf_{upd} of (17). The last two checks encode the Merkle verification of pf_{wire} and pf_{perm} as in (17). As described in Fig. 4, the prover issues proofs for up to $\lceil \log m \rceil$ instances of this update relation, with different capacities and roots. Similar to the checks of Indexed oracle relation 5.7, most of the checks described above reduce to zerocheck and multiset equality check. The remaining checks require building a PIOP for $\mathcal{R}_{\text{partial}}$, which we do in the following section.

5.3 PIOPs for Poseidon and Merkle relations

Proving Poseidon2 hashes. An important ingredient of proving $\text{cm}_0 = \text{MT.Commit}(\mathbf{w})$ is proving many Poseidon2 hash evaluations. Thus, we first discuss a PIOP for proving a batch of 2^v such hash evaluation claims, where $v \geq 1$. At the beginning of the PIOP, the prover possesses four v -variate multilinear polynomials $\hat{A}, \hat{B}, \hat{C}, \hat{O}$, representing the three inputs of Pos2 and the output hash, respectively. The verifier possesses $[[\hat{A}]], [[\hat{B}]], [[\hat{C}]], [[\hat{O}]]$. The claim is

$$\hat{O}(\mathbf{x}) = \text{Pos2}(\hat{A}(\mathbf{x}), \hat{B}(\mathbf{x}), \hat{C}(\mathbf{x})) \quad \text{for every } \mathbf{x} \in \{0, 1\}^v. \quad (19)$$

The PIOP is implemented in a manner similar to the GKR protocol [GKR15]: the implementation of Pos2 can be interpreted as a layered circuit, and we prove the application of each layer using sumcheck. Let $T_0, \dots, T_{63} : \mathbb{F}^3 \rightarrow \mathbb{F}^3$ be the 64 round transformations described in Section 5.1 with the initial multiplication by $\mathbf{M}_F$ absorbed into T_0 . Otherwise, each T_i is either an instance of F or P . Let $\hat{\mathbf{s}}_j$ denote the three multilinear extensions of the states before round j , so that $\hat{\mathbf{s}}_0 = (\hat{A}, \hat{B}, \hat{C})$, and for each $\mathbf{x} \in \{0, 1\}^v$, we have $\hat{\mathbf{s}}_{j+1}(\mathbf{x}) = T_j(\hat{\mathbf{s}}_j(\mathbf{x}))$. Thus, $\hat{O} = \hat{\mathbf{s}}_{64}[0]$. Note that these are polynomials for the prover's use, and the prover doesn't send them as oracles to the verifier. In fact, in this PIOP the prover does not send any polynomial oracles to the verifier, and the oracles of $\hat{A}, \hat{B}, \hat{C}, \hat{O}$ are already possessed by the verifier before the PIOP begins, which we now describe in detail.

The verifier samples $\mathbf{r}_{64} \xleftarrow{\$} \mathbb{F}^v$, and queries $[[\hat{O}]]$ to obtain $\sigma_{64} = \hat{O}(\mathbf{r}_{64})$. Using sumcheck, the prover then reduces the claim

$$\sum_{\mathbf{x} \in \{0, 1\}^v} \text{eq}_v(\mathbf{r}_{64}, \mathbf{x}) \cdot T_{63}(\hat{\mathbf{s}}_{63}(\mathbf{x})) [0] = \sigma_{64} ,$$

to a claim $\text{eq}_v(\mathbf{r}_{64}, \mathbf{r}_{63}) \cdot T_{63}(\hat{\mathbf{s}}_{63}(\mathbf{r}_{63})) [0] = y_{63}$, for some random vector $\mathbf{r}_{63} \in \mathbb{F}^v$ comprising of the sumcheck random challenges. Since the degree in x_k of the sumcheck claim is ≤ 6 , the prover sends a polynomial of degree ≤ 6 in each round, which is specified by its evaluations on $\{0, 1, -1, 2, -2, 1/2, -1/2\}$.

Now the prover needs to prove that $\text{eq}_v(\mathbf{r}_{64}, \mathbf{r}_{63}) \cdot T_{63}(\hat{\mathbf{s}}_{63}(\mathbf{r}_{63})) [0] = y_{63}$. The prover sends $\mathbf{a}_{63} \in \mathbb{F}^3$ to the verifier, which are the purported values of $\hat{\mathbf{s}}_{63}(\mathbf{r}_{63})$. The verifier can then apply T_{63} to $\mathbf{a}_{63}$ to verify the above claim, so now the prover has to prove that

$$\mathbf{a}_{63} = \hat{\mathbf{s}}_{63}(\mathbf{r}_{63}) .$$

The verifier sends a challenge $\alpha_{63} \xleftarrow{\$} \mathbb{F}$ and reduces the above 3 claims into a single one by a random linear combination. Then, we can expand $\hat{\mathbf{s}}_{63}(\mathbf{r}_{63})$ to get the new claim

$$\mathbf{a}_{63} \cdot (1, \alpha_{63}, \alpha_{63}^2) = \sum_{\mathbf{x} \in \{0, 1\}^v} \text{eq}_v(\mathbf{r}_{63}, \mathbf{x}) \cdot T_{62}(\hat{\mathbf{s}}_{62}(\mathbf{x})) \cdot (1, \alpha_{63}, \alpha_{63}^2) .$$

Again, we can apply sumcheck to reduce the above claim to a random evaluation claim about $\hat{\mathbf{s}}_{62}$. We keep going until we reach the claim

$$\mathbf{a}_0 = \hat{\mathbf{s}}_0(\mathbf{r}_0) ,$$

for some random point $\mathbf{r}_0 \in \mathbb{F}^v$. These are simply three claims $\hat{A}(\mathbf{r}_0) = \mathbf{a}_0[0]$, $\hat{B}(\mathbf{r}_0) = \mathbf{a}_0[1]$, and $\hat{C}(\mathbf{r}_0) = \mathbf{a}_0[2]$, which can all be checked by the verifier since it possesses the polynomial oracles for $\hat{A}, \hat{B}, \hat{C}$.

Input: Public parameters $(\mathbb{F}, v, \text{Pos2})$ and four v -variate multilinear polynomials $\hat{A}, \hat{B}, \hat{C}, \hat{O}$ known to $\mathbf{P}$. $\mathbf{V}$ has the oracles $[\hat{A}], [\hat{B}], [\hat{C}], [\hat{O}]$. The claimed polynomial oracle relation is $\mathcal{R}_{\text{Pos2}}(\mathbb{F}, v, \text{Pos2})$.

1. $\mathbf{P}$ computes the intermediate state polynomials $\hat{\mathbf{s}}_0, \dots, \hat{\mathbf{s}}_{64}$ using the round maps $T_0, \dots, T_{63}$ defined above, and does not send their oracles.

2. $\mathbf{V}$ samples $\mathbf{r}_{64} \xleftarrow{\$} \mathbb{F}^v$, sends $\mathbf{r}_{64}$ to $\mathbf{P}$, and queries $[\hat{O}]$ to initialize

$$\boldsymbol{\lambda}_{64} := (1, 0, 0), \quad \sigma_{64} := \hat{O}(\mathbf{r}_{64}) .$$

3. For $j = 63, 62, \dots, 0$, the parties reduce the current claim $\boldsymbol{\lambda}_{j+1} \cdot \hat{\mathbf{s}}_{j+1}(\mathbf{r}_{j+1}) = \sigma_{j+1}$ as follows:

(a) $\mathbf{P}$ and $\mathbf{V}$ engage in sumcheck, with individual-degree bound 6, for the claim

$$\sum_{\mathbf{x} \in \{0,1\}^v} \text{eq}_v(\mathbf{r}_{j+1}, \mathbf{x}) \cdot \boldsymbol{\lambda}_{j+1} \cdot T_j(\hat{\mathbf{s}}_j(\mathbf{x})) = \sigma_{j+1} .$$

Denote its final random point by $\mathbf{r}_j \in \mathbb{F}^v$ and its claimed final evaluation by y_j . $\mathbf{V}$ rejects if any sumcheck degree or consistency check fails.

(b) $\mathbf{P}$ sends $\mathbf{a}_j \in \mathbb{F}^3$, the claimed value of $\hat{\mathbf{s}}_j(\mathbf{r}_j)$. $\mathbf{V}$ rejects unless

$$y_j = \text{eq}_v(\mathbf{r}_{j+1}, \mathbf{r}_j) \cdot \boldsymbol{\lambda}_{j+1} \cdot T_j(\mathbf{a}_j) .$$

(c) If $j > 0$, after receiving $\mathbf{a}_j$, $\mathbf{V}$ samples and sends $\alpha_j \xleftarrow{\$} \mathbb{F}$. Both parties set

$$\boldsymbol{\lambda}_j := (1, \alpha_j, \alpha_j^2), \quad \sigma_j := \boldsymbol{\lambda}_j \cdot \mathbf{a}_j .$$

4. $\mathbf{V}$ queries the input oracles and accepts if all preceding checks pass and

$$(\hat{A}(\mathbf{r}_0), \hat{B}(\mathbf{r}_0), \hat{C}(\mathbf{r}_0)) = \mathbf{a}_0 .$$

Figure 6: PIOP for $\mathcal{R}_{\text{Pos2}}$.

Proving full Merkle trees. We now use the PIOP for the batch Poseidon relation to prove that $\text{cm}_0 = \text{MT.Commit}(\mathbf{w})$. Recall that $\mathbf{w} := E(\mathbf{z}_0)$ which is the result of applying the map defined in (18) to each element of the initial vector $\mathbf{z}_0$. Also, set $v = h + 1$.

