SnarkMux: Streaming SNARK Aggregation with Constant Online Verification

No Author Given

No Institute Given

Abstract. SNARKs let a verifier check an expensive computation via a succinct proof, without re-executing the computation. However, many applications still verify proofs individually, creating verifier load and high fees in replicated or constrained-verifier settings. Existing proof aggregation schemes largely follow the generalized inner product argument (GIPA) line, achieving $O(\log n)$ proof size and verifier work for aggregating n proofs, but remaining costly in practice and operating in batch mode. Rather than batching until a full window is available, we present SnarkMux, a streaming SNARK aggregation scheme that maintains a constant-size aggregate and requires $O(1)$ *online* verifier work per window of size n. Motivated by incremental verifiable computation, SnarkMux directly folds the Groth16 and Plonk verification relations and composes these proof-system-specific folds with Nova's circuit-layer fold; this defers expensive target-verifier logic (*e.g.*, pairings) to the final check while keeping constant aggregator state. As part of public-input aggregation, we propose a generic instance-delegation technique: the prover aggregates instances, and the verifier performs a simple binding preprocessing costing only $O(n)$ hashes per window, after which online verification is constant-cost. Notably, our Groth16 and Plonk instantiations cover both circuit-specific and universal-setup SNARKs, with preprocessing that makes Plonk challenges folding-compatible, leading to a 0.8 KB Groth16 aggregate with 18 ms verification and a 0.5 KB Plonk one with 2 ms verification for 256 proofs.

Keywords: Proof Aggregation · Incrementally Verifiable Computation · Folding Scheme.

1 Introduction

Succinct non-interactive argument of knowledge (SNARK) lets a prover convince a verifier that a statement is true using a short proof and efficient verification, without interaction. A zk-SNARK additionally provides zero-knowledge, meaning the proof reveals nothing beyond validity. SNARK verification is increasingly deployed in security-critical systems with constrained or replicated verifiers, where verification cost directly limits throughput and usability. In blockchains, zk-rollups submit proofs to a layer-1 chain to attest to off-chain state transitions [26], and zkEVM systems prove correct smart-contract execution [21]. Despite this promise, deployed SNARK-based systems face practical resource

constraints, because on-chain proof verification consumes scarce execution and storage resources and is priced accordingly via transaction fees. For instance, a recent study reports that a Tornado Cash deposit can cost on the order of tens of USD in gas under typical price conditions, illustrating how verification-heavy workloads translate into user-visible fees [27].

A proof-aggregation scheme mitigates this issue by offloading certain tasks to an aggregator with powerful hardware. Specifically, it operates on a *window* of n instance-proof pairs $(u_i, \pi_i)_{i=1}^n$ and outputs an aggregate instance u^*, an *aggregate proof* π^*, and an *aggregation proof* π_{AGG}.[1] If the verifier accepts $(u^*, \pi^*, \pi_{\mathsf{AGG}})$, then all n original statements are valid. In our setting, proofs arrive as an unbounded stream, so once n proofs are accumulated, the aggregator finalizes the window and sends $(\pi^*, \pi_{\mathsf{AGG}})$ to the verifier. The verifier can compute the folded instance u^* locally from $(u_i)_{i=1}^n$; π^* is then verified against this verifier-derived u^*.

A large class of practical constructions follows the generalized inner product argument (GIPA) line [8,13]. These schemes achieve $O(\log n)$-size aggregates via Bulletproofs-style compression [4] and Kate–Zaverucha–Goldberg (KZG) polynomial commitments [17]. As representative GIPA-based baselines, we compare against TIPP [8], SnarkPack [13], and aPlonk [1]. However, verification can remain costly in constrained-verifier deployments because it entails multiple pairing checks (*e.g.*, Ethereum charges $45{,}000 + 34{,}000 \cdot k$ gas for k pairings [9]). Moreover, these schemes are typically batch-oriented: the aggregator must first receive and store the full window before producing the aggregate, incurring $O(n)$ space and added latency under asynchronous arrivals.

Our goal is to design a prover-side *streaming* aggregator that processes proofs incrementally and maintains only an $O(1)$-size state. We realize streaming aggregation as a recursive process, instantiated via incrementally verifiable computation (IVC), so the aggregator updates a constant-size state per incoming proof. To reduce verifier online work, we delegate folding $(u_i)_{i=1}^n$ into u^* to the aggregator, and the *aggregated proof* π^* attests to the validity of u^*. However, without additionally binding the intended window, a malicious aggregator/prover can replace, reorder, or otherwise substitute elements of $(u_i)_{i=1}^n$ (*e.g.*, swapping an unsatisfied instance with a trivially satisfied one) while still producing an accepting aggregate for the substituted window. A verifier that performs only $O(1)$ online work would be unable to detect such instance substitution without reading $(u_i)_{i=1}^n$. To address this, we propose a preprocessing/online split. For each window, the verifier computes a binding claim $h \leftarrow \mathrm{Hash}\big((u_i)_{i=1}^n\big)$ during preprocessing, and later performs only $O(1)$ online work to check a constant-size aggregate against the prover-side claim carried in the *aggregation proof* π_{AGG} (which commits to the same h). This claim binds the window's contents and order by Lemma 2 during the preprocessing step, so any successful substitution would imply a hash collision, except with negligible probability. These guaran-

[1] π^* is the compact proof object verified against u^* (in place of verifying all (u_i, π_i) individually), and π_{AGG} certifies that (u^*, π^*) is the correct aggregation of the ordered window $(u_i, \pi_i)_{i=1}^n$ under the prescribed rules.

Table 1: Proof-aggregation complexity for a window of n proofs

Scheme	Size	Memory	Prove	Verifier
TIPP [8]	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n) + O(n)\,\mathbb{G}_1$
SnarkPack [13]	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n) + O(n)\,\mathbb{Z}_p$
aPlonk [1]	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n) + O(n)\,\mathbb{Z}_p$
Mira [2]	$O(1)$	$O(1)$	$O(n)$	VC update, $O(1)$
SnarkMux	$O(1)$	$O(1)$	$O(n)$	$O(n)\,\mathsf{H}, O(1)$

– Verifier cost for SnarkMux is reported as one-time instance binding followed by online aggregate checking. The term $O(n)\,\mathsf{H}$ is the one-time hash binding over $(u_i)_{i=1}^n$, and the term $O(1)$ is the subsequent online checking cost.
– Verifier cost of GIPA-style baselines (TIPP, SnarkPack, and aPlonk) includes proof verification plus verifier-side aggregation of the n public instances. $O(\log n)$ term: proof verification; $O(n)$ term: instance aggregation in $\mathbb{G}_1/\mathbb{Z}_p$. Mira instead maintains an updatable vector commitment VC to instances and verification keys; the update cost depends on the chosen commitment (its paper suggests a Merkle tree), followed by a constant-size final PCD proof.
– $\mathbb{G}_1$: scalar multiplications, $\mathbb{Z}_p$: field operations, and H: hash evaluations.

tees hold up to the recursion horizon of the underlying instantiation (*i.e.*, the maximum number of proofs that can be aggregated in a window), and we discuss scaling beyond this horizon in Appendix D.

We present SnarkMux, which achieves these tradeoffs via an IVC-based, two-folding design. Table 1 summarizes the resulting tradeoffs relative to representative GIPA-based baselines. In Table 1, verifier cost is reported as (one-time instance binding, online aggregate checking).

1.1 Our Contributions

Direct target-relation folding. We construct folding relations directly for the verification equations of Groth16 and Plonk, rather than using Nova with the step function $F = \mathsf{SNARK.V}$. Nova's relaxed-R1CS fold remains the circuit layer that certifies each step; our target-SNARK fold is the additional, proof-system-specific layer that absorbs each incoming proof into a constant-size target state. This two-folding structure maintains a constant-size aggregate and requires $O(1)$ *online* verifier work per window (for n within the supported recursion horizon; see Table 1).

Preprocessed verification via instance binding and delegation. We introduce a verifier preprocessing/online split for proof aggregation. For each window, the verifier hashes the ordered instance sequence into a binding claim (costing $O(n)$ hashes) and then performs only constant-cost online checking of the reported aggregate. This instance-binding step shifts public-input aggregation from the verifier to the aggregator/prover and prevents instance substitution attacks.

Streaming aggregation with $O(1)$ aggregator state. We give a streaming aggregator that processes proofs one by one and maintains only constant-size state. This avoids the $O(n)$ batch memory footprint and batch-induced latency of GIPA-style aggregation, at the cost of extra aggregator/prover work.

Instantiations for Groth16 and Plonk. We instantiate SnarkMux for Groth16 [16] and Plonk [12], covering both circuit-specific and universal-setup settings and stressing distinct obstacles (pairing-based verification equations versus transcript-derived challenges). For Groth16, we define a relaxed proof relation that supports folding under our verification equation. For Plonk, we give an in-circuit preprocessing step that makes transcript-derived challenges compatible with folding, and we present two instantiations that trade recursive-circuit size against algebraic degree.

Evaluation. We implement SnarkMux and compare it with representative GIPA-based aggregators. In our prototype, SnarkMux aggregates 256 Plonk proofs into a constant-size proof (0.5–1.74 KB depending on the instantiation) with a 2 ms *online* verifier, compared to 10 KB and 17 ms for aPlonk. In streaming settings, we additionally observe up to a 42% reduction in end-to-end latency, reflecting a tradeoff that shifts work from the verifier to the aggregator/prover.

1.2 Technical Overview

$O(1)$ online verification via IVC. Verifier cost is a primary bottleneck for constrained-verifier deployments such as on-chain aggregation. In GIPA-based aggregation, proof verification grows with the window size, and the verifier also processes the public instances. We instead realize aggregation as an n-step IVC and verify only a constant final proof online. For n proofs, online verifier work is therefore constant in n after preprocessing; for Groth16, the final target check still retains its dependence on the public-input length ℓ.

Instance delegation via binding preprocessing. A verifier that computes the folded instance u^* from $(u_i)_{i=1}^n$ must itself process all n public instances. We delegate this computation to the aggregator, but delegation alone would allow a malicious aggregator to substitute a different window. The verifier therefore preprocesses the intended ordered window into a binding claim using $O(n)$ hashes. The recursion maintains the same hash chain, so online verification only checks that the prover-side claim equals the verifier-preprocessed claim, together with the constant-size aggregation proof.

Streaming aggregation with $O(1)$ aggregator state. Batch aggregation requires collecting and storing all n proofs before producing an output, incurring $O(n)$ memory and batch-induced waiting time. In SnarkMux, step i consumes the newly arrived (u_i, π_i), updates the running target-layer and circuit-layer states, and allows previous raw instance–proof pairs to be discarded. Thus, the aggregator maintains only constant-size state in n, while total aggregation work remains linear in the window size.

Two-folding avoids full in-circuit target verification. A direct IVC baseline sets the step function to $F = \mathsf{SNARK.V}$ and therefore embeds the full target

verifier inside the recursive circuit, which is particularly expensive for pairing-based SNARKs [25]. Nova's relaxed-R1CS fold avoids recursively verifying the previous IVC proof, but by itself does not remove the cost of a step function whose job is to verify a fresh SNARK proof [19]. SnarkMux instead adds a proof-system-specific target fold: the Groth16 or Plonk verification relation is folded directly, while Nova's relaxed-R1CS fold certifies the full IVC process. The recursive circuit therefore does not re-run the full target verifier at every arrival; expensive target-verifier logic, in particular the pairing checks, is deferred to the final target-layer check.

Instantiations. For Groth16, naïve random-combination folding does not preserve the pairing verification equation because of cross terms. We therefore define relaxed relations that absorb these terms, including augmented and committed variants that trade target-layer work against recursive-circuit cost. For Plonk, the verification relation depends on transcript-derived Fiat–Shamir challenges. Before each fresh proof enters the folding process, the recursive circuit re-derives and checks its transcript and converts it to a folding-compatible relation. The linear variant supports k-folding but performs more group work in-circuit; the quadratic variant reduces recursive-circuit group work at the cost of a cross term, 2-folding, and higher local prover work.

1.3 Related Work

Proof aggregation. Some work designs cryptographic accumulators for aggregating specialized proofs, such as (non-)membership proofs. Srinivasan *et al.* [24] propose zero-knowledge (non-)membership proof aggregation in bilinear groups. However, this line is not directly suitable for aggregating general-purpose SNARK proofs such as Groth16 and Plonk. Bunz *et al.* [8] adopt GIPA for Groth16 proof aggregation. Their approach reduces verifying n Groth16 proofs to verifying a generalized inner product argument, and applies Bulletproofs-style folding to obtain logarithmic-size proofs. They further introduce an inner pairing product (TIPP) relation and use KZG polynomial commitments to reduce verifier cost. SnarkPack [13] optimizes this approach for aggregating Groth16 proofs and reuses the public parameters from the trusted setup. aPlonk [1] extends related techniques to Plonk aggregation with logarithmic proof size and verification time. We are not aware of any prior pairing-based aggregation interfaces that expose an explicit verifier preprocessing/online split tied to an ordered instance-binding claim, which is the mechanism we use to enable verification.

Another approach for Groth16 proof aggregation relies on recursive composition. Bowe *et al.* [3] construct an additional SNARK for n copies of the Groth16 verifier circuit, but embedding pairings in-circuit incurs high cost [8]. Concurrent work Mira [2] gives a generic Protostar-based compiler for pairing-based arguments and a constant-size Groth16 aggregation protocol, including aggregation across different verification keys. It binds the history of instances and keys with an updatable vector commitment (for example, a Merkle tree). For circuit-specific Groth16, its relaxed relation corresponds, up to encoding, to our basic

relation in Definition 1. Our additional augmented relation reduces the directly comparable target-layer count from $4n$ local pairings to $2n$ local pairings plus two deferred final pairings, and the committed variant removes ℓ-dependent updates from the recursive circuit; neither removes the final Groth16 input check's dependence on ℓ. Our interface instead exposes the verifier's ordered-window binding as a separate VPrep phase followed by a constant-in-n online check, and our construction scope additionally includes two Plonk relations. These are symbolic relation-level distinctions; we do not claim a cross-implementation timing advantage over Mira.

Garuda and Pari [10] are promising further targets, but fewer Fiat–Shamir challenges and a linearly verifiable proof do not alone yield a folding scheme. An extension would still require a closed low-degree relation, in-circuit certification of its transcript, and a fold-soundness proof; we leave these steps open.

Folding schemes. Traditional recursive SNARKs embed a SNARK verifier inside the circuit, which requires implementing full verification logic at every step. Accumulations [7,6] and folding schemes [19,28], provide a more efficient route to recursion. An accumulation/folding scheme combines multiple instances into a single folded instance. If the folded instance is valid, then all folded instances are correct. In the recursive circuit, each step updates a running folded instance and outputs an auxiliary circuit instance attesting that the step was executed correctly. A final verification of the running instance and the auxiliary instance suffices to validate all iterations.

BCMS20 [7] constructs Proof-Carrying Data (PCD, a generalized version of IVC) by folding KZG evaluations. Specifically, each iteration converts the recursive circuit into KZG evaluation proofs via Plonk and subsequently folds these proofs. While their KZG folding technique shares similarities with our "Preprocessed Linear Plonk Relation" (Definition 5), it remains fundamentally distinct from SnarkMux. First, BCMS20 integrates Plonk/KZG folding directly into the PCD framework (*i.e.*, $\mathsf{Fold.V}$), whereas in SnarkMux, this logic resides within the step function F. This architectural difference is significant: to aggregate Plonk proofs using BCMS20, one would still need to implement separate aggregation logic inside F. Furthermore, BCMS20 is inherently less efficient than Nova-based IVC [19]. Unlike Nova, which recurses at the R1CS layer, BCMS20 operates at the polynomial layer. This requires an expensive conversion from R1CS/Plonkish arithmetization to KZG evaluations at every step, leading to higher recursive overhead.

2 Preliminaries

2.1 Notation

We use security parameter λ and negligible function $\mathsf{negl}(\lambda)$. Let $\mathbb{Z}_p$ be a prime field. Let $\mathbb{Z}_p^{<d}[X]$ (resp., $\mathbb{Z}_p^{d}[X]$) be the set of univariate polynomials over $\mathbb{Z}_p$ of degree $<d$ (resp., $\leq d$). We write $r \leftarrow\!\!\$\ \mathbb{Z}_p$ for uniform sampling and $x \leftarrow a$ for assignment. We abbreviate $(a_1, \ldots, a_n)$ as $(a_i)_{i=1}^n$.

Relations. An $\mathcal{NP}$ relation $\mathcal{R}$ is defined over tuples $(\mathsf{pp}, \mathsf{s}, u, w)$, where pp are public parameters, s is a fixed structure shared across instances (*e.g.*, a circuit or statement shape), u is an instance, and w is a witness. We write $(\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}$.

Bounded window size. We use n for the number of proofs in an aggregation window. Our security statements are parameterized by a maximum supported window size $N_{\max}$, *i.e.*, $n \leq N_{\max}$. In SnarkMux, $N_{\max}$ equals the supported recursion length of the chosen folding-based IVC instantiation. In our prototype, we set $N_{\max} = 256$. All security experiments quantify over $n \leq N_{\max}$. As is standard for IVC, knowledge extraction may take time polynomial in λ and exponential in n (equivalently, polynomial in λ and 2^n). Accordingly, our guarantees should be read as holding for any fixed $n \leq N_{\max}$, reflecting an inherent limitation from IVC.

2.2 (zk-)SNARKs and Proof Aggregation

(zk-)SNARK syntax. Let $\mathcal{R}$ be a $\mathcal{NP}$ relation. A SNARK is described by four algorithms below.

• **Parameter generation** ($\mathsf{SNARK.G}$): $\mathsf{pp} \leftarrow \mathsf{SNARK.G}(1^\lambda)$: Given security parameter λ, sample public parameters pp.

• **Key (CRS/SRS) generation**[2] ($\mathsf{SNARK.K}$): $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{SNARK.K}(\mathsf{pp}, \mathsf{s})$: Taking the public parameters pp and a structure s, generate the prover's proving key pk and the verifier's verification key vk.

• **Proving** ($\mathsf{SNARK.P}$): $\pi \leftarrow \mathsf{SNARK.P}(\mathsf{pk}, u, w)$: Given an instance-witness pair (u, w), output a succinct proof π under pk that $(\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}$.

• **Verification** ($\mathsf{SNARK.V}$): $0/1 \leftarrow \mathsf{SNARK.V}(\mathsf{vk}, u, \pi)$: Given an instance u and the corresponding proof π, output either 1 by accepting the proof or 0 by rejecting it.

A SNARK satisfies *completeness*, *knowledge soundness*, and *succinctness* properties. Zk-SNARK is a variant of SNARK with *zero-knowledge* property: the proof π reveals nothing about w (formally defined in Appendix A.1).

Proof aggregation syntax. Given a SNARK system SNARK and a window of n instance-proof pairs $(u_i, \pi_i)_{i=1}^n$, we call AGG a proof aggregation scheme defined as follows.

• **Parameter generation** ($\mathsf{AGG.G}$): $\mathsf{pp} \leftarrow \mathsf{AGG.G}(1^\lambda)$ generates public parameters pp from the security parameter λ.

• **Key generation** ($\mathsf{AGG.K}$): $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{AGG.K}(\mathsf{pp})$ generates the proving key pk and verification key vk. If AGG uses a (potentially different) SNARK as a building block, we treat its structure s' as an additional input, *i.e.*, $\mathsf{AGG.K}(\mathsf{pp}, \mathsf{s}')$.