Recall the structure of a Merkle tree: node 1 is the root, nodes $2j$ and $2j + 1$ are the children of node j , and node $4N + i$ is leaf i . The prover initializes the vector $\mathbf{O} \in \mathbb{F}^{8N}$, representing the entire Merkle tree as follows:

$$\mathbf{O}[j] := \begin{cases} \text{Pos2}(\mathbf{w}[j - 4N], 0, 1) & \text{if } 4N \leq j < 8N , \\ \text{Pos2}(\mathbf{O}[2j], \mathbf{O}[2j + 1], 2) & \text{if } 2 \leq j < 4N , \\ \text{cm}_0 & \text{if } j = 1 , \\ \text{Pos2}(0, 0, 0) & \text{if } j = 0 , \end{cases}$$

and sends $\llbracket \hat{O} \rrbracket$ to the verifier. Using this polynomial oracle, we can initialize three input polynomials, which represent the first, second, and third inputs of each Poseidon hash that we need to prove:

$$\begin{aligned}\hat{A}(\mathbf{x}, b) &:= (1 - b) \cdot \hat{O}(0, \mathbf{x}) + b \cdot \hat{w}(\mathbf{x}) - I_0(\mathbf{x}, b) \cdot \text{Pos2}(0, 0, 0) , \\ \hat{B}(\mathbf{x}, b) &:= (1 - b) \cdot \hat{O}(1, \mathbf{x}) - I_0(\mathbf{x}, b) \cdot \text{cm}_0 , \\ \hat{C}(\mathbf{x}, b) &:= b + 2(1 - b) - 2I_0(\mathbf{x}, b) ,\end{aligned}$$

where $I_0(\mathbf{x}, b) := (1 - b) \prod_{k=0}^{h-1} (1 - x_k)$, $(\mathbf{x}, b) \in \{0, 1\}^v$ is the polynomial that is 1 on an all zero input and 0 otherwise. The verifier can initialize I_0 on its own. $\hat{A}, \hat{B}, \hat{C}$ can all be derived from $\hat{w}, \hat{O}, I_0$, so the prover does not have to explicitly send oracles of these three polynomials. Finally, the verifier just needs to do two additional checks: $\hat{O}(0^v) = \text{Pos2}(0, 0, 0)$ and $\hat{O}(1, 0^{v-1}) = \text{cm}_0$.

The Merkle-tree PIOP thus invokes the Poseidon-hash PIOP once on $(\hat{A}, \hat{B}, \hat{C}, \hat{O})$, which proves the original claim that $\text{cm}_0 = \text{MT.Commit}(\mathbf{w})$. Note that the Merkle-tree PIOP requires sending only one new polynomial oracle of length $8N$.

Input: Public parameters $(\mathbb{F}, h, \text{Pos2})$ and a root $\text{cm}_0 \in \mathbb{F}$. $\mathbf{P}$ knows an h -variate multilinear polynomial $\hat{w}$, and $\mathbf{V}$ has $\llbracket \hat{w} \rrbracket$. The claimed polynomial oracle relation is $\mathcal{R}_{\text{full}}(\mathbb{F}, h, \text{Pos2})$, with oracle input $(\text{cm}_0, \llbracket \hat{w} \rrbracket)$.

1. $\mathbf{P}$ constructs the complete-tree vector $\mathbf{O} \in \mathbb{F}^{2^{h+1}}$ in the layout defined above, substituting 2^h for $4N$, and sends its $(h + 1)$ -variate oracle $\llbracket \hat{O} \rrbracket$.
2. Both parties derive the $(h + 1)$ -variate input polynomials $\hat{A}, \hat{B}, \hat{C}$ from $\hat{w}, \hat{O}, I_0$ using the formulas above. $\mathbf{V}$ evaluates these inputs using $\llbracket \hat{w} \rrbracket, \llbracket \hat{O} \rrbracket$ and the public polynomial I_0 ; no additional input oracles are sent.
3. $\mathbf{V}$ queries $\llbracket \hat{O} \rrbracket$ and rejects unless

$$\hat{O}(0^{h+1}) = \text{Pos2}(0, 0, 0), \quad \hat{O}(1, 0^h) = \text{cm}_0 .$$

4. $\mathbf{P}$ and $\mathbf{V}$ engage in the batch Poseidon2 PIOP of Fig. 6 for $\mathcal{R}_{\text{Pos2}}(\mathbb{F}, h + 1, \text{Pos2})$, on polynomials $(\hat{A}, \hat{B}, \hat{C}, \hat{O})$ and their corresponding oracles. $\mathbf{V}$ accepts if this subprotocol and all preceding checks accept.

Figure 7: PIOP for $\mathcal{R}_{\text{full}}$.

Proving partial Merkle trees. Consider the first Merkle verification clause of UpdateStmt,

$$\text{VC.Verify}(\text{pp}, \text{cm}_0, U, \{\alpha_i\}_{i \in U}, \text{pf}_{\text{upd}}) = 1 ,$$

which corresponds to the membership claim $(\text{cm}_0; \hat{L}_\alpha, \hat{B}_{\text{upd}}) \in \mathcal{R}_{\text{partial}}$ as defined in Indexed oracle relation 5.8. The following construction generalizes to the other three Merkle verification clauses. Let $k := |U| \geq 1$ and use $h := \log(4N)$ from the full Merkle tree PIOP. The indices $i \in U$, values α_i , and sibling hashes in pf_{upd} are part of the witness, and only the root cm_0 is public.

Consider a matrix $\mathbf{U} \in \mathbb{F}^{2^w \times 2^u}$ that describes the k update records $(i, \alpha_i, \beta_i, \sigma_i)$, where $2^w = O(k)$ is an upper bound on the number of update records, and $2^u = O(\log N)$; the derived leaf tables

have $v_L = w$. Each active row represents an update record, and the columns include its selector s_U , i , α_i , β_i , σ_i , and whether its wire belongs to an addition gate, a multiplication gate, or a public input. Each active row also contains bit representations of its wire coordinates, used to identify the corresponding gate and the leaf position $p(i)$ below; these account for the total $O(\log N)$ columns. The prover sends $\llbracket \hat{U} \rrbracket$ to the verifier.

Now, consider the partial tree formed by the paths from the k selected leaves to the Merkle root, plus the boundary nodes containing the sibling hashes from $\mathbf{pf}_{\text{upd}}$. If there are I internal nodes, including the root, this partial tree has $2I + 1$ nodes, of which $I + 1 - k$ are boundary nodes. Clearly, $I \leq kh$, and the total number of nodes in the partial tree is at most $2I + 1 = O(k \log N)$. We choose R as the least power of two at least $2h \cdot 2^w + 1$.

Besides verifying the hash evaluations, the PIOP must prove that these nodes form the ordered partial tree specified by the Merkle opening: every selected leaf must connect to the public root at its claimed position and depth. We use auxiliary routing labels for this structural check. Thus, given $z \in \mathbb{F}^\times$ with multiplicative order greater than $\max\{R, h\}$, we require that the address polynomials

$$A_{b_1 \dots b_d}(z) = z^d + \sum_{j=1}^d b_j z^{d-j} \quad (0 \leq d \leq h, b_j \in \{0, 1\})$$

have distinct evaluations for distinct tree positions. The address label A records the root-to-node path, and the level label is $L = z^\ell$ for a node of height ℓ above the leaves. Write $\text{enc}_z(p) = \sum_{j=0}^{h-1} p_j z^j$ for the binary digits p_j of p , and $\rho(L) = [L(L-1)]^{-1}$ for internal-node inverse witnesses.

The prover sends $\llbracket \hat{T} \rrbracket$ for a matrix $\mathbf{T} \in \mathbb{F}^{R \times 8}$, one row corresponding to each node in the partial tree (padded to a power of 2). Its columns and honest rows are specified below, where $\zeta = \text{Pos2}(0, 0, 0)$ and the row type τ is one of five distinct public field elements. The symbols d, d_L, d_R denote hashes placed in the indicated columns.

	τ	A	L	q	a	b	c	o
selected leaf	L	$z^h + \text{enc}_z(p(i))$	1	0	α_i	0	1	d
internal	I	A	L	$\rho(L)$	d_L	d_R	2	d
root	R	1	z^h	$\rho(z^h)$	d_L	d_R	2	cm_0
boundary	B	A	L	d	0	0	0	ζ
padding	0	0	0	0	0	0	0	ζ

(20)

In the above table, the a, b, c, o columns correspond to the inputs and output of the Poseidon2 hash function for that node. The auxiliary column q serves two purposes, depending on the row type. On internal and root rows, the polynomial constraint $L(L-1)q = 1$ enforces $L \neq 0$ and $L \neq 1$. On boundary rows, it stores the sibling hash contained in $\mathbf{pf}_{\text{upd}}$. Note that this hash is not recomputed, so the hash columns just contain the dummy evaluation $(0, 0, 0, \zeta)$. Thus, the node hash d used in the routing checks below is o on computed nodes and q on boundary nodes.

A zerocheck enforces the row types and the equalities in the table, with row zero designated as the unique root. The four hash columns are v_R -variate restrictions $\hat{A}, \hat{B}, \hat{C}, \hat{O}$ of $\hat{T}$, so the Poseidon2 PIOP can be invoked on these four polynomials with the verifier only having $\llbracket \hat{T} \rrbracket$.

Finally, we can connect the verified hashes into a partial tree and bind its leaves and boundary nodes using three multiset-equality checks. The first check uses $\llbracket \hat{T} \rrbracket$ to match the non-root node records (A, L, d) with the child references $(zA, L/z, a)$ and $(zA + 1, L/z, b)$ from internal and root rows. This ensures that each child hash used as the input to a hash invocation is obtained from

the corresponding node. The second check uses $\llbracket \hat{T} \rrbracket$ and $\llbracket \hat{U} \rrbracket$ to match the selected-leaf records $(A - z^h, a)$ with the update records $(\text{enc}_z(p(i)), \alpha_i)$. The third matches the boundary-node records (A, L, q) with the active records (A_B, L_B, d_B) in $\hat{B}$, binding both update-root invocations to the same opening. We use the multiset equality PIOP for this check as described in Section 5.1.

To see why these checks establish the opening's tree structure, match child-reference occurrences with non-root node occurrences using multiset equality, preserving multiplicities. Every non-root occurrence then has exactly one parent, so a component disconnected from the unique root would contain a directed cycle of computed nodes. A cycle of length $c \leq R$ would imply $L = z^{-c}L$; since $L \neq 0$, this contradicts the order of z . Thus every node is connected to the root. At depth d , its level label is z^{h-d} ; at depth h this equals 1, where computed internal nodes are forbidden. Selected leaves have level label 1 and depth h , so address injectivity uniquely defines their positions. In our $\mathbb{F}_{2^{128}}$ implementation in Section 5.4, $z = X$ is the polynomial-basis element of multiplicative order $2^{128} - 1$. For $h < 128$, address injectivity follows because distinct address polynomials differ by a nonzero polynomial of degree less than the degree-128 defining polynomial of the field.

The prover sends $\llbracket \hat{U} \rrbracket$ and the boundary column oracles before the partial-tree invocations. The node oracle $\llbracket \hat{T} \rrbracket$ has $8R = O(k \log N)$ entries and $v_R + 3$ variables, independent of the number of Poseidon2 rounds. The update-record oracle $\llbracket \hat{U} \rrbracket$ has $2^{u+w} = O(k \log N)$ entries and $u + w$ variables.

Relation and input: $\mathcal{R}_{\text{partial}}(\mathfrak{i})$, with public root cm . The index is $\mathfrak{i} = (\mathbb{F}, h, \text{Pos2}, v_L, v_B, v_R, z)$, where $h \geq 1$, v_L, v_B are the leaf and boundary row arities, $R := 2^{v_R}$ is the node capacity, and z satisfies the order and address-injectivity conditions in the text. The prover knows the leaf and boundary tuples $\hat{L}, \hat{B}$ of multilinear column polynomials from Indexed oracle relation 5.6, with column arities v_L and v_B respectively; the verifier has their column oracles. Leaf columns may be derived from $\llbracket \hat{U} \rrbracket$, with consistency enforced by the update-record checks. All these input oracles are supplied by the caller and fixed before this invocation.

1. **P**: Form the node table $\mathbf{T} \in \mathbb{F}^{R \times 8}$ as in (20), with the root in row zero, using the supplied leaf and boundary records. Send $\llbracket \hat{T} \rrbracket$ and a leaf row $\mathbf{x}_L \in \{0, 1\}^{v_L}$ with $\hat{s}_L(\mathbf{x}_L) = 1$. The verifier checks that $\mathbf{x}_L$ is binary and queries $\llbracket \hat{s}_L \rrbracket$ to check this equality.
2. **P** and **V**: Write $(\tau, A, L, q, a, b, c, o)$ for the column restrictions of $\hat{T}$, suppressing hats on these and the input columns below. Let $\mathcal{T} = \{0, \text{L}, \text{I}, \text{R}, \text{B}\}$ be the five distinct public type constants, and define the virtual polynomials

$$\begin{aligned} \delta_t &:= \prod_{u \in \mathcal{T} \setminus \{t\}} \frac{\tau - u}{t - u} \quad (t \in \mathcal{T}) , \\ \delta_C &:= \delta_I + \delta_R , \\ \delta_N &:= \delta_L + \delta_I + \delta_B , \\ d &:= (\delta_L + \delta_C)o + \delta_B q , \\ e_0(\mathbf{x}) &:= \text{eq}_{v_R}(0^{v_R}, \mathbf{x}) . \end{aligned}$$

Write $\zeta = \text{Pos2}(0, 0, 0)$. Engage in zerocheck PIOPs for $\mathcal{R}_{\text{zero}}$, enforcing that all the following

expressions are zero on $\{0, 1\}^{v_R}$:

$$\prod_{t \in \mathcal{T}} (\tau - t) \ , \ \delta_R - e_0 \ , \ \delta_L(L - 1, q, b, c - 1) \ , \ \delta_C(L(L - 1)q - 1, c - 2) \ , \\ \delta_R(A - 1, L - z^h, o - \text{cm}) \ , \ \delta_B(a, b, c, o - \zeta) \ , \ \delta_0(A, L, q, a, b, c, o - \zeta) \ .$$

Also engage in zerocheck PIOPs on $\{0, 1\}^{v_L}$ and $\{0, 1\}^{v_B}$, respectively, for

$$s_L(s_L - 1) \ , \ (s_L p_j(p_j - 1))_{j=0}^{h-1} \ , \ s_B(s_B - 1) \ .$$

3. **P** and **V**: Engage in the batch Poseidon2 PIOP of Fig. 6 on the four restrictions (a, b, c, o) of $\hat{T}$. Include the dummy boundary and padding hashes in this batch.
4. **P** and **V**: Engage in a PIOP for $\mathcal{R}_{\text{mset}}$ with the selected tuple expressions

$$\mathbf{M}(\delta_N; (A, L, d)) = \mathbf{M}(\delta_C; (zA, L/z, a)) \uplus \mathbf{M}(\delta_C; (zA + 1, L/z, b)) \ .$$

The right side uses an additional binary column for the two child collections.

5. **P** and **V**: Engage in a PIOP for $\mathcal{R}_{\text{mset}}$ with node-row and leaf-row domains $\{0, 1\}^{v_R}$ and $\{0, 1\}^{v_L}$, and selected tuple expressions

$$\mathbf{M}(\delta_L; (A - z^h, a)) = \mathbf{M} \left(s_L; \left(\sum_{j=0}^{h-1} p_j z^j, \nu \right) \right) \ .$$

6. **P** and **V**: Engage in a PIOP for $\mathcal{R}_{\text{mset}}$ with node-row and boundary-row domains $\{0, 1\}^{v_R}$ and $\{0, 1\}^{v_B}$, and selected tuple expressions

$$\mathbf{M}(\delta_B; (A, L, q)) = \mathbf{M}(s_B; (A_B, L_B, d_B)) \ .$$

Use the same $\hat{B}$ in both update-root invocations.

7. **V**: Accept if and only if the direct check and all invoked PIOPs accept.

Figure 8: PIOP for $\mathcal{R}_{\text{partial}}$.

5.4 Implementation and evaluation

We evaluate our dynamic SNARK against recomputing a HyperPlonk proof from scratch after each update. We measure prover time, verifier time, and serialized proof size, and separately evaluate how initialization is amortized over long update sequences. We do not implement the zero-knowledge compiler modifications described in Appendix C.

Implementation and cryptographic parameters. Our Rust implementation uses Ligerito [NA25] for multilinear polynomial commitments. We use the $\mathbb{F}_{2^{128}}$ arithmetic implementation provided by flock-core [BRW27], with hardware-accelerated carry-less multiplication. The baseline instantiates the HyperPlonk PIOP [CBBZ23] with the same field and Ligerito backend as

the dynamic SNARK, and proves the circuit constraints and public inputs without the additional assignment-Merkle claims needed by the dynamic construction. We use Ligerito’s Fast profile: rate-1/2 encodings and a 100-bit per-round target in the Johnson-bound regime with out-of-domain sampling. Ligerito’s Merkle commitments and internal Fiat–Shamir transcript use SHA-256, while the surrounding PIOP transcripts use BLAKE3. These settings are shared by the baseline, initialization, and polynomial update proofs. The implementation is available at <https://github.com/randomshuffle/dynamic-snarks-system>.

Relation-specific Merkle trees use 64-byte-to-32-byte Poseidon2b compression [GKK+26], with four binary-field words, eight full rounds, 58 partial rounds, and an x^7 S-box. The evaluated implementation retains the leaf index as a hash input and uses a height-independent internal hash. The Poseidon hash proof commits one four-word checkpoint after round 33 and batches two 33-round GKR segments.

Buffered updates. We do not issue static SNARK proofs for small updates, and simply send the updated indices and values along with the shared Merkle opening proof that certifies the new commitment. The verifier authenticates the old and new values using the shared Merkle opening proof and checks the affected gate and wiring constraints directly. We bound the total size of these SNARK-free updates by 2 MiB. When the accumulated SNARK-free updates exceed this size cap, we generate a SNARK for them and insert it into the hierarchy, performing any required merges. We call this a *SNARK promotion*.

Experimental setup. We run the experiments on an AWS `r7a.8xlarge` instance with an AMD EPYC 9R14 processor, 32 vCPUs (32 physical cores, one hardware thread per core), and 256 GiB of memory, running Ubuntu 24.04 on x86-64. We compile with Rust 1.92.0 in release mode and `target-cpu=native`, and use 32 Rayon workers. The circuit has $N = 2n = 2^{23}$ gates, equally divided between addition and multiplication, with identity wiring and 256 fixed public inputs.

Update-size sweep. We generate a full initial dynamic proof once and start each update-size trajectory from a copy of the same initialized state. For each $g \in \{1, 2, 4, 8, 16, 32, 64, 128\}$, we apply 40 consecutive updates, each selecting g distinct gates uniformly without replacement. All three wire positions of each selected gate are updated, so $|U| = 3g$, while public inputs remain fixed. We generate a fresh baseline proof of the same updated assignment at every revision and alternate the execution order of each baseline/update pair. We report the mean and maximum over each 40-update trajectory. Prover times include witness construction and data structure maintenance, and exclude indexing, verification, serialization, statistics collection, workload generation, and I/O.