• **Aggregation** ($\mathsf{AGG.P}$): $((u^*, \pi^*), \pi_{\mathsf{AGG}}) \leftarrow \mathsf{AGG.P}\big(\mathsf{pk}, (u_i, \pi_i)_{i=1}^n\big)$ outputs a folded target pair and an aggregation proof: the *aggregate proof* π^* is checked

[2] We follow Nova [19] in viewing $(\mathsf{pk}, \mathsf{vk})$ as public key material, *i.e.*, the common reference string (CRS) or structured reference string (SRS) for the relevant interfaces. The same convention applies to proof aggregation.

by the selected aggregate target-relation verifier $\mathsf{V}_{\mathsf{tgt}}$ against the folded instance u^*; for SnarkMux, $\mathsf{V}_{\mathsf{tgt}}$ is the relaxed Groth16 or Plonk relation check, while $\mathsf{SNARK.V}$ remains the original verifier applied to extracted fresh proofs; and the *aggregation proof* π_{AGG} certifies that π^* is the correct aggregation of the ordered window $(u_i, \pi_i)_{i=1}^n$ under the prescribed streaming update rules.

• **Aggregate verification** ($\mathsf{AGG.V}$): $0/1 \leftarrow \mathsf{AGG.V}\big(\mathsf{vk}, n, (u_i)_{i=1}^n, (u^*, \pi^*), \pi_{\mathsf{AGG}}\big)$ accepts iff $\big((u^*, \pi^*), \pi_{\mathsf{AGG}}\big)$ is valid for the claimed instances.

We first give a generic (batch-style) interface. Later, our construction instantiates a verifier with a preprocessing/online split. A proof aggregation scheme requires perfect completeness and knowledge soundness (Appendix A.2).

2.3 IVC and Folding

IVC syntax. IVC proves the correctness of an iterative computation $z_i = F(z_{i-1}, \omega_i)$, where z_i is the step-i state and ω_i is the step-i auxiliary input. It should be complete, knowledge sound, and succinct (Appendix A.3).

• **Parameter generation** ($\mathsf{IVC.G}$): $\mathsf{pp} \leftarrow \mathsf{IVC.G}(1^\lambda)$ generates public parameters.

• **Key generation** ($\mathsf{IVC.K}$): $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{IVC.K}(\mathsf{pp}, F)$ generates key material specialized to the step function F.

• **Step proving** ($\mathsf{IVC.P}$): $\Pi_i \leftarrow \mathsf{IVC.P}(\mathsf{pk}, i, z_0, z_i, z_{i-1}, \omega_{i-1}, \Pi_{i-1})$ outputs a proof Π_i for the i-step execution.

• **Final verification** ($\mathsf{IVC.V}$): $0/1 \leftarrow \mathsf{IVC.V}(\mathsf{vk}, i, z_0, z_i, \Pi_i)$ accepts iff Π_i certifies the execution from z_0 to z_i.

Folding scheme syntax. A folding scheme reduces verification of two instances of a relation to verification of one folded instance of the same size. It should be complete and knowledge sound (Appendix A.4).

• **Parameter generation** ($\mathsf{Fold.G}$): $\mathsf{pp} \leftarrow \mathsf{Fold.G}(1^\lambda)$ generates public parameters.

• **Key generation** ($\mathsf{Fold.K}$): $(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Fold.K}(\mathsf{pp}, \mathsf{s})$ generates key material for folding instances that share structure s.

• **Folding (prover)** ($\mathsf{Fold.P}$): $(u, w) \leftarrow \mathsf{Fold.P}(\mathsf{pk}, (u_1, w_1), (u_2, w_2))$ outputs a folded instance-witness pair.

• **Folded instance computation (verifier-side)** ($\mathsf{Fold.V}$): $u \leftarrow \mathsf{Fold.V}(\mathsf{vk}, u_1, u_2)$ computes the folded instance. This is a deterministic computation (not a validity check), and soundness is defined for the overall folding relation.

3 SnarkMux

We present SnarkMux, a proof aggregation scheme built from folding-based IVC. We call the proof-system-specific Groth16/Plonk relation the *target (SNARK)*

layer; $\mathsf{Fold}_{\mathsf{SNARK}}$ folds this relation and $\mathsf{V}_{\mathsf{tgt}}$ checks its final folded pair. This final relation check is distinct from the original $\mathsf{SNARK.V}$, which checks each fresh constituent proof. The *circuit layer* is Nova's relaxed-R1CS relation; $\mathsf{Fold}_{\mathsf{R1CS}}$ certifies that each recursive step performed the target-fold and hash updates correctly. The aggregator processes instance-proof pairs (u_i, π_i) one by one and maintains a constant-size (aggregation) state. For a window of n proofs (bounded by the recursion horizon), it outputs a constant-size aggregate output $(u^*, \pi^*, \pi_{\mathsf{AGG}})$, where (u^*, π^*) is the aggregate instance-proof pair and π_{AGG} is an aggregation proof. We refer to $(\pi^*, \pi_{\mathsf{AGG}})$ as the aggregate proof package. The verifier may precompute a binding claim by hashing the window's instance sequence once and then perform only a constant-cost online check. Looking ahead, we will instantiate SnarkMux for Groth16 and Plonk, stressing distinct obstacles (pairing equations versus transcript-derived challenges).

3.1 A Baseline: Proof Aggregation as IVC

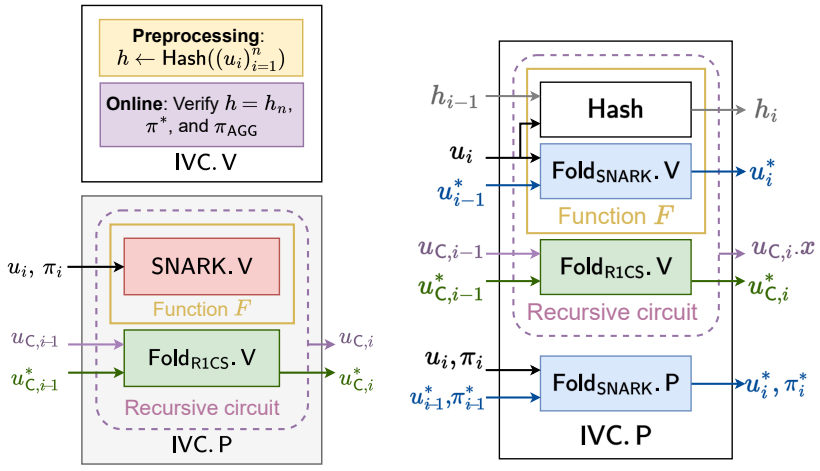
A natural baseline models aggregation as an n-step IVC where each step verifies one incoming SNARK proof. At step i, the step function takes (u_i, π_i) and checks $\mathsf{SNARK.V}(u_i, \pi_i) = 1$, so the final IVC proof implies that all n proofs are valid.

This interface is attractive because the verifier checks only the final IVC proof. However, this particular choice $F = \mathsf{SNARK.V}$ is typically impractical because the recursive circuit must implement the target verifier $\mathsf{SNARK.V}$ (Figure 1a, grey box), which is expensive for pairing-based SNARKs (*e.g.*, Groth16, due to non-native pairings; and Plonk, due to both pairing and transcript-derived Fiat–Shamir challenges). These costs dominate prover time and circuit size, negating the intended benefits of building aggregation from IVC.

This motivates our design: SnarkMux avoids running $\mathsf{SNARK.V}$ inside recursion. Instead, each step proves the correctness of a lightweight folding update that incrementally incorporates (u_i, π_i) into a running state. The next subsection presents the resulting two-folding IVC structure.

3.2 IVC Design

Figure 1 contrasts two ways to realize streaming aggregation as an n-step IVC. In the baseline (Figure 1a, grey box), step i consumes (u_i, π_i) and the circuit re-runs $\mathsf{SNARK.V}$ to validate π_i, which is prover-expensive when $\mathsf{SNARK.V}$ involves pairings or transcript-derived checks. In SnarkMux (Figure 1b, white boxes), step i first certifies any proof-system-specific conversion of the fresh pair (u_i, π_i) to its target relation (notably $\mathsf{Preprocess}^{\mathrm{lin}}$ or $\mathsf{Preprocess}^{\mathrm{quad}}$ for Plonk), then updates a constant-size running state: it (i) updates a binding hash $h_i \leftarrow \mathrm{Hash}(u_i, h_{i-1})$, (ii) folds the running SNARK-layer instance u^*_{i-1} with u_i using $\mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{V}$, and (iii) folds the running circuit-layer instance $u^*_{C,i-1}$ using $\mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{V}$. Crucially, the circuit performs only these deterministic update rules, while the prover updates the corresponding witnesses $w^*_{C,i-1}$ and the running proof object π^*_{i-1}



(a) Straightforward design (grey box)

(b) Two-folding design (white boxes)

Fig. 1: Per-step view of proof aggregation as an n-step IVC: (a) Baseline: the recursive circuit runs SNARK.V on (u_i, π_i). (b) SnarkMux: the recursive circuit certifies fresh-input conversion and deterministic fold updates, while the prover updates proof objects outside the circuit (in-circuit step-relation view)

outside the circuit using $\mathsf{Fold}_{\mathsf{R1CS}}$.P and $\mathsf{Fold}_{\mathsf{SNARK}}$.P. Thus, aggregation is incremental (one update per arrival) and the state remains $O(1)$ throughout the window. The next step takes $(h_i, u^*_i, u_{C,i}, u^*_{C,i})$ (and the prover-held $(\pi^*_i, w_{C,i}, w^*_{C,i})$) as its input; no prior (u_j, π_j) are stored. Throughout, $\mathrm{Hash}(u_i, h_{i-1})$ abbreviates a domain-separated hash of the index, an unambiguous fixed encoding of u_i, and h_{i-1}; the index and fixed window length prevent ambiguity between differently parsed sequences.

We express the logic of the recursive circuit as follows:

$$\begin{aligned} h_i &\leftarrow \mathrm{Hash}(u_i, h_{i-1}), \\ u^*_i &\leftarrow \mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{V}(\mathsf{vk}_{\mathsf{FS}}, u_i, u^*_{i-1}), \\ u^*_{\mathsf{C},i} &\leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{V}(\mathsf{vk}_{\mathsf{FC}}, u_{\mathsf{C},i-1}, u^*_{\mathsf{C},i-1}). \end{aligned} \tag{1}$$

The recursive circuit then produces an instance-witness pair $(u_{\mathsf{C},i}, w_{\mathsf{C},i})$ attesting that Equation (1) was executed correctly.

In parallel, the prover performs the corresponding prover-side folding updates and computes the same h_i locally:

$$\begin{aligned} h_i &\leftarrow \mathrm{Hash}(u_i, h_{i-1}), \\ (u^*_i, \pi^*_i) &\leftarrow \mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{P}\big(\mathsf{pk}_{\mathsf{FS}}, (u_i, \pi_i), (u^*_{i-1}, \pi^*_{i-1})\big), \\ (u^*_{\mathsf{C},i}, w^*_{\mathsf{C},i}) &\leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{P}\big(\mathsf{pk}_{\mathsf{FC}}, (u_{\mathsf{C},i-1}, w_{\mathsf{C},i-1}), (u^*_{\mathsf{C},i-1}, w^*_{\mathsf{C},i-1})\big). \end{aligned} \tag{2}$$

The above construction has a subtle issue. The values h_i, u_i^*, and $u_{\mathsf{C},i}^*$ are outputs of the recursive circuit, so correctness requires them to be exposed as part of the circuit instance $u_{\mathsf{C},i}$ (we denote its public input by $u_{\mathsf{C},i}.x$). However, if $u_{\mathsf{C},i}.x$ explicitly contains $(h_i, u_i^*, u_{\mathsf{C},i}^*)$, then the running instance $u_{\mathsf{C},i}^*$ has a different structure from the fresh instance $u_{\mathsf{C},i}$, and the next fold is ill-typed. To fix this mismatch, we expose only a hash digest of the cross-step values as Nova [19]. Specifically, we set $u_{\mathsf{C},i}.x \leftarrow \mathrm{Hash}(\mathsf{vk}, i, h_i, u_i^*, u_{\mathsf{C},i}^*)$, so all circuit instances share the same public-input structure across steps. Accordingly, the recursive circuit interface is defined as $u_{\mathsf{C},i}.x \leftarrow \mathsf{RC}(\mathsf{vk}, i, h_{i-1}, u_i, u_{i-1}^*, u_{\mathsf{C},i-1}, u_{\mathsf{C},i-1}^*)$. The formal IVC construction is illustrated in Appendix B.3.

3.3 *SnarkMux* Construction

Interface. Relative to the generic aggregation interface AGG in Section 2.2, our construction $\mathcal{S}$ adds a verifier-side preprocessing algorithm $h \leftarrow \mathcal{S}.\mathsf{VPrep}(\mathsf{vk}, (u_i)_{i=1}^n)$ that hashes the window's instance sequence into a binding claim. Accordingly, the prover outputs an aggregate pair (u^*, π^*) and an aggregation proof π_{AGG} carrying a prover-side claim h_n. The verifier checks $h = h_n$ and then verifies (u^*, π^*) together with the recursion proof carried in π_{AGG}.

To make π_{AGG} compact, we use another SNARK SNARK' to compress the circuit witness into a proof. Consequently, the structure s' of SNARK' is taken as an input to $\mathcal{S}.\mathsf{K}$.

As with AGG, we require that $\mathcal{S}$ satisfies perfect completeness and knowledge soundness (Appendix A.5).

Construction. Let $(u_\perp^*, \pi_\perp^*)$ be a fixed satisfying dummy instance-proof pair, and let $(u_{\mathsf{C},\perp}^*, w_{\mathsf{C},\perp}^*)$ be a fixed satisfying dummy circuit instance-witness pair. Figure 2 gives the full construction. The prover runs $\mathsf{IVC.P}$ for n steps to obtain a final IVC proof Π_n, and parses Π_n to obtain: the prover-side binding claim h_n, the running SNARK pair (u_n^*, π_n^*), the circuit pair $(u_{\mathsf{C},n}, w_{\mathsf{C},n})$, and the running circuit pair $(u_{\mathsf{C},n}^*, w_{\mathsf{C},n}^*)$. The aggregate SNARK pair is $(u^*, \pi^*) \leftarrow (u_n^*, \pi_n^*)$.

Thus the two verifier-visible proof objects have separate roles: (u^*, π^*) is the final target-layer fold checked by $\mathsf{V}_{\mathsf{tgt}}$, whereas π_{AGG} certifies that the ordered hash chain and both target- and circuit-layer states were updated consistently.

To compute π_{AGG}, the prover additionally folds $(u_{\mathsf{C},n}, w_{\mathsf{C},n})$ and $(u_{\mathsf{C},n}^*, w_{\mathsf{C},n}^*)$ into a single pair (u', w'), and uses SNARK' to prove knowledge of w' for the folded circuit relation induced by RC. The aggregation proof is $\pi_{\mathsf{AGG}} \leftarrow (h_n, u_{\mathsf{C},n}, u_{\mathsf{C},n}^*, \pi')$.

Theorem 1. *Assume collision resistance of* Hash*, perfect completeness and knowledge soundness of the target* SNARK *and final* SNARK'*, and perfect completeness and knowledge soundness of* $\mathsf{Fold}_{\mathsf{SNARK}}$ *and* $\mathsf{Fold}_{\mathsf{R1CS}}$ *(in the random-oracle model for their Fiat–Shamir versions). Then the IVC construction from these two folds and* SNARK' *yields a proof aggregation scheme with instance delegation for the target SNARK. The maximum number of aggregated proofs equals the supported recursion length of* IVC*, i.e.,* $N_{\max}$*.*

$\mathsf{pp} \leftarrow \mathcal{S}.\mathsf{G}(1^\lambda)$

1 : $\mathsf{pp}_{\mathsf{IVC}} \leftarrow \mathsf{IVC}.\mathsf{G}(1^\lambda)$,
2 : $\mathsf{pp}_{\mathsf{SNARK}'} \leftarrow \mathsf{SNARK}'.\mathsf{G}(1^\lambda)$,
3 : **output** $\mathsf{pp} \leftarrow (\mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{SNARK}'})$.

$(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}.\mathsf{K}(\mathsf{pp}, \mathsf{s}')$

1 : **parse** pp **as** $(\mathsf{pp}_{\mathsf{IVC}}, \mathsf{pp}_{\mathsf{SNARK}'})$,
2 : $(\mathsf{pk}_{\mathsf{IVC}}, \mathsf{vk}_{\mathsf{IVC}}) \leftarrow \mathsf{IVC}.\mathsf{K}(\mathsf{pp}_{\mathsf{IVC}}, \mathsf{RC})$,
3 : $(\mathsf{pk}_{\mathsf{SNARK}'}, \mathsf{vk}_{\mathsf{SNARK}'}) \leftarrow \mathsf{SNARK}'.\mathsf{K}(\mathsf{pp}_{\mathsf{SNARK}'}, \mathsf{s}')$,
4 : $\mathsf{pk} \leftarrow (\mathsf{s}', \mathsf{pk}_{\mathsf{IVC}}, \mathsf{pk}_{\mathsf{SNARK}'})$,
5 : $\mathsf{vk} \leftarrow (\mathsf{s}', \mathsf{vk}_{\mathsf{IVC}}, \mathsf{vk}_{\mathsf{SNARK}'})$,
6 : **output** $(\mathsf{pk}, \mathsf{vk})$.

$((u^*, \pi^*), \pi_{\mathsf{AGG}}) \leftarrow \mathcal{S}.\mathsf{P}\big(\mathsf{pk}, (u_i, \pi_i)_{i=1}^n\big)$ new

1 : **parse** pk **as** $(\mathsf{s}', \mathsf{pk}_{\mathsf{IVC}}, \mathsf{pk}_{\mathsf{SNARK}'})$,
2 : **parse** $\mathsf{pk}_{\mathsf{IVC}}$ **as** $(\mathsf{pk}_{\mathsf{FC}}, \mathsf{pk}_{\mathsf{FS}}, \ldots)$,
3 : $\mathit{\Pi}_0 \leftarrow \bot$,
4 : **for** $i = 1$ **to** n
5 : $\quad \mathit{\Pi}_i \leftarrow \mathsf{IVC}.\mathsf{P}\big(\mathsf{pk}_{\mathsf{IVC}}, i, (u_i, \pi_i), \mathit{\Pi}_{i-1}\big)$;
6 : $\big(h_n, (u_n^*, \pi_n^*), (u_{\mathsf{C},n}, w_{\mathsf{C},n}), (u_{\mathsf{C},n}^*, w_{\mathsf{C},n}^*)\big) \leftarrow \mathit{\Pi}_n, \quad (u^*, \pi^*) \leftarrow (u_n^*, \pi_n^*)$,
7 : $(u', w') \leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{P}\big(\mathsf{pk}_{\mathsf{FC}}, (u_{\mathsf{C},n}, w_{\mathsf{C},n}), (u_{\mathsf{C},n}^*, w_{\mathsf{C},n}^*)\big)$,
8 : $\pi' \leftarrow \mathsf{SNARK}'.\mathsf{P}(\mathsf{pk}_{\mathsf{SNARK}'}, u', w')$,
9 : $\pi_{\mathsf{AGG}} \leftarrow (h_n, u_{\mathsf{C},n}, u_{\mathsf{C},n}^*, \pi')$,
10 : **output** $\big((u^*, \pi^*), \pi_{\mathsf{AGG}}\big)$.

$h \leftarrow \mathcal{S}.\mathsf{VPrep}\big(\mathsf{vk}, (u_i)_{i=1}^n\big)$ new

1 : $h_0 \leftarrow h_\perp$,
2 : **for** $i = 1$ **to** n,
3 : $\quad h_i \leftarrow \mathrm{Hash}(u_i, h_{i-1})$;
4 : **output** $h \leftarrow h_n$.

$0/1 \leftarrow \mathcal{S}.\mathsf{V}\big(\mathsf{vk}, n, h, (u^*, \pi^*), \pi_{\mathsf{AGG}}\big)$ new

1 : **parse** vk **as** $(\mathsf{s}', \mathsf{vk}_{\mathsf{IVC}}, \mathsf{vk}_{\mathsf{SNARK}'})$,
2 : **parse** $\mathsf{vk}_{\mathsf{IVC}}$ **as** $(\mathsf{vk}_{\mathsf{FC}}, \mathsf{vk}_{\mathsf{FS}}, \ldots)$,
3 : **parse** s' **as** $(\mathsf{vk}_{\mathsf{FS}}, \mathsf{s})$,
4 : **parse** π_{AGG} **as** $(h_n, u_{\mathsf{C},n}, u_{\mathsf{C},n}^*, \pi')$,
5 : **check** $h = h_n$ and $u_{\mathsf{C},n}.x = \mathrm{Hash}(\mathsf{vk}, n, h_n, u^*, u_{\mathsf{C},n}^*)$,
6 : **compute** $u' \leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{V}\big(\mathsf{vk}_{\mathsf{FC}}, u_{\mathsf{C},n}, u_{\mathsf{C},n}^*\big)$,
7 : **check** $\mathsf{V}_{\mathsf{tgt}}(\mathsf{vk}_{\mathsf{FS}}, u^*, \pi^*) = 1$,
8 : **check** $\mathsf{SNARK}'.\mathsf{V}(\mathsf{vk}_{\mathsf{SNARK}'}, u', \pi') = 1$,
9 : **check** $u_{\mathsf{C},n}$ is a non-relaxed instance.

Fig. 2: The SNARKMUX scheme (new highlights the added verifier-preprocessing interface (VPrep), and proving and verification (P and V) with an explicit binding claim (h_n and h).)

Proof Sketch. We argue the two properties below.

Perfect completeness. For honestly generated inputs, acceptance follows from perfect completeness of SNARK and SNARK', and perfect completeness of $\mathsf{Fold}_{\mathsf{SNARK}}$ and $\mathsf{Fold}_{\mathsf{R1CS}}$ for honestly generated folds.

Knowledge soundness. Fix any instance sequence $(u_i)_{i=1}^n$. Let $h \leftarrow \mathcal{S}.\mathsf{VPrep}(\mathsf{vk}, (u_i)_{i=1}^n)$. Let $\mathcal{A}$ be any adversary that outputs $((u^*, \pi^*), \pi_{\mathsf{AGG}})$ where $\mathcal{S}.\mathsf{V}(\mathsf{vk}, n, h, (u^*, \pi^*), \pi_{\mathsf{AGG}}) = 1$. Parse π_{AGG} as $(h_n, u_{\mathsf{C},n}, u^*_{\mathsf{C},n}, \pi')$, and let $u' \leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{V}(\mathsf{vk}_{\mathsf{FC}}, u_{\mathsf{C},n}, u^*_{\mathsf{C},n})$ as in the verifier.

The knowledge soundness of SNARK' allows extraction of a witness w' such that (u', w') satisfies the folded circuit relation asserted by $\mathsf{SNARK}'.\mathsf{V}(\mathsf{vk}_{\mathsf{SNARK}'}, u', \pi') = 1$. Applying the knowledge extractor of $\mathsf{Fold}_{\mathsf{R1CS}}$ to this accepting fold yields valid witnesses for the final non-relaxed circuit instance and the running relaxed circuit instance. Repeating the circuit-fold extraction through the fixed n-step transcript exposes each certified recursive step; applying the $\mathsf{Fold}_{\mathsf{SNARK}}$ extractor at those steps recovers a valid target proof for every fresh target instance, except with the accumulated negligible extraction error.

The verifier also checks $h = h_n$ and $u_{\mathsf{C},n}.x = \mathrm{Hash}(\mathsf{vk}, n, h_n, u^*, u^*_{\mathsf{C},n})$. By the binding property of $\mathcal{S}.\mathsf{VPrep}$ (Lemma 2), the value h uniquely binds, except with negligible probability, to the instance sequence $(u_i)_{i=1}^n$ and its order. Therefore, satisfaction of the extracted recursive-circuit constraints implies that u^* is the result of iteratively applying $\mathsf{Fold}_{\mathsf{SNARK}}$ across the bound sequence, up to the maximum supported recursion length of the underlying folding-based IVC construction.

Lastly, the verifier checks $\mathsf{V}_{\mathsf{tgt}}(\mathsf{vk}_{\mathsf{FS}}, u^*, \pi^*) = 1$. By the target-relation check and the preceding $\mathsf{Fold}_{\mathsf{SNARK}}$ extraction, this implies, except with negligible probability, that all extracted constituent proofs are valid, *i.e.*, $\mathsf{SNARK}.\mathsf{V}(\mathsf{vk}_{\mathsf{SNARK}}, u_i, \pi_i) = 1$ for all $i \in [n]$. The stated step bound follows from the maximum supported recursion length of the underlying folding-based IVC, *i.e.*, $N_{\max}$.

4 Instantiating SnarkMux with Groth16

4.1 Groth16 Background

Groth16 is a pairing-based zk-SNARK with succinct proofs and fast verification [16]. It is defined over bilinear groups $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, e)$ where p is the prime order and $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is the bilinear mapping. Let $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$ be generators. For $x \in \mathbb{Z}_p$, define $[x]_1 \coloneqq g^x \in \mathbb{G}_1$, $[x]_2 \coloneqq h^x \in \mathbb{G}_2$, and $[x]_T \coloneqq e(g, h)^x \in \mathbb{G}_T$.

Setup and keys. Groth16 uses a circuit-specific SRS. Following our convention (as in Nova [19]), we treat the SRS as both pk and vk.

Proof format. For an R1CS instance-witness pair (u, w), the prover outputs a proof $\pi_{\mathsf{Groth16}} = (A, B, C) \in (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_1)$.

Verification equation. The public input (instance) is $u = (a_1, \ldots, a_\ell) \in \mathbb{Z}_p^\ell$ (sometimes we write $(a_0, \ldots, a_\ell)$ where $a_0 = 1$). The verifier checks a single

pairing product equation of the form

$$e(A, B) = e(C, [\delta]_2) \cdot e\left(\prod_{i=0}^{\ell} S_i^{a_i}, [\gamma]_2\right) \cdot e([\alpha]_1, [\beta]_2), \tag{3}$$

where $([\alpha]_1, [\beta]_2, [\gamma]_2, [\delta]_2, (S_i)_{i=0}^{\ell}) \in (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_2, \mathbb{G}_2, \mathbb{G}_1^{\ell+1})$ are SRS elements derived from the circuit. Full Groth16 details appear in Appendix B.1.

4.2 Folding Scheme for Groth16

Obstacle. For clarity, we simplify Groth16 verification in Equation (3) as $e(A, B) = e(C, [\delta]_2) \cdot e(\prod_{i=0}^{\ell} S_i^{a_i}, [\gamma]_2) \cdot D$ for $D = e([\alpha]_1, [\beta]_2)$. A naïve random-linear fold of two Groth16 instances introduces cross terms, so the verifier equation in Equation (3) is not preserved under folding.

Fix. We *relax* (and later augment) the Groth16 relation with auxiliary terms that absorb these cross terms. This yields a folding-compatible relation while keeping the final verification check in the original pairing form.

First attempt: relaxed Groth16. This part (i) defines the relaxed (and augmented) Groth16 relation used by folding, (ii) gives the folding protocol and its knowledge-soundness argument, and (iii) quantifies the resulting prover/verifier cost impact (in particular, what work is deferred to the final check).

Our first attempt is to propose a relaxed relation, named "relaxed" Groth16, with auxiliary terms absorbing cross terms.

Definition 1 (Relaxed Groth16 Proof Relation). *Given the structure with* $([\delta]_2, [\gamma]_2, (S_i)_{i=0}^{\ell}, D) \in (\mathbb{G}_2, \mathbb{G}_2, \mathbb{G}_1^{\ell+1}, \mathbb{G}_T)$*, a relaxed Groth16 proof relation consists of an instance* $(\boldsymbol{a}, \mu, E) \in (\mathbb{Z}_p^{\ell+1}, \mathbb{Z}_p, \mathbb{G}_T)$ *and a proof* $(A, B, C) \in (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_1)$*, such that*

$$e(A, B) \cdot e(C, [\delta]_2)^{-\mu} \cdot e(\prod_{i=0}^{\ell} S_i^{a_i}, [\gamma]_2)^{-\mu} \cdot D^{-\mu^2} = E,$$

where $\boldsymbol{a} = (a_0, \ldots, a_\ell) \in \mathbb{Z}_p^{\ell+1}$.

The relaxed relation introduces two additional elements, E and μ. Specifically, E is used to absorb the cross-term generated by folding, and μ is used to absorb additional factors. For traditional (non-relaxed) Groth16 proof relations, $\mu = 1$ and $E = [0]_T$.

We describe the folding scheme for relaxed Groth16. Given two instance-proof pairs, $(u_1, \pi_1) = \big((\boldsymbol{a}_1, \mu_1, E_1), (A_1, B_1, C_1)\big)$ and $(u_2, \pi_2) = \big((\boldsymbol{a}_2, \mu_2, E_2), (A_2, B_2, C_2)\big)$, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ engage in the following protocol:

1. $\mathcal{P} \to \mathcal{V}$: Compute and send the cross-term $T \in \mathbb{G}_T$:

$$\begin{aligned} T \leftarrow & e(A_1, B_2) \cdot e(A_2, B_1) \cdot e(C_1^{-\mu_2} C_2^{-\mu_1}, [\delta]_2) \\ & \cdot e(\prod_{i=0}^{\ell} S_i^{-\mu_2 a_{1,i} - \mu_1 a_{2,i}}, [\gamma]_2) \cdot D^{-2\mu_1\mu_2}. \end{aligned}$$

2. $\mathcal{V} \to \mathcal{P}$: Sample and send a challenge $r \leftarrow\!\!\$\ \mathbb{Z}_p$.
3. $\mathcal{P}$ and $\mathcal{V}$: Compute the folded relaxed Groth16 instance $(\boldsymbol{a}^*, \mu^*, E^*)$:

$$\boldsymbol{a}^* \leftarrow \boldsymbol{a}_1 + r \cdot \boldsymbol{a}_2, \quad \mu^* \leftarrow \mu_1 + r \cdot \mu_2, \quad E^* \leftarrow E_1 \cdot T^r \cdot E_2^{r^2}.$$

4. $\mathcal{P}$: Compute the folded relaxed Groth16 (A^*, B^*, C^*):

$$A^* \leftarrow A_1 \cdot A_2^r, \quad B^* \leftarrow B_1 \cdot B_2^r, \quad C^* \leftarrow C_1 \cdot C_2^r.$$

Reducing pairings: augmented relaxed Groth16. In the above construction, $\mathcal{P}$ is required to compute *four* pairings for T at Step 1 in each iteration. Although the pairing is performed locally (not in the recursive circuit), it introduces a non-negligible cost. To improve efficiency, $\mathcal{P}$ can compute and send $T' = e(A_1, B_2) \cdot e(A_2, B_1)$, $R = C_1^{-\mu_2} C_2^{-\mu_1}$, $\boldsymbol{t} = -\mu_2 \boldsymbol{a}_1 - \mu_1 \boldsymbol{a}_2$, and $\kappa = -2\mu_1\mu_2$ with *two* pairings in Step 1. Consequently, E^* in Step 3 becomes $E^* = E_1 \cdot \big(T' \cdot e(R, [\delta]_2) \cdot e(\prod_{i=0}^{\ell} S_i^{t_i}, [\gamma]_2) \cdot D^{\kappa}\big)^r \cdot E_2^{r^2}$. This may not seem helpful since computing E^* requires two additional pairings, but we observe that R and $\prod_{i=0}^{\ell} S_i^{t_i}$ (and κ) parts can be further *folded* in each iteration, and the pairing operation (and $\mathbb{G}_T$ operation) can be deferred to the final step. This indicates that we need only $2n+2$ pairings for folding n proofs instead of $4n$ pairings. To achieve this optimization, we introduce the concept of an "augmented relaxed Groth16 proof relation."

Definition 2 (Augmented Relaxed Groth16 Relation). *Given the structure with $([\delta]_2, [\gamma]_2, (S_i)_{i=0}^{\ell}, D) \in (\mathbb{G}_2, \mathbb{G}_2, \mathbb{G}_1^{\ell+1}, \mathbb{G}_T)$, an augmented relaxed Groth16 proof relation consists of an instance $(\boldsymbol{a}, \mu, E, R, \boldsymbol{t}, \kappa) \in (\mathbb{Z}_p^{\ell+1}, \mathbb{Z}_p, \mathbb{G}_T, \mathbb{G}_1, \mathbb{Z}_p^{\ell+1}, \mathbb{Z}_p)$ and a proof $(A, B, C) \in (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_1)$, such that*

$$\begin{aligned} & e(A, B) \cdot e(C, [\delta]_2)^{-\mu} \cdot e(\prod_{i=0}^{\ell} S_i^{a_i}, [\gamma]_2)^{-\mu} \cdot D^{-\mu^2} \\ = \; & E \cdot e(R, [\delta]_2) \cdot e(\prod_{i=0}^{\ell} S_i^{t_i}, [\gamma]_2) \cdot D^{\kappa}. \end{aligned}$$

For traditional Groth16 relations, we can simply set $\mu = 1$, $E = [0]_T$, $R = [0]_1$, $\boldsymbol{t} = \boldsymbol{0}$, and $\kappa = 0$. Given two instance-proof pairs $(u_1, \pi_1) = \big((\boldsymbol{a}_1, \mu_1, E_1, R_1, \boldsymbol{t}_1, \kappa_1), (A_1, B_1, C_1)\big)$ and $(u_2, \pi_2) = \big((\boldsymbol{a}_2, \mu_2, E_2, R_2, \boldsymbol{t}_2, \kappa_2), (A_2, B_2, C_2)\big)$, we describe the construction for augmented relaxed Groth16 in the following protocol. The non-interactive version can be implemented by the Fiat–Shamir transformation.

1. $\mathcal{P} \to \mathcal{V}$: Compute $T', R, \boldsymbol{t}, \kappa$. Send the cross-term T:

$$\begin{aligned} T' &\leftarrow e(A_1, B_2) \cdot e(A_2, B_1), \quad & R &\leftarrow C_1^{-\mu_2} C_2^{-\mu_1}, \\ \boldsymbol{t} &\leftarrow -\mu_2 \boldsymbol{a}_1 - \mu_1 \boldsymbol{a}_2, & \kappa &\leftarrow -2\mu_1\mu_2, \\ T &\leftarrow (T', R, \boldsymbol{t}, \kappa). \end{aligned} \tag{4}$$

2. $\mathcal{V} \to \mathcal{P}$: Sample and send a challenge $r \leftarrow\!\!\$\ \mathbb{Z}_p$.
3. $\mathcal{P}$ and $\mathcal{V}$: Compute the folded augmented relaxed Groth16 instance $(\boldsymbol{a}^*, \mu^*, E^*, R^*, \boldsymbol{t}^*, \kappa^*)$:

$$\begin{aligned} \boldsymbol{a}^* &\leftarrow \boldsymbol{a}_1 + r \cdot \boldsymbol{a}_2, & \mu^* &\leftarrow \mu_1 + r \cdot \mu_2, \\ E^* &\leftarrow E_1 \cdot (T')^r \cdot (E_2)^{r^2}, & R^* &\leftarrow R_1 \cdot R^r \cdot (R_2)^{r^2}, \\ \boldsymbol{t}^* &\leftarrow \boldsymbol{t}_1 + r \cdot \boldsymbol{t} + r^2 \cdot \boldsymbol{t}_2, & \kappa^* &\leftarrow \kappa_1 + r \cdot \kappa + r^2 \cdot \kappa_2. \end{aligned} \tag{5}$$

4. $\mathcal{P}$: Compute the folded augmented relaxed Groth16 proof (A^*, B^*, C^*):

$$A^* \leftarrow A_1 \cdot A_2^r, \quad B^* \leftarrow B_1 \cdot B_2^r, \quad C^* \leftarrow C_1 \cdot C_2^r.$$

Theorem 2. *The construction of the folding scheme for augmented relaxed Groth16 satisfies perfect completeness and knowledge soundness under the random oracle model.*

Proof Sketch. The perfect completeness holds trivially. For the knowledge soundness, we prove this interactive version of the protocol via the forking lemma in Lemma 1, which implies the knowledge soundness of the non-interactive construction under the Fiat–Shamir heuristic in the random oracle model. We present a formal proof in Appendix C.3.

4.3 Variant for Large ℓ Cases

Large ℓ case: committed augmented relaxed Groth16. The above construction requires $O(\ell)$ $\mathbb{Z}_p$ operations inside the recursive circuit. Here ℓ is the public-input length of the Groth16 statement, *i.e.*, the number of S_i terms appearing in the verification equation. When ℓ is large, we use a committed variant that removes this ℓ-dependent update work from the step circuit while keeping the per-step circuit cost $O(1)$.

This is achieved by redefining Definition 2 in committed form. We incorporate $\boldsymbol{a}$ into the proof and replace the explicit $S_i^{a_i}$ factors by a single committed term $H \coloneqq \prod_{i=0}^{\ell} S_i^{a_i}$ carried in the instance. The resulting relation is given in Definition 3.