Proof-size accounting. Proof size is the complete `bincode` serialization, including field elements, hashes, indices, dimensions, and container overhead. $\mathbb{F}_{2^{128}}$ elements occupy 16 bytes and hashes occupy 32 bytes. For a dynamic update, the reported size includes the initial proof, SNARK proofs for the updates, the SNARK-free update proofs, and the current public-input opening.

Long-sequence methodology. We additionally run 10,000 successive one-gate updates and 1,000 successive 128-gate updates, each from a fresh initialization with the same circuit and parameters. Every dynamic update is measured, including all SNARK promotions and hierarchy merges. To limit experiment duration, we measure 101 fresh baseline proofs along each trajectory: at revision 1 and every 100 revisions for the one-gate experiment, and at revision 1 and every 10 revisions for the 128-gate experiment, including both final revisions. If T_{init} is the measured initialization time, u_i is the dynamic prover time at revision i , and $\bar{b}$ is the mean of these baseline samples, the cumulative

	Prover (ms)		Verifier (ms)		Proof (MiB)		Prover speedup
	Mean	Max.	Mean	Max.	Mean	Max.	
HyperPlonk	7,970	8,560	4.70	4.88	1.22	1.22	1.00×
Initial dynamic	310,000	—	13.5	—	2.28	—	—
Update size 1	0.39	0.56	15.4	17.4	2.31	2.33	20,500×
Update size 2	1.27	1.74	17.3	21.2	2.33	2.38	6,300×
Update size 4	2.15	2.58	21.1	28.3	2.38	2.46	3,690×
Update size 8	3.10	3.71	28.3	41.7	2.46	2.61	2,570×
Update size 16	3.84	4.90	42.1	67.9	2.62	2.91	2,070×
Update size 32	4.03	6.24	68.6	118	2.91	3.46	1,980×
Update size 64	66.5	2,430	101	193	3.41	4.27	120×
Update size 128	190	4,850	107	202	4.09	5.73	41.8×

Table 3: Sweep results for $N = 2^{23}$ gates. Update rows summarize 40 proofs each. Each speedup is the ratio of the corresponding 40 paired baseline and update mean prover times, excluding initialization. The speedup decreases drastically from update size 32 to 64 since that is the smallest update size for which 40 sequential updates result in a SNARK promotion.

speedup after t updates is

$$S(t) = \frac{t\bar{b}}{T_{\text{init}} + \sum_{i=1}^t u_i} .$$

Tables 4 and 5 compare each sequence with the HyperPlonk baseline. The baseline recomputes each proof from scratch and has no separate proof initialization, so its amortized prover time equals its sampled mean. For both systems, proof sizes and verifier times are measured for the complete proof at the final revision. Figures 9 and 10 then chart the prover speedup obtained over the course of the 10,000 one-gate updates and 1,000 128-gate updates respectively.

	HyperPlonk (baseline)	Dynamic SNARK
Initialization (s)	—	311
Mean prover time per update (ms)	8,010	23.2
Amortized prover time including init (ms)	8,010	54.3
Speedup including initialization	1.00×	147×
Speedup also including indexing	1.00×	144×
Break-even revision	—	39
SNARK promotions	—	4
Final proof size (MiB)	1.22	4.63
Final verifier time (ms)	4.84	84.9

Table 4: Results for 10,000 successive one-gate updates at $N = 2^{23}$ gates. Baseline prover means use 101 sampled proofs; dynamic means include every SNARK promotion and hierarchy merge. The final initialization-inclusive speedup is also the maximum observed in the measured sequence.

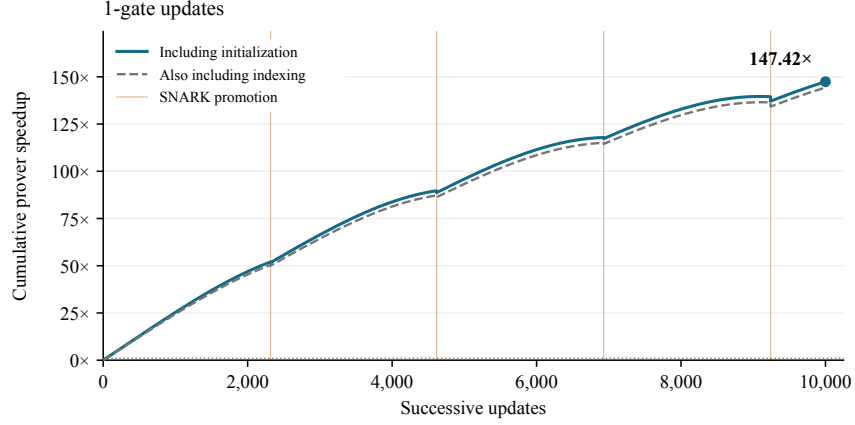


Figure 9: Cumulative prover speedup over 10,000 successive one-gate updates. The solid curve charges a full initialization once. The dashed curve also includes indexing on both sides. Vertical markers indicate SNARK promotions. The baseline total uses the mean of 101 sampled proofs.

	HyperPlonk (baseline)	Dynamic SNARK
Initialization (s)	—	310
Mean prover time per update (ms)	7,800	358
Amortized prover time including init (ms)	7,800	668
Speedup including initialization	1.00×	11.7×
Speedup also including indexing	1.00×	11.5×
Break-even revision	—	41
SNARK promotions	—	52
Final proof size (MiB)	1.21	9.35
Final verifier time (ms)	5.19	211

Table 5: Results for 1,000 successive 128-gate updates at $N = 2^{23}$ gates. Baseline prover means use 101 sampled proofs; dynamic means include every SNARK promotion and hierarchy merge. The final initialization-inclusive speedup is also the maximum observed in the measured sequence.

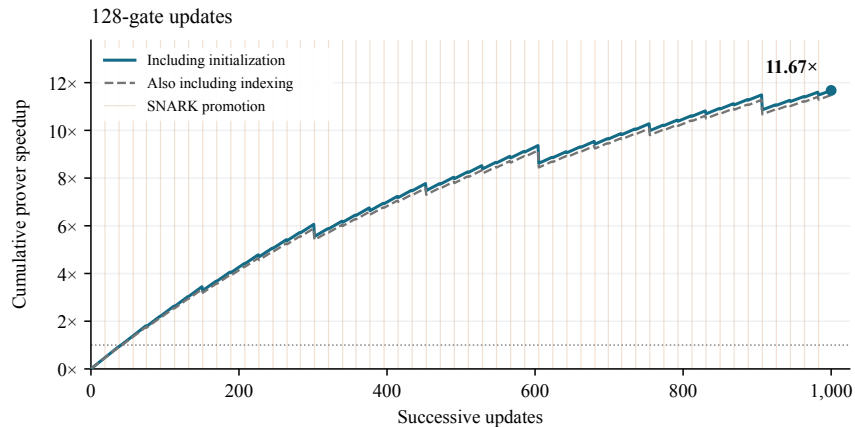


Figure 10: Cumulative prover speedup over 1,000 successive 128-gate updates. Initialization, indexing, promotion markers, and sampled-baseline accounting follow Fig. 9.

AI use disclosure

We used ChatGPT 5.6 Sol and ChatGPT 6 Astra for implementing and benchmarking our construction, drafting proof sketches, creating figures, revising exposition, formatting, and checking our proofs for errors. We take full responsibility for the content of the paper, and all the proofs were finalized and verified by us. The code was verified through tests written by the models and extensive conversations where the models demonstrated a thorough understanding of the algorithms.

References

- [ABC+12] J. H. Ahn, D. Boneh, J. Camenisch, S. Hohenberger, A. Shelat, and B. Waters. “Computing on Authenticated Data”. In: TCC ’12. URL: <https://eprint.iacr.org/2011/096>.
- [ABF24] G. Anthoine, D. Balbás, and D. Fiore. “Fully-Succinct Multi-Key Homomorphic Signatures from Standard Assumptions”. In: CRYPTO ’24. URL: <https://eprint.iacr.org/2024/895>.
- [ACFY25] G. Arnon, A. Chiesa, G. Fenzi, and E. Yogev. “WHIR: Reed–Solomon proximity testing with super-fast verification”. In: EUROCRYPT ’25. URL: <https://eprint.iacr.org/2024/1586>.
- [AO26] E. Aldema Tshuva and R. Oshman. “Model-Generic Incrementally Verifiable Computation from Updatable BARGs”. In: vol. 362. ITCS ’26. URL: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITCS.2026.6>.
- [APKP24] K. Abbaszadeh, C. Pappas, J. Katz, and D. Papadopoulos. “Zero-Knowledge Proofs of Training for Deep Neural Networks”. In: CCS ’24. URL: <https://eprint.iacr.org/2024/162>.
- [BC23] B. Bünz and B. Chen. “Protostar: Generic Efficient Accumulation/Folding for Special-Sound Protocols”. In: ASIACRYPT ’23. URL: <https://eprint.iacr.org/2023/620>.
- [BCC+16] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. “Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting”. In: EUROCRYPT ’16. URL: <https://eprint.iacr.org/2016/263>.
- [BCCT13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. “Recursive Composition and Bootstrapping for SNARKs and Proof-Carrying Data”. In: STOC ’13. URL: <https://eprint.iacr.org/2012/095>.
- [BCG+13] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. “SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge”. In: CRYPTO ’13. URL: <https://eprint.iacr.org/2013/507>.
- [BCG+14] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: S&P ’14. URL: <https://eprint.iacr.org/2014/349>.
- [BCG+17] E. Ben-Sasson, A. Chiesa, A. Gabizon, M. Riabzev, and N. Spooner. “Interactive Oracle Proofs with Constant Rate and Query Complexity”. In: ICALP ’17. URL: <https://eprint.iacr.org/2016/324>.
- [BCMS20] B. Bünz, A. Chiesa, P. Mishra, and N. Spooner. “Proof-Carrying Data from Accumulation Schemes”. In: TCC ’20. URL: <https://eprint.iacr.org/2020/499>.
- [BCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. “Interactive Oracle Proofs”. In: TCC ’16. URL: <https://eprint.iacr.org/2016/116>.
- [BCTV14a] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Scalable Zero Knowledge via Cycles of Elliptic Curves”. In: CRYPTO ’14. URL: <https://eprint.iacr.org/2014/595>.