Definition 3 (Committed Augmented Relaxed Groth16 Relation). *Given the structure with $([\delta]_2, [\gamma]_2, (S_i)_{i=0}^{\ell}, D) \in (\mathbb{G}_2, \mathbb{G}_2, \mathbb{G}_1^{\ell+1}, \mathbb{G}_T)$, a committed augmented relaxed Groth16 relation consists of an instance $(H, \mu, E, R, U, \kappa) \in (\mathbb{G}_1, \mathbb{Z}_p, \mathbb{G}_T, \mathbb{G}_1, \mathbb{G}_1, \mathbb{Z}_p)$ and a proof $(\boldsymbol{a}, A, B, C) \in (\mathbb{Z}_p^{\ell+1}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_1)$, such that*

$$\begin{aligned} & e(A, B) \cdot e(C, [\delta]_2)^{-\mu} \cdot e(H, [\gamma]_2)^{-\mu} \cdot D^{-\mu^2} \\ = {} & E \cdot e(R, [\delta]_2) \cdot e(U, [\gamma]_2) \cdot D^{\kappa}, \quad \wedge \quad H = \prod\nolimits_{i=0}^{\ell} S_i^{a_i}. \end{aligned}$$

Given two committed augmented relaxed Groth16 instance-proof pairs $\big((H_1, \mu_1, E_1, R_1, U_1, \kappa_1), (\boldsymbol{a}_1, A_1, B_1, C_1)\big)$ and $\big((H_2, \mu_2, E_2, R_2, U_2, \kappa_2), (\boldsymbol{a}_2, A_2, B_2, C_2)\big)$, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ engage in the following protocol:

1. $\mathcal{P} \to \mathcal{V}$: Compute T', R, κ as in Equation (4). Compute $V \leftarrow H_1^{-\mu_2} H_2^{-\mu_1}$. Set and send the cross terms $T \leftarrow (T', R, V, \kappa)$.
2. $\mathcal{V} \to \mathcal{P}$: Sample and send a challenge $r \leftarrow\$ \mathbb{Z}_p$.
3. $\mathcal{P}$ and $\mathcal{V}$: Compute $\mu^*, E^*, R^*, \kappa^*$ as in Equation (5). Compute H^* and U^*:

$$H^* \leftarrow H_1 \cdot H_2^r, \quad U^* \leftarrow U_1 \cdot V^r \cdot U_2^{r^2}.$$

4. $\mathcal{P}$: Compute the folded proof $(\boldsymbol{a}^*, A^*, B^*, C^*)$:

$$\begin{aligned} \boldsymbol{a}^* &\leftarrow \boldsymbol{a}_1 + r \cdot \boldsymbol{a}_2, \quad & A^* &\leftarrow A_1 \cdot A_2^r, \\ B^* &\leftarrow B_1 \cdot B_2^r, & C^* &\leftarrow C_1 \cdot C_2^r. \end{aligned}$$

Theorem 3. *The construction of the above folding scheme for committed Groth16 satisfies perfect completeness and knowledge soundness under the random oracle model.*

Proof Sketch. For the perfect completeness, $H^* = \prod_{i=0}^{\ell} S_i^{a_i^*}$ holds since $H^* = H_1 \cdot H_2^r = \big(\prod_{i=0}^{\ell} S_i^{a_{1,i}}\big) \cdot \big(\prod_{i=0}^{\ell} S_i^{a_{2,i}}\big)^r = \prod_{i=0}^{\ell} S_i^{a_i^*}$. The remaining parts are identical to the proof of Theorem 2. As for knowledge soundness, within the three-branch degree-two fork used for Theorem 2, any two distinct challenges interpolate $\hat{\boldsymbol{a}}_1$ and $\hat{\boldsymbol{a}}_2$, and the remaining branch enforces the quadratic relation identity. Other parts are almost identical to those of Theorem 2.

4.4 Non-uniform Structure

The constructions we have presented so far are only applicable in cases of uniform structure, where different instance-proof pairs share the same structure $([\delta]_2, [\gamma]_2, (S_i)_{i=0}^{\ell}, D)$. For non-uniform pairs with different structures, we can also construct a similar folding scheme at the cost of some additional pairing operations. For this extension, $\mathcal{S}.\mathsf{VPrep}$ hashes the ordered pairs $(\mathsf{vk}_{\mathsf{SNARK},i}, u_i)$, rather than u_i alone, so the binding claim covers the changing verification structures as well as the instances.

Definition 4 (Non-uniform Committed Relaxed Groth16). *A non-uniform committed relaxed Groth16 proof consists of a structure $([\delta]_2, [\gamma]_2, (S_i)_{i=0}^{\ell}, D) \in (\mathbb{G}_2, \mathbb{G}_2, \mathbb{G}_1^{\ell+1}, \mathbb{G}_T)$, an instance $(H, \mu, E, F) \in (\mathbb{G}_1, \mathbb{Z}_p, \mathbb{G}_T, \mathbb{G}_1)$, and a proof $(\boldsymbol{a}, A, B, C) \in (\mathbb{Z}_p^{\ell+1}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_1)$, such that*

$$\begin{aligned} & e(A, B) \cdot e(C, [\delta]_2)^{-1} \cdot e(H, [\gamma]_2)^{-1} \cdot D^{-\mu} = E, \\ \wedge \quad & H^{-\mu} \cdot \prod_{i=0}^{\ell} S_i^{a_i} = F. \end{aligned}$$

Given two non-uniform structure-instance-proof tuples $\big(([\delta_1]_2, [\gamma_1]_2, (S_{1,i})_{i=0}^{\ell}, D_1), (H_1, \mu_1, E_1, F_1), (\boldsymbol{a}_1, A_1, B_1, C_1)\big)$ and $\big(([\delta_2]_2, [\gamma_2]_2, (S_{2,i})_{i=0}^{\ell}, D_2), (H_2, \mu_2, E_2, F_2), (\boldsymbol{a}_2, A_2, B_2, C_2)\big)$, $\mathcal{P}$ and $\mathcal{V}$ engage in the following protocol:

1. $\mathcal{P} \to \mathcal{V}$: Compute and send the cross-terms T_1 and T_2:

$$\begin{aligned} T_1 \leftarrow & e(A_1, B_2) \cdot e(A_2, B_1) \cdot e(C_1, [\delta_2]_2)^{-1} \cdot e(C_2, [\delta_1]_2)^{-1} \\ & \cdot e(H_1, [\gamma_2]_2)^{-1} \cdot e(H_2, [\gamma_1]_2)^{-1} \cdot D_1^{-\mu_2} \cdot D_2^{-\mu_1}, \\ T_2 \leftarrow & H_1^{-\mu_2} \cdot H_2^{-\mu_1} \cdot \prod_{i=0}^{\ell} \left(S_{1,i}^{a_{2,i}} \cdot S_{2,i}^{a_{1,i}} \right). \end{aligned}$$

2. $\mathcal{V} \to \mathcal{P}$: Sample and send a challenge $r \leftarrow\!\!\$\, \mathbb{Z}_p$.
3. $\mathcal{P}$ and $\mathcal{V}$: Compute the folded committed Groth16 structure-instance pair $\big(([\delta^*]_2, [\gamma^*]_2, (S_i^*)_{i=0}^{\ell}, D^*), (H^*, \mu^*, E^*, F^*)\big)$:

$$\begin{aligned} [\delta^*]_2 \leftarrow [\delta_1]_2 \cdot [\delta_2]_2^r, \quad & [\gamma^*]_2 \leftarrow [\gamma_1]_2 \cdot [\gamma_2]_2^r, \\ S_i^* \leftarrow S_{1,i} \cdot S_{2,i}^r, \quad & D^* \leftarrow D_1 \cdot D_2^r, \\ H^* \leftarrow H_1 \cdot H_2^r, \quad & \mu^* \leftarrow \mu_1 + r \cdot \mu_2, \\ E^* = E_1 \cdot T_1^r \cdot E_2^{r^2}, \quad & F^* = F_1 \cdot T_2^r \cdot F_2^{r^2}. \end{aligned}$$

4. $\mathcal{P}$: Compute the folded relaxed Groth16 proof $(\boldsymbol{a}^*, A^*, B^*, C^*)$:

$$\begin{aligned} \boldsymbol{a}^* \leftarrow \boldsymbol{a}_1 + r \cdot \boldsymbol{a}_2, \quad & A^* \leftarrow A_1 \cdot A_2^r, \\ B^* \leftarrow B_1 \cdot B_2^r, \quad & C^* \leftarrow C_1 \cdot C_2^r. \end{aligned}$$

The *perfect completeness* property holds by expansion, and *knowledge soundness* uses a three-branch fork sharing (T_1, T_2) because the folded identities have degree two in the challenge. The proof steps closely follow those of Theorem 2.

5 Instantiating *S*NARKMUX with Plonk

5.1 Plonk Background

Plonk is a zk-SNARK with a universal SRS and verification based on KZG polynomial commitments [12,17]. We describe Plonk following its original construction [12], but only record the interface points used later by our Plonk folding preprocessing in Section 5.2.

Setup and keys. Plonk uses a universal structured reference string (SRS) that supports committing to polynomials. As above, we treat the SRS as both pk and vk.

Proof format. A Plonk proof consists of (i) a constant number of commitments in $\mathbb{G}_1$ and (ii) a constant number of field elements in $\mathbb{Z}_p$ that open those commitments at transcript-derived challenge points. We denote a Plonk proof abstractly by π_{Plonk} and refer to Appendix B.2 for the concrete tuple.

Verification structure. The verifier recomputes Fiat–Shamir challenges from the transcript and checks: (i) algebraic consistency of claimed evaluations in $\mathbb{Z}_p$, and (ii) correctness of KZG openings via a batched pairing check. Concretely, verification reduces to a single pairing equation of the form

$$e(\mathsf{LHS}(\pi_{\mathsf{Plonk}}), [x]_2) = e(\mathsf{RHS}(\pi_{\mathsf{Plonk}}), [1]_2),$$

where x is the SRS trapdoor element and $\mathsf{LHS}, \mathsf{RHS}$ are group elements derived from the proof, public inputs, and transcript challenges. This transcript dependence is the key obstacle for folding, and motivates our in-circuit preprocessing that binds the transcript and converts the proof into a folding-friendly relation in Section 5.2.

Note on batching. The above pairing-shaped equality is not exactly the standard formulation of Plonk's batched KZG opening verification, where the verifier combines the required opening checks using transcript-derived randomizers (Fiat–Shamir challenges) and checks a single pairing-shaped identity. Appendix B.2 spells out the rearrangement and the standard randomized-batching soundness intuition.

5.2 Folding Scheme for Plonk

Unlike Groth16, Plonk is made non-interactive via Fiat–Shamir. In Plonk, verifier challenges are defined as hashes of the evolving transcript.

Obstacle. When two proofs are folded, the folded transcript induces new challenges, so correctness cannot be ensured by folding the verifier equations while reusing the original challenge values. Therefore, folding must also enforce transcript consistency, *i.e.*, that the challenges used by the folded relation match the hashes of the folded transcript.

Fix. We add an in-circuit transformation that re-derives and checks the relevant transcript hashes and maps each fresh proof to a folding-compatible algebraic relation. We outline two preprocessed Plonk relations: a linear relation and a quadratic relation. We then give the corresponding folding constructions, prove perfect completeness and knowledge soundness in the ROM, and summarize the circuit-cost tradeoff between the two instantiations. These procedures are unrelated to verifier-side $\mathcal{S}.\mathsf{VPrep}$: $\mathsf{Preprocess}^{\mathrm{lin}}$ and $\mathsf{Preprocess}^{\mathrm{quad}}$ are prover-executed fresh-input conversions whose correctness is certified inside each recursive step. The linear variant derives two group elements and supports k-folding, but uses 22 in-circuit $\mathbb{G}_1$ exponentiations (31,980 primary constraints in our implementation). The quadratic variant keeps scalar coefficients and uses one cross term for 2-folding, reducing this to 13 exponentiations (22,989 constraints) while increasing local work from $22n$ to $50n$ $\mathbb{G}_1$ operations and the target/circuit output accounting from 6 to 15 $\mathbb{G}_1$ elements.

Linear relation. Observe that the final verification of Plonk is in a linear form of $e(P, [x]_2) \stackrel{?}{=} e(Q, [1]_2)$. We can preprocess the proof to $P, Q \in \mathbb{G}_1$ and construct (multiple) folding without incurring any cross-term. Specifically, the preprocessing procedure $\mathsf{Preprocess}^{\mathrm{lin}}$ includes a Plonk verification except for the pairing check part, which is defined in Figure 3. This isolates all transcript-dependent checks into hash/field logic that is circuit-friendly, leaving a single pairing-shaped equation that is closed under random-linear-combination folding. In the figure, $(\beta, \theta, \alpha, \lambda, v, \gamma)$ are transcript challenges; M and N batch proof and verification-key commitments; F encodes the claimed evaluations; and (P, Q) are the two sides of the remaining pairing equation.

$(P, Q) \leftarrow \mathsf{Preprocess}^{\mathrm{lin}}(\mathsf{vk}_{\mathsf{Plonk}}, u_{\mathsf{Plonk}}, \pi_{\mathsf{Plonk}})$

$\beta \leftarrow \mathrm{Hash}(A, B, C),\ \theta \leftarrow \mathrm{Hash}(\beta), \alpha \leftarrow \mathrm{Hash}(\theta, Z),$

$\lambda \leftarrow \mathrm{Hash}(\alpha, T_l, T_{mi}, T_h),$

$v \leftarrow \mathrm{Hash}(\lambda, \bar{a}, \bar{b}, \bar{c}, \bar{s}_{\sigma_1}, \bar{s}_{\sigma_2}, \bar{z}_\omega),$

$\gamma \leftarrow \mathrm{Hash}(v, W, V),$

$M \leftarrow A^{v} \cdot B^{v^2} \cdot C^{v^3} \cdot (S_{\sigma_1})^{v^4} \cdot (S_{\sigma_2})^{v^5},$

$N \leftarrow (Q_M)^{\bar{a}\bar{b}} \cdot (Q_L)^{\bar{a}} \cdot (Q_R)^{\bar{b}} \cdot (Q_O)^{\bar{c}} \cdot Q_C$

$\quad \cdot (Z)^{\alpha(\bar{a}+\beta\lambda+\theta)\left(\bar{b}+\beta k_1\lambda+\theta\right)(\bar{c}+\beta k_2\lambda+\theta)+\alpha^2 L_1(\lambda)+\gamma}$

$\quad \cdot (S_{\sigma_3})^{-\alpha\left(\bar{a}+\beta\bar{s}_{\sigma_1}+\theta\right)\left(\bar{b}+\beta\bar{s}_{\sigma_2}+\theta\right)\beta\bar{z}_\omega}$

$\quad \cdot (T_l)^{-z_H(\lambda)} \cdot (T_{mi})^{-z_H(\lambda)\cdot\lambda^m} \cdot (T_h)^{-z_H(\lambda)\cdot\lambda^{2m}},$

$r_0 \leftarrow u(\lambda) - \alpha^2 L_1(\lambda) - \alpha(\bar{a} + \beta\bar{S}_{\sigma_1} + \theta)(\bar{b} + \beta\bar{S}_{\sigma_2} + \theta)(\bar{c} + \theta)\bar{z}_\omega,$

$F \leftarrow [-r_0 + v\bar{a} + v^2\bar{b} + v^3\bar{c} + v^4\bar{s}_{\sigma_1} + v^5\bar{s}_{\sigma_2} + \gamma\bar{z}_\omega]_1,$

$P \leftarrow W \cdot V^{\gamma},$

$Q \leftarrow W^{\lambda} \cdot V^{\gamma\lambda\omega} \cdot M \cdot N \cdot F^{-1},$

return (P, Q).

Fig. 3: Preprocessing a Plonk proof into a linear relation

Definition 5 (Folded Linear Plonk Relation). *Given a structure* $([x]_2, \mathsf{vk}_{\mathsf{Plonk}})$*, the folded linear Plonk relation consists of an instance* (P, Q) *and an empty witness* $\perp$ *such that*

$$e(P, [x]_2) = e(Q, [1]_2).$$

Before a fresh pair $(u_{\mathsf{Plonk}}, \pi_{\mathsf{Plonk}})$ enters this relation, the recursive circuit checks $(P, Q) = \mathsf{Preprocess}^{\mathrm{lin}}(\mathsf{vk}_{\mathsf{Plonk}}, u_{\mathsf{Plonk}}, \pi_{\mathsf{Plonk}})$. Thus transcript certification is a fresh-input conversion, while the running folded relation contains only (P, Q); it is not required to be the preprocessing of any single Plonk transcript. Specifically, given two instances with (P_1, Q_1) and (P_2, Q_2), we can fold them into $P^* = P_1 \cdot P_2^r$ and $Q^* = Q_1 \cdot Q_2^r$ using a challenge r. It is clear that $e(P^*, [x]_2) = e(Q^*, [1]_2)$ still holds. This folding scheme also supports k-folding, which allows for the folding of k instances within a single incremental step. Given k fresh Plonk pairs and their certified images $\left(u_{\mathsf{Plonk},i}, \pi_{\mathsf{Plonk},i}, P_i, Q_i\right)_{i=1}^{k}$, the detailed k-folding construction is shown as follows:

1. $\mathcal{V} \to \mathcal{P}$: Sample and send a challenge $r \leftarrow\!\$\ \mathbb{Z}_p$.
2. $\mathcal{P}$: Compute P^* and Q^* and return the folded instance-witness pair $\left((P^*, Q^*), \perp\right)$:

$$P^* \leftarrow \prod_{i=1}^{k} P_i^{r^{i-1}}, \qquad Q^* \leftarrow \prod_{i=1}^{k} Q_i^{r^{i-1}}.$$

$(\lambda, \gamma, v, \bar{a}, \bar{b}, \bar{c}, \boldsymbol{t}) \leftarrow \mathsf{Preprocess}^{\mathrm{quad}}(\mathsf{vk}_{\mathsf{Plonk}}, u_{\mathsf{Plonk}}, \pi_{\mathsf{Plonk}})$

1 : $\beta \leftarrow \mathrm{Hash}(A, B, C),\ \theta \leftarrow \mathrm{Hash}(\beta),\ \alpha \leftarrow \mathrm{Hash}(\theta, Z),$
2 : $\lambda \leftarrow \mathrm{Hash}(\alpha, T_l, T_{mi}, T_h),$
3 : $v \leftarrow \mathrm{Hash}(\lambda, \bar{a}, \bar{b}, \bar{c}, \bar{s}_{\sigma_1}, \bar{s}_{\sigma_2}, \bar{z}_\omega),$
4 : $\gamma \leftarrow \mathrm{Hash}(v, W, V),$
5 : $t_1 \leftarrow v^2, \quad t_2 \leftarrow v^3, \quad t_3 \leftarrow v^4, \quad t_4 \leftarrow v^5,$
6 : $t_5 \leftarrow \alpha(\bar{a} + \beta\lambda + \theta)\left(\bar{b} + \beta k_1 \lambda + \theta\right)(\bar{c} + \beta k_2 \lambda + \theta) + \alpha^2 L_1(\lambda) + \gamma,$
7 : $t_6 \leftarrow -\alpha\left(\bar{a} + \beta\bar{s}_{\sigma 1} + \theta\right)\left(\bar{b} + \beta\bar{s}_{\sigma 2} + \theta\right)\beta\bar{z}_\omega,$
8 : $t_7 \leftarrow -z_H(\lambda), \quad t_8 \leftarrow -z_H(\lambda) \cdot \lambda^m, \quad t_9 \leftarrow -z_H(\lambda) \cdot \lambda^{2m},$
9 : $r_0 \leftarrow u(\lambda) - \alpha^2 L_1(\lambda) - \alpha(\bar{a} + \beta\bar{s}_{\sigma_1} + \theta)(\bar{b} + \beta\bar{s}_{\sigma_2} + \theta)(\bar{c} + \theta)\bar{z}_\omega,$
10 : $t_{10} \leftarrow -r_0 + v\bar{a} + v^2\bar{b} + v^3\bar{c} + v^4\bar{s}_{\sigma_1} + v^5\bar{s}_{\sigma_2} + \gamma\bar{z}_\omega,$
11 : $t_{11} \leftarrow \gamma\lambda\omega,$
12 : $\boldsymbol{t} \leftarrow (t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}),$
13 : **return** $(\lambda, \gamma, v, \bar{a}, \bar{b}, \bar{c}, \boldsymbol{t})$

Fig. 4: Preprocessing a Plonk proof into a quadratic relation

3. $\mathcal{V}$: For $i = 1$ to k, check

$$(P_i, Q_i) = \mathsf{Preprocess}^{\mathrm{lin}}(\mathsf{vk}_{\mathsf{Plonk}}, u_{\mathsf{Plonk},i}, \pi_{\mathsf{Plonk},i})$$

and compute the folded instance (P^*, Q^*):

$$P^* \leftarrow \prod_{i=1}^{k} P_i^{r^{i-1}}, \qquad Q^* \leftarrow \prod_{i=1}^{k} Q_i^{r^{i-1}}.$$

This can be made non-interactive by Fiat–Shamir transformation.

Theorem 4. *The linear Plonk folding scheme is perfectly complete and knowledge sound in the random oracle model.*

Proof Sketch. For each fresh satisfying Plonk pair, the recursive circuit certifies $(P_i, Q_i) = \mathsf{Preprocess}^{\mathrm{lin}}(\mathsf{vk}_{\mathsf{Plonk}}, u_{\mathsf{Plonk},i}, \pi_{\mathsf{Plonk},i})$. Furthermore, $e(P^*, [x]_2) = \prod_{i=1}^{k} e(P_i, [x]_2)^{r^{i-1}} = \prod_{i=1}^{k} e(Q_i, [1]_2)^{r^{i-1}} = e(Q^*, [1]_2)$. For the knowledge soundness, the extractor can trivially set the extracted witnesses to $\perp$. If any input pairing equation is false, acceptance of the random linear combination occurs with probability at most $(k-1)/|\mathbb{Z}_p|$ by Schwartz–Zippel. Hence, except with that error, an accepting fold implies that all (P_i, Q_i) satisfy the folded linear relation; the certified fresh-input conversions then tie them to the claimed Plonk pairs.

Quadratic relation. $\mathsf{Preprocess}^{\mathrm{lin}}$ requires the prover to conduct many group operations in the recursive circuit. Alternatively, $\mathsf{Preprocess}^{\mathrm{quad}}$ emits only scalar

coefficients: $t_1, \ldots, t_{11}$ are the exponents needed to reconstruct M, N, and the two opening-proof terms. This reduces the group operations in the recursive circuit, converting the proof into a quadratic relation at the cost of a single cross term in 2-folding cases. We denote this preprocessing as $\mathsf{Preprocess}^{\mathrm{quad}}$, which is defined in Figure 4.

Definition 6 (Relaxed Quadratic Plonk Relation). *Given a structure* $([x]_2, \mathsf{vk}_{\mathsf{Plonk}})$*, an instance contains scalars* $(\lambda, \gamma, v, \bar{a}, \bar{b}, \bar{c}, \boldsymbol{t}, \mu)$*, the carried public input* u_{Plonk}*, algebraic proof components* π_{Plonk}*, and error terms* (E_P, E_Q)*. Together with the empty witness* $\perp$*, it satisfies the relaxed quadratic Plonk relation if*

$$
\begin{aligned}
&M \leftarrow A^{v} \cdot B^{t_1} \cdot C^{t_2} \cdot S_{\sigma_1}^{t_3} \cdot S_{\sigma_2}^{t_4},\\
&N \leftarrow Q_M^{\bar{a}\bar{b}} \cdot Q_L^{\mu\bar{a}} \cdot Q_R^{\mu\bar{b}} \cdot Q_O^{\mu\bar{c}} \cdot Q_C^{\mu^2} \cdot Z^{t_5} \cdot S_{\sigma_3}^{t_6} \cdot T_l^{t_7} \cdot T_{mi}^{t_8} \cdot T_h^{t_9},\\
&P \leftarrow W^{\mu} \cdot V^{\gamma}, \quad Q \leftarrow W^{\lambda} \cdot V^{t_{11}} \cdot M \cdot N \cdot [-\mu t_{10}]_1,\\
&e(P, [x]_2) \cdot e(Q, [1]_2)^{-1} = e(E_P, [x]_2) \cdot e(E_Q, [1]_2)^{-1}.
\end{aligned}
$$

The last verification can be checked with two pairings. For a fresh pair, the recursive circuit checks $(\lambda, \gamma, v, \bar{a}, \bar{b}, \bar{c}, \boldsymbol{t}) = \mathsf{Preprocess}^{\mathrm{quad}}(\mathsf{vk}_{\mathsf{Plonk}}, u_{\mathsf{Plonk}}, \pi_{\mathsf{Plonk}})$ and initializes $\mu = 1$ and $E_P = E_Q = [0]_1$ before folding. The running relaxed instance need not itself be a valid Fiat–Shamir transcript: it contains the linear combinations of the certified algebraic proof components, public inputs, and challenges, while the public-input component is retained for the common target-state encoding.

We describe the interactive version of the folding scheme. Given two relaxed quadratic Plonk instances, $(\lambda_i, \gamma_i, v_i, \bar{a}_i, \bar{b}_i, \bar{c}_i, \boldsymbol{t}_i, u_{\mathsf{Plonk},i}, \pi_{\mathsf{Plonk},i}, \mu_i, E_{P,i}, E_{Q,i})_{i\in\{1,2\}}$, where $\boldsymbol{t}_i = (t_{1,i}, \ldots, t_{11,i})$, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ engage in the following protocol:

1. $\mathcal{P} \to \mathcal{V}$: Compute and send the cross-terms T_P and T_Q:

$$
\begin{aligned}
T_M \leftarrow& A_1^{v_2} A_2^{v_1} \cdot B_1^{t_{1,2}} B_2^{t_{1,1}} \cdot C_1^{t_{2,2}} C_2^{t_{2,1}} \cdot S_{\sigma_1,1}^{t_{3,2}} S_{\sigma_1,2}^{t_{3,1}} \cdot S_{\sigma_2,1}^{t_{4,2}} S_{\sigma_2,2}^{t_{4,1}},\\
T_N \leftarrow& Q_M^{\bar{a}_1\bar{b}_2+\bar{a}_2\bar{b}_1} \cdot Q_L^{\mu_1\bar{a}_2+\mu_2\bar{a}_1} \cdot Q_R^{\mu_1\bar{b}_2+\mu_2\bar{b}_1} \cdot Q_O^{\mu_1\bar{c}_2+\mu_2\bar{c}_1}\\
&\cdot Q_C^{2\mu_1\mu_2} \cdot Z_1^{t_{5,2}} Z_2^{t_{5,1}} \cdot S_{\sigma_3,1}^{t_{6,2}} S_{\sigma_3,2}^{t_{6,1}} \cdot T_{l,1}^{t_{7,2}} T_{l,2}^{t_{7,1}}\\
&\cdot T_{mi,1}^{t_{8,2}} T_{mi,2}^{t_{8,1}} \cdot T_{h,1}^{t_{9,2}} T_{h,2}^{t_{9,1}},\\
T_P \leftarrow& W_1^{\mu_2} W_2^{\mu_1} \cdot V_1^{\gamma_2} V_2^{\gamma_1},\\
T_Q \leftarrow& W_1^{\lambda_2} W_2^{\lambda_1} \cdot V_1^{t_{11,2}} V_2^{t_{11,1}} \cdot T_M \cdot T_N \cdot [-\mu_1 t_{10,2} - \mu_2 t_{10,1}]_1.
\end{aligned}
$$

2. $\mathcal{V} \to \mathcal{P}$: Sample and send a challenge $r \leftarrow\!\$\ \mathbb{Z}_p$.
3. $\mathcal{P}$ and $\mathcal{V}$: Compute the folded relaxed quadratic Plonk instance $(\lambda^*, \gamma^*, v^*, \bar{a}^*, \bar{b}^*, \bar{c}^*, \boldsymbol{t}^*, u^*_{\mathsf{Plonk}}, \pi^*_{\mathsf{Plonk}}, \mu^*, E^*_P, E^*_Q)$:

$$
\begin{aligned}
&\lambda^* \leftarrow \lambda_1 + r\lambda_2, \quad \gamma^* \leftarrow \gamma_1 + r \cdot \gamma_2, \quad v^* \leftarrow v_1 + r \cdot v_2, \quad \bar{a}^* \leftarrow \bar{a}_1 + r \cdot \bar{a}_2,\\
&\bar{b}^* \leftarrow \bar{b}_1 + r \cdot \bar{b}_2, \quad \bar{c}^* \leftarrow \bar{c}_1 + r \cdot \bar{c}_2, \quad \boldsymbol{t}^* \leftarrow \boldsymbol{t}_1 + r \cdot \boldsymbol{t}_2,\\
&(u^*_{\mathsf{Plonk}}, \pi^*_{\mathsf{Plonk}}) \leftarrow \mathsf{Lin}\big((u_{\mathsf{Plonk},1}, \pi_{\mathsf{Plonk},1}), (u_{\mathsf{Plonk},2}, \pi_{\mathsf{Plonk},2})\big),\\
&\mu^* \leftarrow \mu_1 + r \cdot \mu_2, \quad E^*_P \leftarrow E_{P,1} \cdot T_P^r \cdot E_{P,2}^{r^2}, \quad E^*_Q \leftarrow E_{Q,1} \cdot T_Q^r \cdot E_{Q,2}^{r^2},
\end{aligned}
$$

where Lin performs a scalar multiplication for each $\mathbb{G}_1$ element and a linear combination for each $\mathbb{Z}_p$ element (*e.g.*, $A^* = A_1 \cdot A_2^r$ and $\bar{z}^*_\omega = \bar{z}_{\omega,1} + r \cdot \bar{z}_{\omega,2}$).

Theorem 5. *The 2-folding Plonk scheme is perfectly complete and knowledge sound in the random oracle model.*

Proof Sketch. The perfect completeness and knowledge soundness of the protocol can be proved with an approach similar to the proof of Theorem 4. We present a formal proof in Appendix C.4.

5.3 Non-uniform Structure

The folding scheme for quadratic Plonk relations can be extended to support non-uniform Plonk proofs of different structures. In this extension, $\mathcal{S}.\mathsf{VPrep}$ binds the ordered pairs $(\mathsf{vk}_{\mathsf{Plonk},i}, u_{\mathsf{Plonk},i})$ so that the changing verification keys cannot be substituted. To achieve this, we introduce an additional step in $\mathsf{Preprocess}^{\mathrm{quad}}$ to compute $t_{12} \leftarrow \bar{a} \cdot \bar{b}$ and return t_{12} in $\boldsymbol{t}$. For each fresh proof, the recursive circuit certifies this extended preprocessing and initializes $P = WV^\gamma$, $\mu = 1$, $E_P = [0]_1$, and $E = [0]_T$; the folded relation below does not re-run Fiat–Shamir on the combined transcript. The new relation is as follows:

Definition 7 (Non-uniform Preprocessed Quadratic Plonk Relation). *A non-uniform preprocessed quadratic Plonk proof relation consists of a structure* $([x]_2, \mathsf{vk}_{\mathsf{Plonk}})$*, an instance* $(\lambda, \gamma, v, \bar{a}, \bar{b}, \bar{c}, \boldsymbol{t}, P, u_{\mathsf{Plonk}}, \pi_{\mathsf{Plonk}}, \mu, E_P, E)$*, and an empty witness* $\perp$ *such that*

$$
\begin{aligned}
&M \leftarrow A^v \cdot B^{t_1} \cdot C^{t_2} \cdot S_{\sigma_1}^{t_3} \cdot S_{\sigma_2}^{t_4},\\
&N \leftarrow Q_M^{t_{12}} \cdot Q_L^{\bar{a}} \cdot Q_R^{\bar{b}} \cdot Q_O^{\bar{c}} \cdot Q_C^{\mu} \cdot Z^{t_5} \cdot S_{\sigma_3}^{t_6} \cdot T_l^{t_7} \cdot T_{mi}^{t_8} \cdot T_h^{t_9},\\
&Q \leftarrow W^\lambda \cdot V^{t_{11}} \cdot M \cdot N \cdot [-\mu t_{10}]_1,\\
&P^{-\mu} \cdot W^\mu \cdot V^\gamma = E_P,\\
&e(P, [x]_2) \cdot e(Q, [1]_2)^{-1} = E.
\end{aligned}
$$

We briefly describe the interactive version of the folding scheme for non-uniform Plonk proofs. Given two non-uniform preprocessed quadratic Plonk structure-instance-witness tuples $\big(([x_1]_2, \mathsf{vk}_{\mathsf{Plonk},1}), (\lambda_1, \gamma_1, v_1, \bar{a}_1, \bar{b}_1, \bar{c}_1, \boldsymbol{t}_1, P_1, u_{\mathsf{Plonk},1}, \pi_{\mathsf{Plonk},1}, \mu_1, E_{P,1}, E_1), \perp\big)$ and $\big(([x_2]_2, \mathsf{vk}_{\mathsf{Plonk},2}), (\lambda_2, \gamma_2, v_2, \bar{a}_2, \bar{b}_2, \bar{c}_2, \boldsymbol{t}_2, P_2, u_{\mathsf{Plonk},2}, \pi_{\mathsf{Plonk},2}, \mu_2, E_{P,2}, E_2), \perp\big)$, the prover $\mathcal{P}$ and verifier $\mathcal{V}$ engage in the following protocol:

1. $\mathcal{P} \to \mathcal{V}$: Compute and send the cross-terms T_P and T:

$$
\begin{aligned}
T_M \leftarrow& A_1^{v_2} A_2^{v_1} \cdot B_1^{t_{1,2}} B_2^{t_{1,1}} \cdot C_1^{t_{2,2}} C_2^{t_{2,1}} \cdot S_{\sigma_1,1}^{t_{3,2}} S_{\sigma_1,2}^{t_{3,1}} \cdot S_{\sigma_2,1}^{t_{4,2}} S_{\sigma_2,2}^{t_{4,1}},\\
T_N \leftarrow& Q_{M,1}^{t_{12,2}} Q_{M,2}^{t_{12,1}} \cdot Q_{L,1}^{\bar{a}_2} Q_{L,2}^{\bar{a}_1} \cdot Q_{R,1}^{\bar{b}_2} Q_{R,2}^{\bar{b}_1} \cdot Q_{O,1}^{\bar{c}_2} Q_{O,2}^{\bar{c}_1} \cdot Q_{C,1}^{\mu_2} Q_{C,2}^{\mu_1},\\
T_Q \leftarrow& W_1^{\lambda_2} W_2^{\lambda_1} \cdot V_1^{t_{11,2}} V_2^{t_{11,1}} \cdot T_M \cdot T_N \cdot [-\mu_1 t_{10,2} - \mu_2 t_{10,1}]_1,\\
T_P \leftarrow& P_1^{-\mu_2} P_2^{-\mu_1} \cdot W_1^{\mu_2} W_2^{\mu_1} \cdot V_1^{\gamma_2} V_2^{\gamma_1},\\
T \leftarrow& e(P_1, [x_2]_2) \cdot e(P_2, [x_1]_2) \cdot e(T_Q, [1]_2)^{-1}.
\end{aligned}
$$

Table 2: Size of verifier-received outputs for an aggregation window of n proofs

Scheme	Output Size
Groth16 aggregation	
TIPP	6 $\mathbb{G}_1$, 6 $\mathbb{G}_2$, $(12\log n + 5)$ $\mathbb{G}_T$
SnarkPack	7 $\mathbb{G}_1$, 6 $\mathbb{G}_2$, $(12\log n + 5)$ $\mathbb{G}_T$
Mira [2]	3 $\mathbb{G}_1$, 1 $\mathbb{G}_2$, 2 $\mathbb{G}_T$, 1 $\mathbb{Z}_p$, $\lvert\pi_{\mathsf{SNARK}'}\rvert$
SnarkMux (aug.)	$(2\ell + 8)$ $\mathbb{Z}_p$, 7 $\mathbb{G}_1$, 1 $\mathbb{G}_2$, 1 $\mathbb{G}_T$, $\lvert\pi_{\mathsf{SNARK}'}\rvert$
SnarkMux (com.)	9 $\mathbb{G}_1$, 1 $\mathbb{G}_2$, 1 $\mathbb{G}_T$, $\lvert\pi_{\mathsf{SNARK}'}\rvert$
Plonk aggregation	
aPlonk	19 $\mathbb{Z}_p$, $(2\log_2(3n) + 10)$ $\mathbb{G}_1$, 2 $\mathbb{G}_2$, $(2\log_2(3n) + 3)$ $\mathbb{G}_T$
SnarkMux (lin.)	4 $\mathbb{Z}_p$, 6 $\mathbb{G}_1$, $\lvert\pi_{\mathsf{SNARK}'}\rvert$
SnarkMux (quad.)	$(\ell + 26)$ $\mathbb{Z}_p$, 15 $\mathbb{G}_1$, $\lvert\pi_{\mathsf{SNARK}'}\rvert$

2. $\mathcal{V} \to \mathcal{P}$: Sample and send a challenge $r \leftarrow\!\!\$\ \mathbb{Z}_p$.
3. $\mathcal{P}$ and $\mathcal{V}$: Compute $(\lambda^*, \gamma^*, v^*, \bar{a}^*, \bar{b}^*, \bar{c}^*, \boldsymbol{t}^*, u^*_{\mathsf{Plonk}}, \pi^*_{\mathsf{Plonk}}, \mu^*)$ part using the same method as in the folding scheme for quadratic relations, and (P^*, E^*_P, E^*) part as

$$P^* \leftarrow P_1 \cdot P_2^r, \quad E^*_P \leftarrow E_{P,1} \cdot T_P^r \cdot E_{P,2}^{r^2}, \quad E^* \leftarrow E_1 \cdot T^r \cdot E_2^{r^2}.$$

The non-uniform preprocessed quadratic Plonk relation satisfies perfect completeness and knowledge soundness in the random oracle model by the same hybrid-and-extraction argument as Theorem 5. The only changes are (i) accounting for the preprocessing constraints and (ii) bounding the additional quadratic terms, which do not affect the random-oracle programming or the extractor. We omit the full proof.

6 Performance Evaluation

6.1 Theoretical Analysis

We compare the performance of SnarkMux with common baselines. For simplicity, in all experimental results, we use "aug.", "com.", and "quad." to represent the augmented, committed, and quadratic versions of SnarkMux, respectively.

Counting conventions. In Table 2, we count verifier-received output sizes in numbers of $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, and $\mathbb{Z}_p$ elements. In Table 3 and Table 4, we report dominant operation counts (scalar multiplications in $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$, pairings P, hashes H, and random-oracle queries RO). For verifier costs in Table 4, we separate online verification from per-window instance processing (or verifier preprocessing for SnarkMux). For SnarkMux in Table 2, the reported size includes the final-step proof $\pi_{\mathsf{SNARK}'}$ output by the chosen IVC instantiation.

Table 3: Dominant operation counts of the prover side for aggregating a window of n proofs ("LC" denotes local computation. "RC" denotes recursive circuit.)