- [BCTV14b] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. “Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture”. In: USENIX Security ’14. URL: <https://eprint.iacr.org/2013/879>.
- [BDF+11] D. Boneh, Ö. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry. “Random Oracles in a Quantum World”. In: ASIACRYPT ’11. URL: <https://eprint.iacr.org/2010/428>.
- [BEG+91] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. “Checking the correctness of memories”. In: FOCS ’91. URL: <https://doi.org/10.1109/SFCS.1991.185352>.
- [BFR+13] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. “Verifying computations with state”. In: SOSP ’13. URL: <https://eprint.iacr.org/2013/356>.
- [BFRW26] B. Bünz, G. Fenzi, R. D. Rothblum, and W. Wang. “TensorSwitch: Nearly Optimal Polynomial Commitments from Tensor Codes”. In: CRYPTO ’26. URL: <https://eprint.iacr.org/2025/2065>.
- [BFS20] B. Bünz, B. Fisch, and A. Szeponiec. “Transparent SNARKs from DARK Compilers”. In: EUROCRYPT ’20. URL: <https://eprint.iacr.org/2019/1229>.
- [BGH19] S. Bowe, J. Grigg, and D. Hopwood. “Halo: Recursive Proof Composition without a Trusted Setup”. Cryptology ePrint Archive, Paper 2019/1021. URL: <https://eprint.iacr.org/2019/1021>.
- [BMM+21] B. Bünz, M. Maller, P. Mishra, N. Tyagi, and P. Vesely. “Proofs for Inner Pairing Products and Applications”. In: ASIACRYPT ’21. URL: <https://eprint.iacr.org/2019/1177>.
- [BMMS26] A. Baweja, P. Mishra, T. Mopuri, and M. Shtepel. “Query-Optimal IOPPs for Linear-Time Encodable Codes”. In: EUROCRYPT ’26. URL: <https://eprint.iacr.org/2025/1588>.
- [BMPR26] M. Batista, Á. Montes, N. Paslis, and C. Ràfols. “Dynasaurs: Efficient Universal Dynamic zkSNARKs from Sparse Linear Arguments”. Cryptology ePrint Archive, Paper 2026/1892. URL: <https://eprint.iacr.org/2026/1892>.
- [BR93] M. Bellare and P. Rogaway. “Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols”. In: CCS ’93. URL: <https://web.cs.ucdavis.edu/~rogaway/papers/ro-abstract.html>.
- [BRW27] B. Bünz, R. D. Rothblum, and W. Wang. “Flock: Fast Proving for Batch Boolean Computations”. In: S&P ’27. URL: <https://eprint.iacr.org/2026/1329>.
- [But21] V. Buterin. “An incomplete guide to rollups”. URL: <https://vitalik.eth.limo/general/2021/01/05/rollup.html>.
- [CBBZ23] B. Chen, B. Bünz, D. Boneh, and Z. Zhang. “HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates”. In: EUROCRYPT ’23. URL: <https://eprint.iacr.org/2022/1355>.
- [CCH+18] R. Canetti, Y. Chen, J. Holmgren, A. Lombardi, G. N. Rothblum, and R. D. Rothblum. “Fiat–Shamir From Simpler Assumptions”. Cryptology ePrint Archive, Paper 2018/1004. URL: <https://eprint.iacr.org/2018/1004>.
- [CDHZ26] A. Chiesa, Z. Di, Z. Hu, and Y. Zheng. “How to Prove Post-Quantum Security for Succinct Non-Interactive Reductions”. In: EUROCRYPT ’26. URL: <https://eprint.iacr.org/2025/2166>.
- [CF13] D. Catalano and D. Fiore. “Vector Commitments and Their Applications”. In: PKC ’13. URL: <https://eprint.iacr.org/2011/495>.
- [CHM+20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS”. In: EUROCRYPT ’20. URL: <https://eprint.iacr.org/2019/1047>.

- [CMS19] A. Chiesa, P. Manohar, and N. Spooner. “Succinct Arguments in the Quantum Random Oracle Model”. In: TCC ’19. URL: <https://eprint.iacr.org/2019/834>.
- [COS20] A. Chiesa, D. Ojha, and N. Spooner. “Fractal: Post-Quantum and Transparent Recursive Proofs from Holography”. In: EUROCRYPT ’20. URL: <https://eprint.iacr.org/2019/1076>.
- [CWSK24] B. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang. “ZKML: An Optimizing System for ML Inference in Zero-Knowledge Proofs”. In: EuroSys ’24. URL: <https://doi.org/10.1145/3627703.3650088>.
- [FKL18] G. Fuchsbauer, E. Kiltz, and J. Loss. “The Algebraic Group Model and its Applications”. In: CRYPTO ’18. URL: <https://eprint.iacr.org/2017/620>.
- [FS24] M. Frigo and A. Shelat. “Anonymous credentials from ECDSA”. Cryptology ePrint Archive, Paper 2024/2010. URL: <https://eprint.iacr.org/2024/2010>.
- [GKK+26] L. Grassi, D. Khovratovich, K. Koschatko, C. Rechberger, M. Schofnegger, V. Schröppel, and Z. Wu. “Poseidon(2)b: Binary Field Versions of Poseidon/Poseidon2”. In: *IACR Communications in Cryptology* (2026). URL: <https://eprint.iacr.org/2025/1893>.
- [GKR+21] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger. “Poseidon: A New Hash Function for Zero-Knowledge Proof Systems”. In: USENIX Security ’21. URL: <https://eprint.iacr.org/2019/458>.
- [GKR15] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. “Delegating Computation: Interactive Proofs for Muggles”. In: *Journal of the ACM* (2015). URL: <https://eccc.weizmann.ac.il/report/2017/108/>.
- [GKS23] L. Grassi, D. Khovratovich, and M. Schofnegger. “Poseidon2: A Faster Version of the Poseidon Hash Function”. In: AFRICACRYPT ’23. URL: <https://eprint.iacr.org/2023/323>.
- [Gro16] J. Groth. “On the Size of Pairing-Based Non-interactive Arguments”. In: EUROCRYPT ’16. URL: <https://eprint.iacr.org/2016/260>.
- [GWC19] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. “PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge”. Cryptology ePrint Archive, Paper 2019/953. URL: <https://eprint.iacr.org/2019/953>.
- [Kil92] J. Kilian. “A note on efficient zero-knowledge proofs and arguments”. In: STOC ’92. URL: <https://doi.org/10.1145/129712.129782>.
- [KMS+16] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. “Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts”. In: S&P ’16. URL: <https://eprint.iacr.org/2015/675>.
- [KST22] A. Kothapalli, S. T. V. Setty, and I. Tzialla. “Nova: Recursive Zero-Knowledge Arguments from Folding Schemes”. In: CRYPTO ’22. URL: <https://eprint.iacr.org/2021/370>.
- [KZG10] A. Kate, G. M. Zaverucha, and I. Goldberg. “Constant-Size Commitments to Polynomials and Their Applications”. In: ASIACRYPT ’10. URL: https://doi.org/10.1007/978-3-642-17373-8_11.
- [LFKN92] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. “Algebraic Methods for Interactive Proof Systems”. In: *Journal of the ACM* (1992). URL: <https://doi.org/10.1145/146585.146605>.
- [Lib24] B. Libert. “Simulation-Extractable KZG Polynomial Commitments and Applications to Hyper-Plonk”. In: PKC ’24. URL: <https://eprint.iacr.org/2024/854>.
- [LXZ+24] T. Liu, T. Xie, J. Zhang, D. Song, and Y. Zhang. “Pianist: Scalable zkRollups via Fully Distributed Zero-Knowledge Proofs”. In: S&P ’24. URL: <https://eprint.iacr.org/2023/1271>.