Scheme	**Prover cost (per window)**
TIPP [8]	$(2\ell + 10n)$ $\mathbb{G}_1$, $8n$ $\mathbb{G}_2$, $5n$ $\mathbb{G}_T$, 1 H, $17n$ P
SnarkPack [13]	$(2n + 4\log n)$ $\mathbb{G}_1$, $7n$ $\mathbb{G}_2$, $\log n$ RO, 2 H, $21n$ P
Mira [2]	LC: $(\ell + 6)n$ $\mathbb{G}_1$, n $\mathbb{G}_2$, n $\mathbb{G}_T$, n H, $4n$ P, $\mathsf{SNARK}'.\mathsf{P}$ RC: $3n$ $\mathbb{G}_1$, n $\mathbb{G}_2$, n $\mathbb{G}_T$, $n \log n$ H
SnarkMux (aug.)	LC: $7n$ $\mathbb{G}_1$, n $\mathbb{G}_2$, n $\mathbb{G}_T$, n RO, $2n$ P, $\mathsf{SNARK}'.\mathsf{P}$ RC: $3n$ $\mathbb{G}_1$, n $\mathbb{G}_T$, n RO, $2n$ H
SnarkMux (com.)	LC: $11n$ $\mathbb{G}_1$, n $\mathbb{G}_2$, n $\mathbb{G}_T$, n RO, $2n$ P, $\mathsf{SNARK}'.\mathsf{P}$ RC: $6n$ $\mathbb{G}_1$, n $\mathbb{G}_T$, n RO, $2n$ H
aPlonk [1]	$(3\zeta n + 6\zeta + 3n + 9\eta)$ $\mathbb{G}_1$, $12n$ $\mathbb{G}_2$, $7n$ P, $\log_2(3n)$ H
SnarkMux (lin.)	LC: $22n$ $\mathbb{G}_1$, $7n$ RO, $\mathsf{SNARK}'.\mathsf{P}$ RC: $22n$ $\mathbb{G}_1$, $7n$ RO, $2n$ H
SnarkMux (quad.)	LC: $50n$ $\mathbb{G}_1$, $7n$ RO, $\mathsf{SNARK}'.\mathsf{P}$ RC: $13n$ $\mathbb{G}_1$, $7n$ RO, $2n$ H

We first compare proof sizes in Table 2. Here ℓ denotes the public-input length for the circuit, and $|\pi_{\mathsf{SNARK}'}|$ denotes the proof size of the final SNARK SNARK' produced by the IVC. SnarkMux and Mira yield constant-size outputs, whereas the GIPA baselines grow as $O(\log n)$ in the window size n.

For Groth16 aggregation, Table 3 and Table 4 summarize the dominant prover and verifier costs. The "Ins. (pre)proc." column in Table 4 corresponds to one-time verifier preprocessing in SnarkMux (*i.e.*, $\mathcal{S}.\mathsf{VPrep}$) and to per-window instance aggregation in the baselines. We use $\mathsf{SNARK}'.\mathsf{P}$ and $\mathsf{SNARK}'.\mathsf{V}$ for the prover and verifier costs of the final SNARK SNARK', respectively. We focus on group and pairing operations that dominate these costs. In local computation, SnarkMux performs $2n$ pairings, whereas the directly comparable circuit-specific Mira relation performs $4n$ and TIPP and SnarkPack perform $17n$ and $21n$ pairings, respectively. In addition, SnarkMux incurs recursive-circuit overhead to certify that the delegated computation was executed correctly. In our design, each IVC step computes one $\mathbb{G}_T$ scalar multiplication in-circuit to preserve constant-size outputs. Appendix D discusses an alternative that removes $\mathbb{G}_T$ operations from the circuit at the cost of larger proof size. For verification, SnarkMux performs one-time preprocessing costing n hash evaluations, whereas TIPP and SnarkPack require per-window instance aggregation costing $n\ell$ group/field operations as shown in Table 4.

For folding Plonk proofs, Table 3 and Table 4 compare SnarkMux with aPlonk. We use aPlonk's notation ζ and η for the sizes of the circuit and meta-verification circuit, respectively (see [1]). Both provers scale linearly in n, and SnarkMux additionally incurs recursive-circuit overhead. On the verifier side, SnarkMux supports a preprocessing/online split with n hash evaluations in preprocessing and $O(1)$ online work. In contrast, aPlonk's verifier cost grows

Table 4: Dominant operation counts of the verifier side for aggregating a window of n proofs

Scheme	Online (per window)	Ins. (pre)proc.
TIPP [8]	ℓ $\mathbb{G}_1$, $12\log n$ $\mathbb{G}_T$, 1 H, 18 P	$n\ell$ $\mathbb{G}_1$
SnarkPack [13]	ℓ $\mathbb{G}_1$, 5 $\mathbb{G}_2$, $12\log n$ $\mathbb{G}_T$, $\log n$ RO, 2 H, 18 P	$n\ell$ $\mathbb{Z}_p$, ℓ $\mathbb{G}_1$
Mira [2]	ℓ $\mathbb{G}_1$, 4 $\mathbb{G}_T$, 4 P, $\mathsf{SNARK}'.\mathsf{V}$	$n\log n$ H
SnarkMux (aug.)	2ℓ $\mathbb{G}_1$, 1 RO, 1 H, 5 P, $\mathsf{SNARK}'.\mathsf{V}$	n H ($\mathcal{S}.\mathsf{VPrep}$)
SnarkMux (com.)	ℓ $\mathbb{G}_1$, 1 RO, 1 H, 5 P, $\mathsf{SNARK}'.\mathsf{V}$	n H ($\mathcal{S}.\mathsf{VPrep}$)
aPlonk [1]	$2\log_2(3n)$ $\mathbb{G}_1$, $2\log_2(3n)$ $\mathbb{G}_T$, $\log_2(3n)$ H, 4 P	$n\ell\log\ell$ $\mathbb{Z}_p$
SnarkMux (lin.)	2 $\mathbb{G}_1$, 1 RO, 1 H, 2 P, $\mathsf{SNARK}'.\mathsf{V}$	n H ($\mathcal{S}.\mathsf{VPrep}$)
SnarkMux (quad.)	22 $\mathbb{G}_1$, 1 RO, 1 H, 2 P, $\mathsf{SNARK}'.\mathsf{V}$	n H ($\mathcal{S}.\mathsf{VPrep}$)

Table 5: Verifier-received output (KB) versus window n

Window Size	16	32	64	128	256
Groth16 aggregation					
TIPP [8]	20	24	29	33	38
SnarkPack [13]	20	24	29	33	38
SnarkMux (aug.)	7.2	7.2	7.2	7.2	7.2
SnarkMux (com.)	0.8	0.8	0.8	0.8	0.8
Plonk aggregation					
aPlonk [1]	6.6	7.4	8.2	9.0	9.8
SnarkMux (lin.)	0.5	0.5	0.5	0.5	0.5
SnarkMux (quad.)	1.7	1.7	1.7	1.7	1.7

logarithmically with n and includes per-window instance processing as shown in Table 4.

6.2 Implementation and Evaluation

We implement SnarkMux in Rust[3] based on Nova's implemention [22]. We use the BN254/Grumpkin cycle for recursion, and evaluate aggregation for Groth16 and Plonk instantiated over BN254. We instantiate Hash with Poseidon at 128-bit security [15]. The final SNARK SNARK' in the IVC is Spartan [23]. All experiments run on an Intel i7-12700KF CPU (3.6 GHz) with 32 GB RAM.

Circuit constraints. Table 6 reports R1CS constraint counts per aggregation step on the BN254/Grumpkin cycle, comparing SnarkMux against the baseline (Figure 1a) that embeds the full $\mathsf{SNARK.V}$ inside the recursive circuit. The

[3] https://anonymous.4open.science/r/SnarkMux

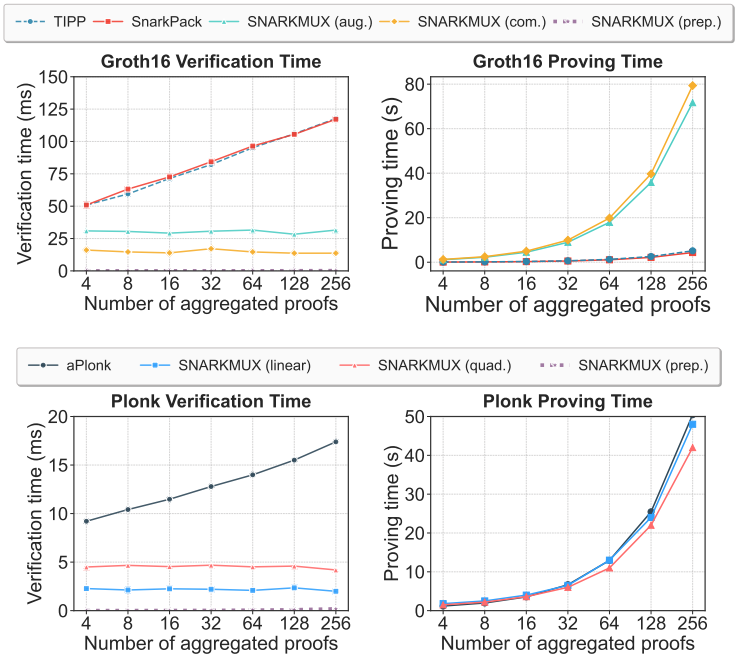
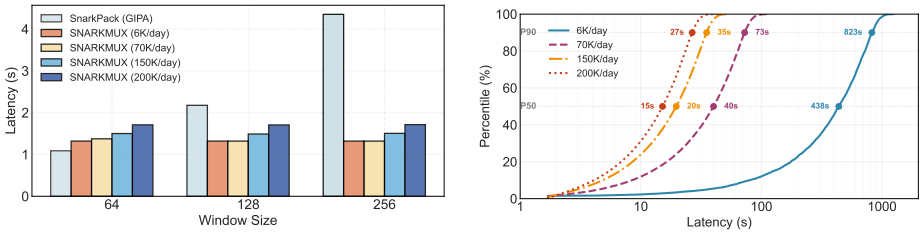


Fig. 5: Comparisons of the verification time and the proving time

primary circuit runs over BN254 and implements the step function F together with $\mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{V}$; the secondary circuit is the companion-curve (Grumpkin). Since SnarkMux employs cycle curves, the constraint counts of the two circuits are not directly summable for a fair comparison against the baseline. We define the *Primary red.* factor as the ratio of the baseline's constraints to the constraints of the primary circuit; it is not a ratio of the displayed totals because the two cycle-curve circuits are not directly summable. For Groth16, the in-circuit baseline costs 1,005,548 constraints per step. SnarkMux (aug.) reduces this to 27,794 constraints (36×) by deferring all pairing checks to the final verification. Within the primary circuit, the in-circuit $\mathbb{G}_T$ scalar multiplication that updates the error term contributes 20,845; the remaining part stems from scalar multiplications in $\mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{V}$ and $\mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{V}$. The committed variant SnarkMux (com.) costs 31,355 constraints per step (32×), as it replaces the ℓ-vector with three additional $\mathbb{G}_1$ multiplications in the primary circuit. For Plonk, the in-circuit baseline costs 208,122 per step. SnarkMux (lin.) reduces this to 31,980: the $\mathsf{Preprocess}^{\mathrm{lin}}$ procedure executes 22 $\mathbb{G}_1$ exponentiations in-circuit per step to derive the folding pair (P, Q). SnarkMux (quad.) achieves 22,989 per step (9×) by executing only 13 $\mathbb{G}_1$ exponentiations in-circuit. The secondary circuit, shared across all four SnarkMux variants, contributes a fixed 8,410-constraint overhead from the Nova companion-curve augmented circuit.

Table 6: R1CS constraint counts per aggregation step ($\ell = 100$ for all Groth16 instances. Counts are per-step and independent of window size n.)

	Scheme	Primary	Secondary	Total	Primary red.
Groth16	In-circuit Groth16.V	1,005,548	-	1,005,548	-
	SnarkMux (aug.)	27,794	8,410	36,204	≈36×
	SnarkMux (com.)	31,355	8,410	39,765	≈32×
Plonk	In-circuit Plonk.V	208,122	-	208,122	-
	SnarkMux (lin.)	31,980	8,410	40,390	≈6.5×
	SnarkMux (quad.)	22,989	8,410	31,399	≈9×



(a) Prover-side latency, measured as the time from the arrival of the last proof to the completion of final aggregation.

(b) Cumulative distribution of per-proof latency for a fixed window size $n = 64$

Fig. 6: Streaming latency versus window size n and per-proof latency CDF, comparing SnarkMux to SnarkPack

Standard performance. We evaluate SnarkMux and aPlonk for aggregating a window of n Groth16 and Plonk proofs. Table 5 reports proof sizes. For Groth16 with $\ell = 100$, SnarkMux outputs 0.8 KB (committed) and 7.2 KB (augmented). For Plonk, SnarkMux remains constant (0.5 KB for the linear variant and 1.7 KB for the quadratic variant), whereas aPlonk grows as $O(\log n)$. Figure 5 reports proving and verification times. For Groth16, SnarkMux incurs recursive-circuit overhead, yielding higher proving time (about 8.7× versus TIPP and SnarkPack in our measurements), while remaining practical (about 280 ms per proof). For Plonk, both SnarkMux and aPlonk scale linearly in n, and SnarkMux is 10–20% faster in our experiments. Verification time for SnarkMux remains constant (about 18–27 ms for Groth16 variants, and about 2–5 ms for Plonk variants), whereas aPlonk grows logarithmically and incurs increasing instance-processing overhead as n grows. SnarkMux supports verifier preprocessing (*e.g.*, about 0.2 ms to hash 256 instances in our setup) followed by $O(1)$ online checking.

Streaming performance. We further evaluate SnarkMux in a streaming scenario, motivated by blockchain-style deployments where proofs arrive over time and are processed online. As a simple baseline, we model proof arrivals

as a homogeneous Poisson process with constant rate, *i.e.*, inter-arrival times are i.i.d. exponential. We use this model as a controlled baseline to study the interaction between arrival jitter and batch-induced waiting time, not as a claim that it best fits any particular deployment. Poisson arrival models have been used for blockchain transaction arrivals in prior work [20,14].

To set concrete parameters for our streaming evaluation, we use public Zcash transaction throughput as a coarse reference point for proof-bearing workloads.[4] These statistics do not distinguish shielded and transparent activity, so we use them only to anchor the order of magnitude of arrival rates. The reference throughput is on the order of 6,000 transactions/day, with a historical peak on the order of 70,000 transactions/day. We additionally simulate higher loads of 150,000 and 200,000 transactions/day as stress tests. We model arrivals as a homogeneous Poisson process with rate r arrivals/second, so inter-arrival times are i.i.d. exponential with mean $1/r$. For throughputs $T \in \{6{,}000, 70{,}000, 150{,}000, 200{,}000\}$ per day, we set $r = T/86{,}400$ so $r \in \{\sim 0.07, \sim 0.81, \sim 1.74, \sim 2.31\}$. Each arrival corresponds to one instance–proof pair, and we treat per-arrival proving work as fixed to focus on latency induced by windowing and aggregation. This Poisson model is a controlled baseline and omits burstiness and temporal correlation (*e.g.*, user-driven spikes), which can further increase latency for batch aggregation. Our goal is to isolate the latency behavior introduced by batch versus streaming aggregation under a simple arrival process.

We first measure *tail aggregation latency*, defined as the time from the arrival of the last proof in a window to the completion of the final aggregate, for window sizes $n \in \{64, 128, 256\}$. We compare against SnarkPack, which is batch-oriented and therefore performs its aggregation work after the full window is available. As shown in Figure 6a, SnarkMux's tail latency remains stable across window sizes, because it performs incremental updates upon each arrival and leaves only constant-size residual work after the window closes. In contrast, SnarkPack's tail latency increases with larger windows due to its per-window aggregation computation. Notably, SnarkMux keeps tail latency within 2 seconds in our experiments, even at high arrival rates. These results indicate that streaming aggregation substantially reduces post-window completion latency relative to batch aggregation.

We additionally measure *per-proof latency*, *i.e.*, the time from a proof's arrival to its inclusion in a completed aggregate. We fix $n = 64$ and report the cumulative distribution in Figure 6b. As the arrival rate r increases, windows fill faster and per-proof waiting time decreases. At high rates (*e.g.*, $\geq$ 70,000 per day), the median (P50) per-proof latency is about 40 seconds and the 90th percentile (P90) is about 73 seconds. In this regime, the dominant component is waiting for the window to fill, while SnarkMux contributes only a small additional processing delay (about 1 second in our measurements). Reducing the window size further decreases per-proof latency at the cost of producing more aggregates. Overall, these results suggest that SnarkMux is well-suited to streaming deployments where proofs arrive over time.

[4] https://bitinfocharts.com/comparison/zcash-transactions.html

7 Conclusion

We presented SnarkMux, a proof aggregation scheme that reduces online verification to constant work by recasting aggregation as IVC with a two-folding structure. Its substantive target layer folds the Groth16 or Plonk verification relation directly, while Nova's relaxed-R1CS fold certifies each target-fold and binding-hash update. The key design is a preprocessing/online split that binds the ordered instance sequence to the aggregation proof, preventing instance substitution while enabling a streaming workflow with $O(1)$ aggregator state. We instantiated SnarkMux for Groth16 and Plonk, and our prototype confirms constant-size aggregates, a small online verifier, and improved end-to-end latency in streaming settings. Our guarantees are parameterized by the supported recursion horizon of the underlying instantiation. Scaling beyond this horizon is an orthogonal systems choice (*e.g.*, tree-style composition) rather than a change to the SnarkMux interface.

References

1. Ambrona, M., Beunardeau, M., Schmitt, A., Toledo, R.R.: a$PlonK$: Aggregated $PlonK$ from multi-polynomial commitment schemes. In: IWSEC. pp. 195–213 (2023)
2. Beal, J., Fisch, B.: Mira: Efficient folding for pairing-based arguments. IACR Cryptology ePrint Archive 2024/2025 (2024), https://eprint.iacr.org/2024/2025
3. Bowe, S., Chiesa, A., Green, M., Miers, I., Mishra, P., Wu, H.: Zexe: Enabling decentralized private computation. In: IEEE Symposium on Security and Privacy (S&P). pp. 947–964 (2020)
4. Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., Maxwell, G.: Bulletproofs: Short proofs for confidential transactions and more. In: IEEE Symposium on Security and Privacy (S&P). pp. 315–334 (2018)
5. Bünz, B., Chen, B.: Protostar: Generic efficient accumulation/folding for special sound protocols. In: ASIACRYPT Part II. pp. 77–110 (2023)
6. Bünz, B., Chiesa, A., Lin, W., Mishra, P., Spooner, N.: Proof-carrying data without succinct arguments. In: CRYPTO Part I,. pp. 681–710. Springer (2021)
7. Bünz, B., Chiesa, A., Mishra, P., Spooner, N.: Recursive proof composition from accumulation schemes. In: Theory of Cryptography Conference (TCC). pp. 1–18 (2020)
8. Bünz, B., Maller, M., Mishra, P., Tyagi, N., Vesely, P.: Proofs for inner pairing products and applications. In: ASIACRYPT Part III. pp. 65–97 (2021)
9. Cardozo, A.S., Williamson, Z.: EIP-1108: Reduce alt_bn128 precompile gas costs. Ethereum Improvement Proposals (May 2018), https://eips.ethereum.org/EIPS/eip-1108, created 2018-05-21
10. Dellepere, M., Mishra, P., Shirzad, A.: Garuda and pari: Faster and smaller SNARKs via equifficient polynomial commitments. IACR Cryptology ePrint Archive 2024/1245 (2024), https://eprint.iacr.org/2024/1245
11. Eagen, L., Gabizon, A.: ProtoGalaxy: Efficient ProtoStar-style folding of multiple instances. IACR Cryptology ePrint Archive 2023/1106 (2023)
12. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: Plonk: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. IACR Cryptology ePrint Archive 2019/953 (2019)

13. Gailly, N., Maller, M., Nitulescu, A.: SnarkPack: Practical SNARK aggregation. In: Financial Cryptography and Data Security (FC). pp. 203–229 (2022)
14. Gao, S., Li, Z., Peng, Z., Xiao, B.: Power adjusting and bribery racing: Novel mining attacks in the bitcoin system. In: Computer & Communications Security (CCS). pp. 833–850 (2019)
15. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A new hash function for zero-knowledge proof systems. In: USENIX Security Symposium. pp. 519–535 (2021)
16. Groth, J.: On the size of pairing-based non-interactive arguments. In: EUROCRYPT Part II. pp. 305–326 (2016)
17. Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: ASIACRYPT. pp. 177–194 (2010)
18. Kothapalli, A., Setty, S.: HyperNova: Recursive arguments for customizable constraint systems. In: CRYPTO Part X,. pp. 345–379 (2024)
19. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive zero-knowledge arguments from folding schemes. In: CRYPTO Part IV,. pp. 359–388 (2022)
20. Kwon, Y., Kim, D., Son, Y., Vasserman, E.Y., Kim, Y.: Be selfish and avoid dilemmas: Fork after withholding (FAW) attacks on bitcoin. In: Computer & Communications Security (CCS). pp. 195–209 (2017)
21. Polygon: Polygon zkEVM. https://github.com/0xpolygonhermez, https://github.com/0xpolygonhermez
22. Setty, S.: Nova project. https://github.com/microsoft/Nova, https://github.com/microsoft/Nova
23. Setty, S.: Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In: CRYPTO Part III,. pp. 704–737 (2020)
24. Srinivasan, S., Karantaidou, I., Baldimtsi, F., Papamanthou, C.: Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In: Computer & Communications Security (CCS). pp. 2719–2733 (2022)
25. Sun, Y.: zkPairing. https://github.com/yi-sun/circom-pairing#benchmarks (2023), https://github.com/yi-sun/circom-pairing#benchmarks
26. Tech., S.: Scroll. https://github.com/scroll-tech (2023), https://github.com/scroll-tech
27. Yang, J., Gao, S., Li, G., Song, R., Xiao, B.: Reducing gas consumption of Tornado cash and other smart contracts in Ethereum. In: TrustCom. pp. 921–926 (2022)
28. Zheng, T., Gao, S., Guo, Y., Xiao, B.: KiloNova: Non-uniform PCD with zero-knowledge property from generic folding schemes. IACR Cryptology ePrint Archive 2023/1579 (2023)

Supporting Material

A Formal Definitions

A.1 (zk-)SNARK

We use the syntax introduced in the preliminaries. The relation $\mathcal{R}$ is over tuples $(\mathsf{pp}, \mathsf{s}, u, w)$, where s is fixed before structure-dependent key generation.

Definition 8 (Perfect Completeness). *A SNARK satisfies perfect completeness if, for every PPT adversary $\mathcal{A}$,*

$$\Pr\left[\mathsf{SNARK.V}(\mathsf{vk}, u, \pi) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{SNARK.G}(1^\lambda), \\ (\mathsf{s}, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}, \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{SNARK.K}(\mathsf{pp}, \mathsf{s}), \\ \pi \leftarrow \mathsf{SNARK.P}(\mathsf{pk}, u, w) \end{array}\right] = 1.$$

Definition 9 (Knowledge Soundness). *A SNARK satisfies knowledge soundness if for all PPT adversaries $\mathcal{A}$ there exists a PPT extractor $\mathcal{E}$ such that for any randomness ρ*

$$\Pr\left[\begin{array}{l} \mathsf{SNARK.V}(\mathsf{vk}, u, \pi) = 1, \\ (\mathsf{pp}, \mathsf{s}, u, w) \notin \mathcal{R} \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{SNARK.G}(1^\lambda), \\ (\mathsf{s}, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}; \rho), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{SNARK.K}(\mathsf{pp}, \mathsf{s}), \\ w \leftarrow \mathcal{E}(\mathsf{pp}; \rho) \end{array}\right] = \mathsf{negl}(\lambda).$$

Definition 10 (Succinctness). *A SNARK system is succinct if the size of a proof π is polylogarithmic in the size of the witness w.*

Definition 11 (Zero-Knowledge). *A zk-SNARK satisfies zero-knowledge if there exists PPT simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$*

$$\left\{(\mathsf{pp}, \mathsf{s}, u, \pi) \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{SNARK.G}(1^\lambda), \\ (\mathsf{s}, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}, \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{SNARK.K}(\mathsf{pp}, \mathsf{s}), \\ \pi \leftarrow \mathsf{SNARK.P}(\mathsf{pk}, u, w) \end{array}\right\}$$
$$\approx \left\{(\mathsf{pp}, \mathsf{s}, u, \pi) \;\middle|\; \begin{array}{l} (\mathsf{pp}, \tau) \leftarrow \mathcal{S}(1^\lambda), \\ (\mathsf{s}, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}, \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{SNARK.K}(\mathsf{pp}, \mathsf{s}), \\ \pi \leftarrow \mathcal{S}(\mathsf{pp}, u, \tau) \end{array}\right\}.$$

A.2 Proof Aggregation

We write $\mathsf{vk}_{\mathsf{SNARK}}$ for the verification key of the original target SNARK contained in, or referenced by, the aggregate verification key vk. We use the aggregation interface from the preliminaries: the prover outputs the folded target pair (u^*, π^*), and the aggregate verifier receives that pair explicitly.

Definition 12 (Perfect Completeness). *A proof aggregation scheme for* SNARK *satisfies perfect completeness if, for every* $n \leq N_{\max}$ *and every PPT adversary* $\mathcal{A}$*,*

$$\Pr\left[\mathsf{AGG.V}\left(\mathsf{vk}, n, (u_i)_{i=1}^n, (u^*, \pi^*), \pi_{\mathsf{AGG}}\right) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{AGG.G}(1^\lambda), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{AGG.K}(\mathsf{pp}), \\ (u_i, \pi_i)_{i=1}^n \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}, \mathsf{vk}), \\ \mathsf{SNARK.V}(\mathsf{vk}_{\mathsf{SNARK}}, u_i, \pi_i) = 1 \;\forall i \in [n], \\ ((u^*, \pi^*), \pi_{\mathsf{AGG}}) \leftarrow \mathsf{AGG.P}\left(\mathsf{pk}, (u_i, \pi_i)_{i=1}^n\right) \end{array}\right]$$

$= 1.$

Definition 13 (Aggregate Knowledge Soundness). *A proof aggregation scheme satisfies aggregate knowledge soundness if, for every fixed* $n \leq N_{\max}$ *and every PPT adversary* $\mathcal{A}$*, there exists an expected PPT extractor* $\mathcal{E}^{\mathcal{A}}$ *such that*

$$\Pr\left[\begin{array}{l} \mathsf{AGG.V}\left(\mathsf{vk}, n, (u_i)_{i=1}^n, (u^*, \pi^*), \pi_{\mathsf{AGG}}\right) = 1 \\ \wedge \\ \exists i \in [n] : \mathsf{SNARK.V}(\mathsf{vk}_{\mathsf{SNARK}}, u_i, \pi_i) = 0 \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{AGG.G}(1^\lambda), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{AGG.K}(\mathsf{pp}), \\ ((u_i)_{i=1}^n, (u^*, \pi^*), \pi_{\mathsf{AGG}}) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}, \mathsf{vk}), \\ (\pi_i)_{i=1}^n \leftarrow \mathcal{E}^{\mathcal{A}}\left(\mathsf{pp}, \mathsf{pk}, \mathsf{vk}, (u_i)_{i=1}^n, (u^*, \pi^*), \pi_{\mathsf{AGG}}\right) \end{array}\right]$$

$\leq \mathsf{negl}(\lambda).$

A.3 Incrementally Verifiable Computation

Definition 14 (Perfect Completeness). *An IVC scheme satisfies perfect completeness if, for every PPT adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$*,*

$$\Pr\left[\mathsf{IVC.V}(\mathsf{vk}, i, z_0, z_i, \mathit{\Pi}_i) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{IVC.G}(1^\lambda), \\ F \leftarrow \mathcal{A}_0(\mathsf{pp}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{IVC.K}(\mathsf{pp}, F), \\ (i, z_0, z_{i-1}, \omega_{i-1}, \mathit{\Pi}_{i-1}) \leftarrow \mathcal{A}_1(\mathsf{pk}, \mathsf{vk}), \\ z_i \leftarrow F(z_{i-1}, \omega_{i-1}), \\ \mathsf{IVC.V}(\mathsf{vk}, i-1, z_0, z_{i-1}, \mathit{\Pi}_{i-1}) = 1, \\ \mathit{\Pi}_i \leftarrow \mathsf{IVC.P}\left(\mathsf{pk}, i, z_0, z_i, z_{i-1}, \omega_{i-1}, \mathit{\Pi}_{i-1}\right) \end{array}\right] = 1.$$

Definition 15 (Knowledge Soundness). *An IVC scheme satisfies knowledge soundness if, for every fixed* $n \leq N_{\max}$ *and every expected PPT adversary* $\mathcal{A} =$

$(\mathcal{A}_0, \mathcal{A}_1)$, *there exists an expected extractor* $\mathcal{E}^{\mathcal{A}}$ *such that*

$$\Pr\left[\begin{array}{l|l} \mathsf{IVC.V}(\mathsf{vk}, n, z_0, z, \Pi) = 1 \\ \wedge \\ z_n \neq z \end{array} \middle| \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{IVC.G}(1^\lambda), \\ F \leftarrow \mathcal{A}_0(\mathsf{pp}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{IVC.K}(\mathsf{pp}, F), \\ (z_0, z, \Pi) \leftarrow \mathcal{A}_1(\mathsf{pk}, \mathsf{vk}), \\ (\omega_0, \ldots, \omega_{n-1}) \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}, F, \mathsf{pk}, \mathsf{vk}, z_0, z, \Pi), \\ z_i \leftarrow F(z_{i-1}, \omega_{i-1}) \quad \forall i \in \{1, \ldots, n\} \end{array}\right]$$

$\leq \mathsf{negl}(\lambda)$.

The expected extraction time may be polynomial in λ *and exponential in this fixed* n*, as stated in Section 2.*

Definition 16 (Succinctness). *An IVC scheme is succinct if the proof size and verifier work are independent of the number of already-certified steps, up to the public step counter and the fixed supported recursion horizon.*

A.4 Folding Schemes

A folding scheme satisfies perfect completeness and knowledge soundness for the relation $\mathcal{R}$ being folded.

Definition 17 (Perfect Completeness). *A folding scheme satisfies perfect completeness if, for every PPT adversary* $\mathcal{A}$*,*

$$\Pr\left[(\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Fold.G}(1^\lambda), \\ (\mathsf{s}, (u_1, w_1), (u_2, w_2)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u_1, w_1) \in \mathcal{R}, \\ (\mathsf{pp}, \mathsf{s}, u_2, w_2) \in \mathcal{R}, \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Fold.K}(\mathsf{pp}, \mathsf{s}), \\ (u, w) \leftarrow \langle \mathcal{P}(\mathsf{pk}, w_1, w_2), \mathcal{V}(\mathsf{vk}) \rangle (u_1, u_2) \end{array}\right] = 1.$$

Definition 18 (Knowledge Soundness). *A folding scheme satisfies knowledge soundness if, for every expected PPT folding prover* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$*, there exists an expected extractor* $\mathcal{E}^{\mathcal{A}}$ *such that*

$$\Pr\left[\begin{array}{c} (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R} \\ \wedge \\ \big((\mathsf{pp}, \mathsf{s}, u_1, w_1) \notin \mathcal{R} \;\vee\; (\mathsf{pp}, \mathsf{s}, u_2, w_2) \notin \mathcal{R}\big) \end{array} \middle| \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Fold.G}(1^\lambda), \\ \mathsf{s} \leftarrow \mathcal{A}_0(\mathsf{pp}), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{Fold.K}(\mathsf{pp}, \mathsf{s}), \\ (u_1, u_2) \leftarrow \mathcal{A}_1(\mathsf{pk}, \mathsf{vk}), \\ (u, w) \leftarrow \langle \mathcal{A}_1, \mathcal{V}(\mathsf{vk}) \rangle (u_1, u_2), \\ (w_1, w_2) \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}, \mathsf{s}, \mathsf{pk}, \mathsf{vk}, u_1, u_2) \end{array}\right]$$

$\leq \mathsf{negl}(\lambda)$.

The extractor refers to the same accepting interaction as the folded pair. For a Fiat–Shamir fold, the experiment and extractor have access to the same random oracle.

A.5 Security of SnarkMux

The following definitions instantiate the aggregate interface with the verifier-preprocessing algorithm $\mathcal{S}.\mathsf{VPrep}$. The final folded pair (u^*, π^*) is checked by the target-relation verifier $\mathsf{V}_{\mathsf{tgt}}$, whereas extracted individual proofs are checked by the SNARK verifier $\mathsf{SNARK.V}$.

Definition 19 (Perfect Completeness). *Our SnarkMux proof aggregation scheme $\mathcal{S}$ satisfies perfect completeness if, for every $n \le N_{\max}$ and every PPT adversary $\mathcal{A}$,*

$$\Pr\left[\mathcal{S}.\mathsf{V}\left(\mathsf{vk}, n, h, (u^*, \pi^*), \pi_{\mathsf{AGG}}\right) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{S}.\mathsf{G}(1^\lambda), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}.\mathsf{K}(\mathsf{pp}), \\ (u_i, \pi_i)_{i=1}^n \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}, \mathsf{vk}), \\ \mathsf{SNARK.V}(\mathsf{vk}_{\mathsf{SNARK}}, u_i, \pi_i) = 1 \;\forall i \in [n], \\ h \leftarrow \mathcal{S}.\mathsf{VPrep}\left(\mathsf{vk}, (u_i)_{i=1}^n\right), \\ ((u^*, \pi^*), \pi_{\mathsf{AGG}}) \leftarrow \mathcal{S}.\mathsf{P}\left(\mathsf{pk}, (u_i, \pi_i)_{i=1}^n\right) \end{array}\right] = 1.$$

Definition 20 (Aggregate Knowledge Soundness). *Our SnarkMux scheme satisfies aggregate knowledge soundness if, for every fixed $n \le N_{\max}$ and every PPT adversary $\mathcal{A}$, there exists an expected extractor $\mathcal{E}^{\mathcal{A}}$ such that*

$$\Pr\left[\begin{array}{c} \mathcal{S}.\mathsf{V}\left(\mathsf{vk}, n, h, (u^*, \pi^*), \pi_{\mathsf{AGG}}\right) = 1 \\ \wedge \\ \exists i \in [n] : \mathsf{SNARK.V}(\mathsf{vk}_{\mathsf{SNARK}}, u_i, \pi_i) = 0 \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{S}.\mathsf{G}(1^\lambda), \\ (\mathsf{pk}, \mathsf{vk}) \leftarrow \mathcal{S}.\mathsf{K}(\mathsf{pp}), \\ ((u_i)_{i=1}^n, (u^*, \pi^*), \pi_{\mathsf{AGG}}) \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{pk}, \mathsf{vk}), \\ h \leftarrow \mathcal{S}.\mathsf{VPrep}\left(\mathsf{vk}, (u_i)_{i=1}^n\right), \\ (\pi_i)_{i=1}^n \leftarrow \mathcal{E}^{\mathcal{A}}\left(\mathsf{pp}, \mathsf{pk}, \mathsf{vk}, (u_i)_{i=1}^n, (u^*, \pi^*), \pi_{\mathsf{AGG}}\right) \end{array}\right]$$
$$\le \mathsf{negl}(\lambda).$$

Extraction is in the random-oracle model used by the folds and may be polynomial in λ and exponential in this fixed n.

B Details of zk-SNARKs

B.1 Groth16

We recall the Groth16 verifier equation in the notation used by the folding construction. Let $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, p, e)$ be bilinear groups of prime order p, and let $g \in \mathbb{G}_1$ and $h \in \mathbb{G}_2$ be generators. For $x \in \mathbb{Z}_p$, write $[x]_1 = g^x$, $[x]_2 = h^x$, and $[x]_T = e(g,h)^x$.

For public input $\boldsymbol{a} = (a_0, \ldots, a_\ell)$, with $a_0 = 1$, a Groth16 proof is $\pi_{\mathsf{Groth}} = (A, B, C) \in \mathbb{G}_1 \times \mathbb{G}_2 \times \mathbb{G}_1$. The verification key contains $(S_i)_{i=0}^{\ell} \in \mathbb{G}_1^{\ell+1}$, $[\alpha]_1 \in \mathbb{G}_1$, and $[\beta]_2, [\gamma]_2, [\delta]_2 \in \mathbb{G}_2$. The verifier accepts if

$$e(A, B) = e(C, [\delta]_2) \cdot e\left(\prod_{i=0}^{\ell} S_i^{a_i}, [\gamma]_2\right) \cdot e([\alpha]_1, [\beta]_2). \tag{6}$$

For compactness, the main construction writes $D = e([\alpha]_1, [\beta]_2)$.

B.2 Plonk

We recall the parts of the original Plonk verifier used by $\mathsf{Preprocess}^{\mathrm{lin}}$ and $\mathsf{Preprocess}^{\mathrm{quad}}$. The universal SRS contains $([1]_1, [x]_1, \ldots), ([1]_2, [x]_2)$, and the verification key contains commitments to the selector and permutation polynomials. A Plonk proof π_{Plonk} contains commitments

$$(A, B, C, Z, T_l, T_{mi}, T_h, W, V) \in \mathbb{G}_1^9$$

and claimed evaluations

$$(\bar{a}, \bar{b}, \bar{c}, \bar{s}_{\sigma_1}, \bar{s}_{\sigma_2}, \bar{z}_\omega) \in \mathbb{Z}_p^6.$$

The verifier derives transcript challenges

$$\begin{aligned}
&\beta \leftarrow \mathrm{Hash}(A, B, C),\\
&\theta \leftarrow \mathrm{Hash}(\beta),\\
&\alpha \leftarrow \mathrm{Hash}(\theta, Z),\\
&\lambda \leftarrow \mathrm{Hash}(\alpha, T_l, T_{mi}, T_h),\\
&v \leftarrow \mathrm{Hash}(\lambda, \bar{a}, \bar{b}, \bar{c}, \bar{s}_{\sigma_1}, \bar{s}_{\sigma_2}, \bar{z}_\omega),\\
&\gamma \leftarrow \mathrm{Hash}(v, W, V).
\end{aligned}$$

Using the notation of Section 5, define

$$\begin{aligned}
M \leftarrow\ & A^{v} \cdot B^{v^2} \cdot C^{v^3} \cdot S_{\sigma_1}^{v^4} \cdot S_{\sigma_2}^{v^5},\\
N \leftarrow\ & Q_M^{\bar{a}\bar{b}} \cdot Q_L^{\bar{a}} \cdot Q_R^{\bar{b}} \cdot Q_O^{\bar{c}} \cdot Q_C\\
&\cdot Z^{\alpha(\bar{a}+\beta\lambda+\theta)(\bar{b}+\beta k_1\lambda+\theta)(\bar{c}+\beta k_2\lambda+\theta)+\alpha^2 L_1(\lambda)+\gamma}\\
&\cdot S_{\sigma_3}^{-\alpha(\bar{a}+\beta\bar{s}_{\sigma_1}+\theta)(\bar{b}+\beta\bar{s}_{\sigma_2}+\theta)\beta\bar{z}_\omega}\\
&\cdot T_l^{-z_H(\lambda)} \cdot T_{mi}^{-z_H(\lambda)\lambda^m} \cdot T_h^{-z_H(\lambda)\lambda^{2m}}.
\end{aligned}$$

Let

$$\begin{aligned}
r_0 \leftarrow\ & u(\lambda) - \alpha^2 L_1(\lambda)\\
&- \alpha(\bar{a} + \beta\bar{s}_{\sigma_1} + \theta)(\bar{b} + \beta\bar{s}_{\sigma_2} + \theta)(\bar{c} + \theta)\bar{z}_\omega,\\
F \leftarrow\ & \left[-r_0 + v\bar{a} + v^2\bar{b} + v^3\bar{c} + v^4\bar{s}_{\sigma_1} + v^5\bar{s}_{\sigma_2} + \gamma\bar{z}_\omega\right]_1.
\end{aligned}$$

The final randomized batched-opening check can be written as

$$e(WV^\gamma, [x]_2) = e\left(W^\lambda V^{\gamma\lambda\omega} MNF^{-1}, [1]_2\right). \tag{7}$$

As in ordinary Plonk, the randomizers used for batching are transcript-derived; our fresh-input conversion re-derives those values before a proof enters the running folded relation. The target Plonk instantiation retains Plonk's universal SRS.