- [LXZ21] T. Liu, X. Xie, and Y. Zhang. “zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy”. In: CCS ’21. URL: <https://eprint.iacr.org/2021/673>.
- [Mer87] R. C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: CRYPTO ’87. URL: https://doi.org/10.1007/3-540-48184-2_32.
- [MHKS14] A. Miller, M. Hicks, J. Katz, and E. Shi. “Authenticated Data Structures, Generically”. In: POPL ’14. URL: <https://doi.org/10.1145/2535838.2535851>.
- [Mic00] S. Micali. “Computationally Sound Proofs”. In: *SIAM Journal on Computing* (2000). Preliminary version appeared in FOCS ’94. URL: <https://doi.org/10.1137/S0097539795284959>.
- [NA25] A. Novakovic and G. Angeris. “Ligerito: A Small and Concretely Fast Polynomial Commitment Scheme”. Cryptology ePrint Archive, Paper 2025/1187. URL: <https://eprint.iacr.org/2025/1187>.
- [NDC+24] W. D. Nguyen, T. Datta, B. Chen, N. Tyagi, and D. Boneh. “Mangrove: A Scalable Framework for Folding-Based SNARKs”. In: CRYPTO ’24. URL: <https://eprint.iacr.org/2024/416>.
- [OWWB20] A. Ozdemir, R. S. Wahby, B. Whitehat, and D. Boneh. “Scaling Verifiable Computation Using Efficient Set Accumulators”. In: USENIX Security ’20. URL: <https://www.usenix.org/conference/usenixsecurity20/presentation/ozdemir>.
- [PGHR13] B. Parno, C. Gentry, J. Howell, and M. Raykova. “Pinocchio: Nearly Practical Verifiable Computation”. In: S&P ’13. URL: <https://eprint.iacr.org/2013/279>.
- [PSG+24] C. Papamanthou, S. Srinivasan, N. Gailly, I. Hishon-Rezaizadeh, A. Salumets, and S. Golemac. “Reckle Trees: Updatable Merkle Batch Proofs with Applications”. In: CCS ’24. URL: <https://eprint.iacr.org/2024/493>.
- [PST13] C. Papamanthou, E. Shi, and R. Tamassia. “Signatures of Correct Computation”. In: TCC ’13. URL: <https://eprint.iacr.org/2011/587>.
- [PSTY13] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. “Streaming Authenticated Data Structures”. In: EUROCRYPT ’13. URL: https://doi.org/10.1007/978-3-642-38348-9_22.
- [PTT11] C. Papamanthou, R. Tamassia, and N. Triandopoulos. “Optimal Verification of Operations on Dynamic Sets”. In: CRYPTO ’11. URL: <https://eprint.iacr.org/2010/455>.
- [RRR16] O. Reingold, R. Rothblum, and G. Rothblum. “Constant-Round Interactive Proofs for Delegating Computation”. In: STOC ’16. URL: <https://eccc.weizmann.ac.il/report/2016/061/>.
- [RWGM23] M. Rosenberg, J. D. White, C. Garman, and I. Miers. “zk-creds: Flexible Anonymous Credentials from zkSNARKs and Existing Identity Infrastructure”. In: S&P ’23. URL: <https://eprint.iacr.org/2022/878>.
- [SCP+22] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang. “Hyperproofs: Aggregating and Maintaining Proofs in Vector Commitments”. In: USENIX Security ’22. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/srinivasan>.
- [SLZ24] H. Sun, J. Li, and H. Zhang. “zkLLM: Zero Knowledge Proofs for Large Language Models”. In: CCS ’24. URL: <https://arxiv.org/abs/2404.16109>.
- [Spi96] D. A. Spielman. “Linear-time encodable and decodable error-correcting codes”. In: *IEEE Transactions on Information Theory* (1996). Preliminary version appeared in STOC ’95. URL: <https://www.cs.yale.edu/homes/spielman/PAPERS/linearTimeIT.pdf>.
- [STW23] S. Setty, J. Thaler, and R. S. Wahby. “Customizable Constraint Systems for Succinct Arguments”. Cryptology ePrint Archive, Paper 2023/552. URL: <https://eprint.iacr.org/2023/552>.
- [TB23] E. N. Tas and D. Boneh. “Vector Commitments with Efficient Updates”. In: AFT ’23. URL: <https://eprint.iacr.org/2023/1830>.

- [Val08] P. Valiant. “Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency”. In: TCC ’08. URL: https://doi.org/10.1007/978-3-540-78524-8_1.
- [Vid15] M. Viderman. “A combination of testability and decodability by tensor products”. In: *Random Structures and Algorithms* (2015). Preliminary version appeared in APPROX-RANDOM ’12. URL: <https://arxiv.org/abs/1105.5806>.
- [WPSP26a] W. Wang, C. Papamanthou, S. Srinivasan, and D. Papadopoulos. “Designated-Verifier Dynamic zk-SNARKs with Applications to Dynamic Proofs of Index”. In: CCS ’26. URL: <https://eprint.iacr.org/2026/144>.
- [WPSP26b] W. Wang, C. Papamanthou, S. Srinivasan, and D. Papadopoulos. “Dynamic zk-SNARKs (with applications to sparse zk-SNARKs and IVC)”. In: EUROCRYPT ’26. URL: <https://eprint.iacr.org/2024/1566>.
- [XZC+22] T. Xie, J. Zhang, Z. Cheng, F. Zhang, Y. Zhang, Y. Jia, D. Boneh, and D. Song. “zkBridge: Trustless Cross-chain Bridges Made Practical”. In: CCS ’22. URL: <https://arxiv.org/abs/2210.00264>.
- [ZGK+17] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. “vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases”. In: S&P ’17. URL: <https://eprint.iacr.org/2017/1145>.
- [ZOZ26] T. Zhang, Y. Ouyang, and Y. Zhang. “Dynark: Making Groth16 Dynamic”. In: EUROCRYPT ’26. URL: <https://eprint.iacr.org/2025/1897>.

A Proof of the dynamic SNARK compiler theorem

A.1 Amortized efficiency

Proof of prover and verifier efficiency claims in Theorem 4.1. The claims concerning initial-prover time, verifier time, and proof size follow directly from the construction. It remains to prove the update-prover time claim.

Consider a non-reinitializing update that produces a table at level t , and let the merged update index set be U_t . We have $2^{\ell-t} \leq |U_t| < 2^{\ell-t+1}$ and hence $|U_t| \leq q_t$. By the construction in Fig. 4, the cost of this update is at most

$$O(|U_t|) + T_{\text{cupd}}^{\text{vc}}(\lambda, m, q_t) + T_{\text{pupd}}^{\text{vc}}(\lambda, m, q_t, n_0) + 3T_{\text{open}}^{\text{vc}}(\lambda, m, q_t) + T_{\mathbf{P}}^{\text{arg}}(\lambda, N_{\text{upd}}(q_t)) .$$

This cost is at most $c \cdot 2^{\ell-t} \cdot \tau_{\text{upd}}(\lambda, m, n_0)$ for some constant $c > 0$, where τ_{upd} is defined in (8).

Now, we discuss the amortized cost. Consider an update of size k such that $2^{\ell-s} \leq k < 2^{\ell-s+1}$ for some $s \in [\ell]$. We charge $c \cdot k \cdot \tau_{\text{upd}}(\lambda, m, n_0)$ to levels $1, 2, \dots, s$. Since $s = O(\log(m/k + 1))$, the total cost charged to this update is $O(k \cdot \log(m/k + 1) \cdot \tau_{\text{upd}}(\lambda, m, n_0))$. By the update rule in Fig. 4, a table at level t is built only when the size of the incoming update plus the total size of the update tables at levels $t + 1, \dots, \ell$ reaches $2^{\ell-t}$. The charge to level t contributed by the current update has not been spent, and neither have the charges to level t contributed by the updates represented in the tables at levels $t + 1, \dots, \ell$ (because these tables must have been built after the last time a table was built at level t). Thus, level t has an unspent charge of at least $c \cdot 2^{\ell-t} \cdot \tau_{\text{upd}}(\lambda, m, n_0)$, which can be used to cover the required cost of building the new table.

Finally, a reinitialization occurs only when all levels are nonempty. At that point, the total update size for all the ℓ tables is at least $\sum_{j=1}^{\ell} 2^{\ell-j} = 2^{\ell} - 1 \geq m$, and the prover has to do $T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0)$ work to reinitialize. We charge $c \cdot k \cdot (T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0)/m)$ to an update of size k . Thus, by the time the total update size reaches m , the total charge is at least $c \cdot T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0)$, which is enough to pay for the reinitialization.

Adding the update charge and the reinitialization charge, each update of size k is charged $O(k \cdot \log(m/k + 1) \cdot \tau_{\text{upd}}(\lambda, m, n_0) + (k/m) \cdot T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0))$ in total, which is the bound in (9) as desired. The stated specialization follows immediately. $\square$

In Appendix B, we discuss how to deamortize the update time without any asymptotic overhead.

A.2 Knowledge soundness

Proof of the knowledge-error and extractor-size claims in Theorem 4.1. Fix a t -size dynamic adversary and let $t_0, \dots, t_{\ell}$ be the corresponding induced adversary sizes from the theorem. For notational convenience, define

$$\forall s \in \{0, \dots, \ell\} : \kappa_s := \kappa^{\text{arg}}(\lambda, N_*, t_s) \quad , \quad C_s := C_{\mathbf{E}}^{\text{arg}}(\lambda, N_*, t_s) .$$

By Lemma 3.11, the strong local-updatability error used below is at most $\epsilon_{\text{pu}}^{\text{vc}}$.

Extractor construction. Let $\mathfrak{x}$ be the current public input. The dynamic SNARK extractor $\mathbf{E}_{\text{DYN}}$ invokes $\mathbf{E}_0$ and obtains the initial witness $\mathfrak{t}_0 := (\mathfrak{x}_0, \mathfrak{w}_0)$ such that

$$((\text{cm}_{\text{perm}}, \text{cm}_0), \mathfrak{t}_0) \in \text{InitStmt} .$$

Also, for each $s \in [\ell]$ with $\pi_s \neq \perp$, $\mathbf{E}_{\text{DYN}}$ invokes $\mathbf{E}_s$ and obtains the witness of UpdateStmnt_s :

$$\mathfrak{t}_s := \left(\text{pf}_{\text{upd},s}, \text{pf}_{\text{perm},s}, \text{pf}_{\text{wire},s}, \mathfrak{u}_s = \{(i, \alpha_i, \beta_i, \sigma_i)\}_{i \in U_s} \right) .$$

For $\pi_s = \perp$, $\mathbf{E}_{\text{DYN}}$ sets $\mathfrak{u}_s = \perp$. Finally, $\mathbf{E}_{\text{DYN}}$ applies the extracted update tables in increasing order to obtain $(\mathfrak{x}_\ell, \mathfrak{w}_\ell)$ and outputs $\mathfrak{w}_\ell$. Its circuit size is $O\left(\sum_{s=0}^\ell C_s\right)$, which is the bound in (13).

Knowledge error analysis. We prove by induction on $s \in \{0, \dots, \ell\}$ that, except with probability

$$\sum_{j=0}^s \kappa_j + (3s+1)\epsilon_{\text{spd}}^{\text{vc}} + s\epsilon_{\text{pu}}^{\text{vc}} ,$$

the following properties hold for $\mathbf{z}_s = [\mathfrak{w}_s, \mathfrak{x}_s]$:

1. *Commitment consistency:* $\text{cm}_s = \text{Com}_{\text{pp}}(\mathbf{z}_s)$, and
2. *Plonk membership:* $(\mathfrak{x}_s, \mathfrak{w}_s) \in \mathcal{R}_{\text{Plonk}}(\mathcal{C})$.

Base case. By knowledge soundness of SNARK, except with probability κ_0 , $\mathfrak{t}_0$ is a valid witness of InitStmnt . The initial relation checks $\text{cm}_0 = \text{Com}_{\text{pp}}(\mathbf{z}_0)$, every gate constraint, and every equality $\mathbf{z}_0[i] = \mathbf{z}_0[\sigma_{\mathcal{C}}[i]]$. Since $\sigma_{\mathcal{C}}$ is the circuit's fixed wiring permutation, hardwired during indexing, both properties above hold for $s = 0$.

Inductive case. Fix $s \in [\ell]$ and assume the claim for $s-1$. If $\pi_s = \perp$, the verifier requires $\text{cm}_s = \text{cm}_{s-1}$ and the extracted update is empty, so $\mathbf{z}_s = \mathbf{z}_{s-1}$ and both properties follow immediately. Now, suppose instead that $\pi_s \neq \perp$. Except with probability κ_s , $\mathfrak{t}_s$ is a valid witness of UpdateStmnt_s . Define

$$\mathbf{z}_s[i] := \begin{cases} \beta_i & i \in U_s, \\ \mathbf{z}_{s-1}[i] & i \notin U_s. \end{cases}$$

First, consider commitment consistency. The inductive hypothesis gives $\text{cm}_{s-1} = \text{Com}_{\text{pp}}(\mathbf{z}_{s-1})$. The validity of $\text{pf}_{\text{upd},s}$ gives us $\alpha_i = \mathbf{z}_{s-1}[i]$ for every $i \in U_s$, except with probability $\epsilon_{\text{spd}}^{\text{vc}}$ by strong position binding. The same proof $\text{pf}_{\text{upd},s}$ authenticates the old values under cm_{s-1} and the new values under cm_s . Since $\mathbf{z}_s$ differs from $\mathbf{z}_{s-1}$ only on U_s , strong local updatability (see Definition 3.10) gives us $\text{cm}_s = \text{Com}_{\text{pp}}(\mathbf{z}_s)$ except with probability $\epsilon_{\text{pu}}^{\text{vc}}$.

Next, consider Plonk membership, for which we have to prove gate and wiring constraints for $\mathbf{z}_s$. Validity of U_s means that every gate touched by U_s has its entire three-wire tuple in U_s , and the update relation checks that tuple using the β values. Untouched gates retain their old values and remain valid by the induction hypothesis. Thus, all gate constraints are satisfied by $\mathbf{z}_s$.

For the wiring constraints, strong position binding between the permutation $\sigma_{\mathcal{C}}$ committed during indexing and $\text{pf}_{\text{perm},s}$ gives

$$\sigma_i = \sigma_{\mathcal{C}}[i] \quad \text{for all } i \in U_s , \tag{21}$$

except with probability $\epsilon_{\text{spd}}^{\text{vc}}$. Similarly, strong position binding for $\text{cm}_s = \text{Com}_{\text{pp}}(\mathbf{z}_s)$ and the opening $\text{pf}_{\text{wire},s}$, which authenticates the pairs $\{\sigma_i : \beta_i\}_{i \in U_s}$, gives

$$\beta_i = \mathbf{z}_s[\sigma_i] \quad \text{for all } i \in U_s , \tag{22}$$

except with probability $\epsilon_{\text{spd}}^{\text{vc}}$. Since $z_s[i] = \beta_i$ by definition, (21) and (22) give us

$$z_s[i] = z_s[\sigma_C[i]] \quad \text{for all } i \in U_s. \quad (23)$$

It remains to show that the wiring constraints outside U_s and the ones that cross U_s also hold. Consider any cycle C of the permutation σ_C . By the induction hypothesis, z_{s-1} is constant on C . If $C \subseteq U_s$, then (23) checks every edge of C , so z_s is constant on C . Otherwise, if $C \cap U_s = \emptyset$, then z_s is unchanged from z_{s-1} on C , so it is constant on C . Otherwise, partition the updated positions of C into maximal consecutive segments in U_s . Consider an arbitrary segment S . By (23), all values within S are equal to each other: say this value is v_1 . Furthermore, all the values in $C \setminus U_s$ are equal to each other by the induction hypothesis: say this value is v_2 . If j is the last position of S , then $\sigma_C[j] \in C \setminus U_s$ (by maximality of S), and (23) gives $z_s[j] = z_s[\sigma_C[j]]$, which implies $v_1 = v_2$. Thus, all values in C are equal under z_s , and the wiring constraints hold for this cycle. Because the cycles partition $[m]_0$, all wiring constraints hold, which completes the inductive step.

Finally, let $P = \{6n, \dots, 6n + n_0 - 1\}$. Since the dynamic SNARK verifier accepts, we have

$$\text{VC.Verify}(\text{pp}, \text{cm}_\ell, P, \mathbb{x}, \text{pf}_{\text{pub}}) = 1.$$

The final inductive step also gives us $\text{cm}_\ell = \text{Com}_{\text{pp}}(z_\ell)$, which is equal to $\mathbb{x}_\ell$ on the set P . Strong position binding therefore implies $\mathbb{x} = \mathbb{x}_\ell$ except with probability $\epsilon_{\text{spd}}^{\text{vc}}$. Since $(\mathbb{x}_\ell, \mathbb{w}_\ell) \in \mathcal{R}_{\text{Plonk}}(\mathcal{C})$, the extractor's output satisfies $(\mathbb{x}, \mathbb{w}_\ell) \in \mathcal{R}_{\text{Plonk}}(\mathcal{C})$ with the claimed error bound.

QROM. The knowledge error for extracting all $\ell + 1 = O(\log m)$ witnesses in the ROM can be upper bounded by the sum of their individual extraction errors. However, in the QROM this is not necessarily the case and we need to consider the *joint extraction* failure probability of all $O(\log m)$ witnesses. All $\mathbf{E}_s$ use the common emulated oracle stated in the theorem, shared with the adversary and verifiers, including the extraction access of [CDHZ26, Definition 10.3]. To apply the composition theorem [CDHZ26, Theorem 10.6], start with $(\mathbb{t}_0, \dots, \mathbb{t}_\ell)$, where $\mathbb{t}_0$ is a witness for `InitStmt` and $\mathbb{t}_s$ is a witness for `UpdateStmts` when $\pi_s \neq \perp$. At step s , checking π_s changes the witness tuple from $(\mathbb{t}_s, \dots, \mathbb{t}_\ell)$ to $(\mathbb{t}_{s+1}, \dots, \mathbb{t}_\ell)$, and the checks for $\pi_0, \dots, \pi_s$ must all accept. For $\pi_s \neq \perp$, $\mathbf{E}_s$ recovers $\mathbb{t}_s$ and adds it back to this tuple during extraction. For $\pi_s = \perp$, we only check $\text{cm}_s = \text{cm}_{s-1}$ and set $\mathbb{t}_s = \perp$. The composition theorem therefore gives $\mathbf{E}_{\text{DYN}}$ that recovers $(\mathbb{t}_0, \dots, \mathbb{t}_\ell)$ for the adversary's proof tuple by running $\mathbf{E}_\ell, \mathbf{E}_{\ell-1}, \dots, \mathbf{E}_0$ in that order, while omitting the $\mathbf{E}_s$ calls where $\pi_s = \perp$. Since there are only $\ell + 1 = O(\log m)$ steps, the factors in the error bound and the number of oracle queries remain polynomial, according to [CDHZ26, Theorem 10.6]. Denote the resulting negligible extraction error, including oracle-emulation error, by κ_{joint} . Finally, $\mathbf{E}_{\text{DYN}}$ applies $\mathbb{u}_1, \dots, \mathbb{u}_\ell$ in increasing order as above.

Except with probability κ_{joint} , $\mathbb{t}_0$ is a valid witness for `InitStmt` and every $\mathbb{t}_s$ with $\pi_s \neq \perp$ is a valid witness for `UpdateStmts`, whenever the verifier accepts. The above described reconstruction and induction proof use quantum strong position binding and proof uniqueness, giving knowledge error at most

$$\kappa_{\text{joint}} + (3\ell + 2)\epsilon_{\text{spd}, Q}^{\text{vc}} + \ell\epsilon_{\text{pu}, Q}^{\text{vc}},$$

where all error terms are evaluated at the full quantum adversary sizes. Simulating the verifier and reconstructing the witness still take polynomial time, so the resulting straight-line extractor is quantum polynomial time. $\square$

B Deamortization of update time

We discuss how to modify Fig. 4 to obtain worst-case $O(k \cdot \tau \cdot \log(m/k + 1) + (k/m) \cdot T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0))$ update time, where $\tau := \tau_{\text{upd}}(\lambda, m, n_0)$ is defined in (8).

Unlike the amortized version, where an incoming update triggers a transition proof computation that must be completed immediately, in the deamortized version an update can trigger potentially multiple transition proof computations, with the update prover only making partial progress towards these computations, leaving the remaining progress to when future updates arrive. The new invariant that the construction satisfies is that for each level of the data structure has at most one ongoing proof computation at any given time. However, to make this more formal, we need to discuss what notion of “time” is being used.

Time as total update volume. The data structure described in Step 2 of Fig. 4 stores up to ℓ transition proofs $\pi_1, \dots, \pi_\ell$, each of which can be interpreted as proving the updates in a certain time range. For example, if the first two levels are non-empty, π_1 proves the transition from time 0 to time t_1 , and π_2 proves the transition from time t_1 to time $t_1 + t_2$, where t_j is the total size of the updates represented by π_j . This total size can exceed the number of distinct updated positions when a position is updated more than once. The non-empty proofs, ordered from level 1 to level ℓ , represent the oldest updates first and the newest updates last. Their time ranges partition the timeline from $T_0 = 0$ to the current time $T_r := \sum_{h=1}^r k_h$, where $k_1, \dots, k_r$ are the sizes, after gate closure, of the r updates since initialization. This ensures that no updates are missed or applied twice.

For $0 \leq x \leq T_r$, define

$$\rho(x) := \min\{T_z : z \in \{0, \dots, r\}, T_z \geq x\}.$$

Thus, $\rho(x) = T_h$ when $T_{h-1} < x \leq T_h$, and $\rho(0) = 0$. Every transition proof certifies a time range of the form $[\rho(x), \rho(y)]$.

Time ranges of transition proofs. For a level $j \in [\ell]$, let $b_j = 2^{\ell-j}$, and index the transition proofs by nonnegative integers x . Let $\pi_{j,x}$ be a transition proof that certifies the time range $[\rho(xb_j), \rho((x+1)b_j)]$. The *start threshold* of $\pi_{j,x}$ is $(x+1)b_j$, which is the time at which the computation of $\pi_{j,x}$ begins. Similarly, the *completion threshold* of $\pi_{j,x}$ is $(x+2)b_j$, which is the time at the computation of $\pi_{j,x}$ ends. When $\pi_{j,x}$ finishes, discard it if there exists a completed proof for a longer time range that already covers its time range. Otherwise, $\pi_{j,x}$ replaces all completed transition proofs whose time ranges are subsets of its time range.

Algorithm for handling incoming update. Consider an update of size k that advances time from T_r to $T_{r+1} = T_r + k$, and let $t \in [\ell]$ satisfy $2^{\ell-t} \leq k < 2^{\ell-t+1}$. Proceed as follows:

1. First, finish all proof computations in progress at levels $t, t+1, \dots, \ell$.
2. Let $b_* := 2^{\ell-t}$ and $x_* = \lfloor T_r/b_* \rfloor$, and immediately compute π_{t,x_*} for the time range $[\rho(x_*b_*), \rho((x_*+1)b_*)]$. Since $T_r < (x_*+1)b_* \leq T_{r+1}$, we have $\rho((x_*+1)b_*) = T_{r+1}$, so this proof includes the entire incoming update. The total size of the updates it represents is $T_{r+1} - \rho(x_*b_*) \leq b_* + k < 3b_*$.
3. For each level $j < t$, let $b_j = 2^{\ell-j}$ and start computing $\pi_{j,x}$ when the time first reaches or exceeds its start threshold $(x+1)b_j$. Perform $c\tau$ operations per unit of time from its start threshold until its completion threshold $(x+2)b_j$. Since $\rho(xb_j) \geq xb_j$ and $\rho((x+1)b_j) < (x+1)b_j + k$, and $k < b_j$, the total size of the updates represented by the new proof is $\rho((x+1)b_j) - \rho(xb_j) < 2b_j$.

Update time. The algorithm starts new proof computations only at levels $j \leq t$, where $k < 2b_j$, so each proof represents fewer than $3b_j$ updated positions and takes at most $cb_j\tau$ time to compute for sufficiently large c . Finishing pending computations at levels $t, \dots, \ell$ and computing $\pi_{t,x*}$ costs $O(\tau \sum_{j=t}^{\ell} 2^{\ell-j}) = O(k\tau)$. Each of the remaining $t - 1 = O(\log(m/k + 1))$ levels receives at most $ck\tau$ work, giving worst-case $O(k\tau \log(m/k + 1))$ update time, excluding reinitialization.

Proof size. Fix $j > 1$, and let $e = (x + 1)b_j$ be the start threshold of $\pi_{j,x}$. The proof $\pi_{j-1, \lfloor x/2 \rfloor}$ covers its time range and has completion threshold $2b_j(\lfloor x/2 \rfloor + 2) \leq e + 3b_j$. After an update reaches this threshold, the algorithm has completed either this covering proof or a proof for an even larger transition, so $\pi_{j,x}$ is no longer retained. Consequently, every retained completed proof satisfies

$$T_r - 3b_j < e \leq T_r .$$

The start thresholds are multiples of b_j , so there are at most three possibilities. Level 1 has the same bound since $T_r < 2m < 4 \cdot 2^{\ell-1}$ between reinitialization boundaries, allowing only three start thresholds at that level. Only completed proofs are included in the proof object, so its size changes by only a constant factor.

Reinitialization. Finally, when time first reaches or exceeds m , retain the completed transition proofs, discard the unfinished computations, and start computing an initial proof for the current instance-witness pair. Process subsequent updates in a fresh data structure starting from this pair, with time reset to $T_0 = 0$ and ρ defined using its update times, and allocate $O(k \cdot T_{\mathcal{P}_{\text{init}}}(\lambda, n, n_0)/m)$ work per update to the initial proof. This proof finishes when the new data structure reaches its own reinitialization boundary, at which point it replaces the earlier initial proof and the preceding sequence of transition proofs. Thus, at most two data structures are retained, so the proof contains at most six transition proofs per level. This gives the bound in (9) in the worst-case.

Verifier modification. The verifier processes the multiple completed transition proofs chronologically at each level, using each proof's level-specific verification key and consecutive commitments as public inputs. Clearly, the verification time, proof length, and knowledge-error bounds incur a constant factor overhead.

C Zero-knowledge compiler

We sketch how to obtain *weak zero-knowledge* for our compiler. After any polynomially bounded sequence of valid updates, the final proof Π_T should be computationally indistinguishable from a simulated proof generated using only the circuit index and the final public instance $\mathbf{x}_T$, and a simulation trapdoor. The guarantee concerns a distinguisher which receives only the final proof, not the entire sequence of proofs. A zero-knowledge definition that considers the entire sequence is discussed later in Remark C.1.

Ingredients. Our compiler takes as input a VC scheme and a static SNARK. We require the VC scheme to be hiding, and for the static SNARK to be jointly zero-knowledge for polynomially many adaptively chosen statements under the same setup. We assume that the VC scheme remains hiding even given the SNARK simulation trapdoor. For the Merkle instantiation, the prover can use the original Merkle trees internally and publish randomized hiding and binding commitments to their roots. The true roots (unrandomized), commitment randomness, and Merkle authentication paths become private witnesses to the static SNARKs. We also use the unbalanced Merkle trees

described in Corollary 4.2: the two children of the roots are roots of the private witness and public input respectively.

Modifying the compiler. The primary modification we make to our compiler is that there are no empty levels: each level always contains a static SNARK proof, and we replace every $\pi_j = \perp$ by a *dummy proof*. At level j , the modified update relation proves either a valid update of size at most q_j (as defined in (4)), using the original update checks on the Merkle roots, or that the two unrandomized roots are equal and hence represent the same state. The modified initial relation similarly checks an opening of the randomized commitment in addition to the original initial checks. Each level uses independent commitment randomness, even when adjacent levels represent the same state, so that the dummy levels cannot be identified by the distinguisher. Finally, we replace the public-input opening pf_{pub} by a zero-knowledge proof that the final randomized commitment opens to a Merkle root whose public-input positions contain $\mathbf{x}_T$. This keeps the witness-dependent root in the original public-input opening private. Thus, the overall resulting proof has a fixed format, with no public indication of which levels are dummy. This modification does not change any of the asymptotic efficiency metrics in Corollary 4.2.

Simulation argument. The simulator for the dynamic SNARK $\mathcal{S}$ generates the static SNARK setup with its simulation trapdoor and derives the keys for the modified relations. Given instance $\mathbf{x}_T$, $\mathcal{S}$ forms an arbitrary dummy assignment $\mathbf{z}^* = [0^{6n}, \mathbf{x}_T]$ and computes its Merkle root. $\mathcal{S}$ commits to this root at every level of the data structure with independent randomness at each level. It also invokes the static SNARK simulator to produce the initial proof, every level proof, and the proof of the final public-input opening, using their respective public statements. $\mathcal{S}$ outputs these commitments and simulated proofs in the fixed format defined above.

We use a hybrid argument to justify that this simulation is indistinguishable from a real proof. Starting with the real proof, first replace all static proofs by simulated proofs for their original, true statements. Joint zero-knowledge makes this change indistinguishable. Next, replace the commitments to the actual roots, one at a time, by independently randomized commitments to the dummy root, regenerating the simulated proofs for the resulting public statements. Commitment hiding makes these replacements indistinguishable, since all roots and authentication paths are now hidden inside simulated proofs. After making all of these replacements, the resulting distribution is exactly the simulator's output. This gives an informal argument for weak zero-knowledge of the modified compiler.

Remark C.1 (Weak versus strong zero-knowledge). A stronger notion requires the simulator to jointly simulate a proof sequence $(\Pi_s)_{s=0}^T$ given only the corresponding instance sequence $(\mathbf{x}_s)_{s=0}^T$, as considered by Wang et al. [WSP26b]. We only claim the weak notion above: comparing successive proofs can reveal which levels were refreshed and hence information about update sizes. In particular, it seems difficult for our construction to achieve this stronger notion. However, in a real-world deployment, observing the prover's running time can also reveal information about the distance between consecutive instance-witness pairs, even when the proof system satisfies the stronger notion of zero-knowledge. This side-channel motivates considering the weak notion when the update time is observable.