B.3 Construction of IVC

We show the formal construction of IVC in Figure 7.

$\mathsf{pp} \leftarrow \mathsf{IVC.G}(1^\lambda)$

1 : $\mathsf{pp}_{\mathsf{FS}} \leftarrow \mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{G}(1^\lambda), \quad \mathsf{pp}_{\mathsf{FC}} \leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{G}(1^\lambda),$
2 : **output** $\mathsf{pp} \leftarrow (\mathsf{pp}_{\mathsf{FS}}, \mathsf{pp}_{\mathsf{FC}})$.

$(\mathsf{pk}, \mathsf{vk}) \leftarrow \mathsf{IVC.K}(\mathsf{pp}, \mathsf{RC})$

1 : $(\mathsf{pk}_{\mathsf{FS}}, \mathsf{vk}_{\mathsf{FS}}) \leftarrow \mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{K}(\mathsf{pp}_{\mathsf{FS}}, \mathsf{s}_{\mathsf{S}})$,
2 : $(\mathsf{pk}_{\mathsf{FC}}, \mathsf{vk}_{\mathsf{FC}}) \leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{K}(\mathsf{pp}_{\mathsf{FC}}, \mathsf{s}_{\mathsf{C}})$,
3 : **output** $(\mathsf{pk}, \mathsf{vk}) \leftarrow \big((\mathsf{RC}, \mathsf{pk}_{\mathsf{FS}}, \mathsf{pk}_{\mathsf{FC}}), (\mathsf{RC}, \mathsf{vk}_{\mathsf{FS}}, \mathsf{vk}_{\mathsf{FC}})\big)$.

$\Pi_i \leftarrow \mathsf{IVC.P}(\mathsf{pk}, i, (u_i, \pi_i), \Pi_{i-1})$

1 : **if** $i = 1$
2 : $\quad (u_0^*, \pi_0^*) \leftarrow (u_\perp^*, \pi_\perp^*),\ (u_{\mathsf{C},0}^*, w_{\mathsf{C},0}^*) \leftarrow (u_{\mathsf{C},\perp}^*, w_{\mathsf{C},\perp}^*)$;
3 : **else**
4 : $\quad$ **parse** Π_{i-1} **as** $\big(h_{i-1}, (u_{i-1}^*, \pi_{i-1}^*), (u_{\mathsf{C},i-1}, w_{\mathsf{C},i-1}), (u_{\mathsf{C},i-1}^*, w_{\mathsf{C},i-1}^*)\big)$,
5 : $\quad$ **compute** $h_i, (u_i^*, \pi_i^*), (u_{\mathsf{C},i}^*, w_{\mathsf{C},i}^*)$ based on Equation (2),
6 : $\quad (u_{\mathsf{C},i}, w_{\mathsf{C},i}) \leftarrow \mathsf{tr}\big(\mathsf{RC}(\mathsf{vk}, i, h_{i-1}, u_i, u_{i-1}^*, u_{\mathsf{C},i-1}, u_{\mathsf{C},i-1}^*)\big)$,
7 : $\quad$ **output** $\Pi_i \leftarrow \big(h_i, (u_i^*, \pi_i^*), (u_{\mathsf{C},i}, w_{\mathsf{C},i}), (u_{\mathsf{C},i}^*, w_{\mathsf{C},i}^*)\big)$.

$0/1 \leftarrow \mathsf{IVC.V}(\mathsf{vk}, i, \Pi_i)$

1 : $\quad$ **parse** Π_i **as** $\big(h_i, (u_i^*, \pi_i^*), (u_{\mathsf{C},i}, w_{\mathsf{C},i}), (u_{\mathsf{C},i}^*, w_{\mathsf{C},i}^*)\big)$,
2 : $\quad$ **check** $u_{\mathsf{C},i}.x = \mathrm{Hash}(\mathsf{vk}, i, h_i, u_i^*, u_{\mathsf{C},i}^*)$,
3 : $\quad$ **check** π_i^* is a satisfying proof to u_i^*,
4 : $\quad$ **check** $w_{\mathsf{C},i}, w_{\mathsf{C},i}^*$ are satisfying witnesses to $u_{\mathsf{C},i}, u_{\mathsf{C},i}^*$,
5 : $\quad$ **check** $u_{\mathsf{C},i}$ is a non-relaxed instance.

Fig. 7: IVC algorithms instantiated by SnarkMux (setup, per-step update, and verification): s_{S} and s_{C} denote the common structure of the SNARK instance and the recursive-circuit instance, respectively. Function $\mathsf{tr}(\cdot)$ maps an executed recursive circuit RC to an instance-witness pair (u, w) for the next step.

C Formal Proofs

C.1 Lemmas

Lemma 1 (Forking Lemma for Folding Schemes [19]). *Consider a $(2k+1)$-move folding scheme $\mathsf{Fold} = (\mathsf{G}, \mathsf{K}, \mathcal{P}, \mathcal{V})$. The scheme satisfies knowledge soundness if there exists a PPT extractor $\mathcal{E}$ such that, for all input instance pairs (u_1, u_2), $\mathcal{E}$ outputs satisfying witnesses (w_1, w_2) with probability $1 - \mathsf{negl}(\lambda)$ when given public parameters pp, a structure s, and an $(n_1, \ldots, n_k)$-tree of accepting transcripts together with the corresponding folded instance-witness pairs (u, w). The tree contains n_1 transcripts with fresh randomness in the verifier's first message; for each such transcript, it contains n_2 transcripts with fresh ran-*

domness in the verifier's second message; and so on. The total number of leaves $\prod_{i=1}^{k} n_i$ *is bounded by* $\mathsf{poly}(\lambda)$.

Lemma 2 (Binding of Verifier Preprocessing). *Assume* Hash *is collision resistant. For every fixed* $n \le N_{\max}$*, the binding claim output by* $\mathcal{S}.\mathsf{VPrep}$ *computationally binds the ordered sequence of encoded instances. More precisely, for every PPT adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{c|l} (u_i)_{i=1}^n \neq (u'_i)_{i=1}^n & ((u_i)_{i=1}^n, (u'_i)_{i=1}^n) \leftarrow \mathcal{A}(1^\lambda), \\ \wedge & h_n \leftarrow \mathcal{S}.\mathsf{VPrep}\left(\mathsf{vk}, (u_i)_{i=1}^n\right), \\ h_n = h'_n & h'_n \leftarrow \mathcal{S}.\mathsf{VPrep}\left(\mathsf{vk}, (u'_i)_{i=1}^n\right) \end{array}\right] \le \mathsf{negl}(\lambda).$$

For non-uniform aggregation, the encoded element at position i *is the pair* (vk_i, u_i).

Proof. Let h_i and h'_i be the two hash chains. Both use the same fixed initial value h_0, the same claimed length n, and the domain-separated, index-aware, unambiguous encoding specified in Section 3.2. Let j be the first index for which $u_j \neq u'_j$. Then $h_{j-1} = h'_{j-1}$, while the inputs to the j-th hash call are distinct because the encoded instances differ. If $h_j = h'_j$, these two inputs immediately give a collision in Hash.

Otherwise, the two chains differ at position j but agree again at position n, because $h_n = h'_n$. Let $k > j$ be the least index for which $h_k = h'_k$. Then $h_{k-1} \neq h'_{k-1}$, so the inputs to the k-th hash calls are distinct, while their outputs are equal. This again yields a collision in Hash. Thus every successful binding adversary induces a collision, proving the claim. The same argument applies to non-uniform aggregation after replacing each encoded u_i by (vk_i, u_i).

C.2 Proof of IVC for Proof Aggregation

We make explicit why the two-folding state used by SnarkMux defines a valid IVC up to the supported recursion horizon.

Perfect completeness. We prove completeness by induction on the step index i. For the base case, the fixed dummy target pair $(u^*_\perp, \pi^*_\perp)$ and fixed dummy circuit pair $(u^*_{\mathsf{C},\perp}, w^*_{\mathsf{C},\perp})$ satisfy their respective relations by construction, and the initial hash is the fixed value $h_\perp$.

Assume the state after step $i-1$ is satisfying. For an honestly generated fresh pair (u_i, π_i), any proof-system-specific fresh-input conversion is checked by the recursive circuit. Perfect completeness of $\mathsf{Fold}_{\mathsf{SNARK}}$ gives a satisfying target pair

$$(u^*_i, \pi^*_i) \leftarrow \mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{P}\left(\mathsf{pk}_{\mathsf{FS}}, (u_i, \pi_i), (u^*_{i-1}, \pi^*_{i-1})\right).$$

The recursive circuit computes the same folded target instance using $\mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{V}$, updates $h_i = \mathrm{Hash}(u_i, h_{i-1})$, and produces a satisfying fresh circuit pair $(u_{\mathsf{C},i}, w_{\mathsf{C},i})$. Perfect completeness of $\mathsf{Fold}_{\mathsf{R1CS}}$ then gives

$$(u^*_{\mathsf{C},i}, w^*_{\mathsf{C},i}) \leftarrow \mathsf{Fold}_{\mathsf{R1CS}}.\mathsf{P}\left(\mathsf{pk}_{\mathsf{FC}}, (u_{\mathsf{C},i-1}, w_{\mathsf{C},i-1}), (u^*_{\mathsf{C},i-1}, w^*_{\mathsf{C},i-1})\right).$$

Hence all components of Π_i are satisfying, completing the induction.

Knowledge soundness. Fix $n \le N_{\max}$ and an accepting final IVC state. Extraction proceeds from the outer circuit fold inward. The final accepting circuit-fold instance admits, by knowledge soundness of $\mathsf{Fold}_{\mathsf{R1CS}}$, witnesses for the fresh circuit instance and the preceding running circuit instance. The extracted fresh circuit witness certifies the step relation, including:

1. the hash update $h_i = \mathrm{Hash}(u_i, h_{i-1})$;
2. any required fresh-input conversion for the target proof system;
3. the verifier-side target fold $u_i^* = \mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{V}(\mathsf{vk}_{\mathsf{FS}}, u_i, u_{i-1}^*)$; and
4. the verifier-side circuit fold that links the preceding circuit state.

Applying the $\mathsf{Fold}_{\mathsf{SNARK}}$ extractor to the corresponding accepting target fold recovers satisfying endpoint proofs for the fresh target instance and the preceding running target instance. Repeating this argument for $i = n, n-1, \ldots, 1$ exposes a satisfying fresh target proof for every step and a consistent sequence of circuit witnesses. The accumulated extraction error over fixed n remains negligible. The recursive extraction may take time exponential in fixed n, which is the standard bounded-horizon qualification already stated in the security model.

The hash-chain binding of Lemma 2 ensures that the extracted sequence is the same ordered sequence represented by the verifier's preprocessed claim. Thus the IVC state cannot certify a different ordered window without either a hash collision or a failure of one of the invoked knowledge-soundness properties.

Succinctness. Each step stores only a constant-size hash value, one target folded pair, one fresh circuit pair, and one running circuit folded pair. The state size is therefore independent of the number of already-processed proofs within the supported recursion horizon. The final compression by SNARK' in Section 3 converts the last circuit witness into the constant-size aggregation proof component used by the online verifier.

C.3 Proof of Theorem 2

We prove perfect completeness and knowledge soundness for the augmented relaxed Groth16 folding relation. Write the two input instances as

$$u_j = (\boldsymbol{a}_j, \mu_j, E_j, R_j, \boldsymbol{t}_j, \kappa_j), \qquad \pi_j = (A_j, B_j, C_j), \qquad j \in \{1, 2\}.$$

For an instance-proof pair (u, π), define its relation error by

$$\begin{aligned} \Delta(u, \pi) := e(A, B) \cdot e(C, [\delta]_2)^{-\mu} \cdot e\left(\prod_{k=0}^{\ell} S_k^{a_k}, [\gamma]_2\right)^{-\mu} \cdot D^{-\mu^2} \\ \cdot E^{-1} \cdot e(R, [\delta]_2)^{-1} \cdot e\left(\prod_{k=0}^{\ell} S_k^{t_k}, [\gamma]_2\right)^{-1} \cdot D^{-\kappa}. \end{aligned}$$

The pair is satisfying exactly when $\Delta(u, \pi) = 1_{\mathbb{G}_T}$.

Perfect completeness. The prover first computes

$$\begin{aligned} T' &\leftarrow e(A_1, B_2) \cdot e(A_2, B_1), \quad & R &\leftarrow C_1^{-\mu_2} C_2^{-\mu_1}, \\ \boldsymbol{t} &\leftarrow -\mu_2 \boldsymbol{a}_1 - \mu_1 \boldsymbol{a}_2, & \kappa &\leftarrow -2\mu_1\mu_2. \end{aligned}$$

For challenge r, the folded values are

$$\begin{aligned} \boldsymbol{a}^* &\leftarrow \boldsymbol{a}_1 + r \cdot \boldsymbol{a}_2, & \mu^* &\leftarrow \mu_1 + r \cdot \mu_2, \\ E^* &\leftarrow E_1 \cdot (T')^r \cdot E_2^{r^2}, & R^* &\leftarrow R_1 \cdot R^r \cdot R_2^{r^2}, \\ \boldsymbol{t}^* &\leftarrow \boldsymbol{t}_1 + r \cdot \boldsymbol{t} + r^2 \cdot \boldsymbol{t}_2, & \kappa^* &\leftarrow \kappa_1 + r \cdot \kappa + r^2 \cdot \kappa_2, \\ A^* &\leftarrow A_1 A_2^r, & B^* &\leftarrow B_1 B_2^r, \\ C^* &\leftarrow C_1 C_2^r. \end{aligned}$$

By bilinearity,

$$\begin{aligned} e(A^*, B^*) &= e(A_1, B_1) \cdot \big(e(A_1, B_2) e(A_2, B_1)\big)^r \cdot e(A_2, B_2)^{r^2}, \\ e(C^*, [\delta]_2)^{-\mu^*} &= e(C_1, [\delta]_2)^{-\mu_1} \cdot e(R, [\delta]_2)^r \cdot e(C_2, [\delta]_2)^{-\mu_2 r^2}, \\ e\left(\prod_{k=0}^{\ell} S_k^{a_k^*}, [\gamma]_2\right)^{-\mu^*} &= e\left(\prod_{k=0}^{\ell} S_k^{a_{1,k}}, [\gamma]_2\right)^{-\mu_1} \cdot e\left(\prod_{k=0}^{\ell} S_k^{t_k}, [\gamma]_2\right)^{r} \cdot e\left(\prod_{k=0}^{\ell} S_k^{a_{2,k}}, [\gamma]_2\right)^{-\mu_2 r^2}, \\ D^{-(\mu^*)^2} &= D^{-\mu_1^2} \cdot D^{\kappa r} \cdot D^{-\mu_2^2 r^2}. \end{aligned}$$

Combining these identities with the definitions of E^*, R^*, $\boldsymbol{t}^*$, and κ^* gives

$$\Delta(u^*, \pi^*) = \Delta(u_1, \pi_1) \cdot \Delta(u_2, \pi_2)^{r^2}. \tag{8}$$

The cross terms cancel by construction. If both inputs are satisfying, the right-hand side equals $1_{\mathbb{G}_T}$, proving perfect completeness.

Knowledge soundness. We analyze the interactive protocol; Fiat–Shamir gives the ROM statement. By Lemma 1, a successful prover can be rewound to a three-branch fork with the same pre-challenge message $T = (T', R, \boldsymbol{t}, \kappa)$ and three distinct challenges r_1, r_2, r_3. Let

$$\big(T, r_i, (\boldsymbol{a}_i^*, \mu_i^*, E_i^*, R_i^*, \boldsymbol{t}_i^*, \kappa_i^*, A_i^*, B_i^*, C_i^*)\big), \qquad i \in \{1, 2, 3\},$$

be the accepting transcripts. The linearly folded proof components interpolate unique endpoint values $(\widehat{A}_1, \widehat{B}_1, \widehat{C}_1)$ and $(\widehat{A}_2, \widehat{B}_2, \widehat{C}_2)$ satisfying

$$\begin{aligned} A_i^* &= \widehat{A}_1 \widehat{A}_2^{r_i}, \\ B_i^* &= \widehat{B}_1 \widehat{B}_2^{r_i}, \\ C_i^* &= \widehat{C}_1 \widehat{C}_2^{r_i}, \qquad i \in \{1, 2, 3\}. \end{aligned}$$

The same interpolation recovers the endpoint scalar and group components of the two input instances.

For each accepting branch,

$$\Delta(u_i^*, \pi_i^*) = 1_{\mathbb{G}_T}. \tag{9}$$

After substituting the folding rules, the quotient relation in Equation 9 is a degree-at-most-two function of the challenge in the prime-order target group. Equivalently, it has the form

$$\Delta_0 \cdot \Delta_1^r \cdot \Delta_2^{r^2} = 1_{\mathbb{G}_T}.$$

The three accepting equations for distinct challenges form an invertible Vandermonde system over $\mathbb{Z}_p$. Therefore each coefficient is the identity, in particular

$$\Delta_0 = \Delta(u_1, \pi_1) = 1_{\mathbb{G}_T}, \qquad \Delta_2 = \Delta(u_2, \pi_2) = 1_{\mathbb{G}_T}.$$

Thus both extracted endpoint pairs satisfy the augmented relaxed Groth16 relation. This proves knowledge soundness of the interactive fold, and the Fiat–Shamir version follows in the random oracle model with the usual forking loss.

C.4 Proof of Theorem 5

We give the full algebraic completeness and extraction argument for the uniform quadratic Plonk relation. For $j \in \{1, 2\}$, write

$$\begin{aligned} P_j &:= W_j^{\mu_j} V_j^{\gamma_j}, \\ Q_j &:= W_j^{\lambda_j} V_j^{t_{11,j}} M_j N_j [-\mu_j t_{10,j}]_1. \end{aligned}$$

Define the relation error

$$\Delta_j := e(P_j, [x]_2) \cdot e(Q_j, [1]_2)^{-1} \cdot e(E_{P,j}, [x]_2)^{-1} \cdot e(E_{Q,j}, [1]_2).$$

The j-th input satisfies Definition 6 exactly when $\Delta_j = 1_{\mathbb{G}_T}$.

Perfect completeness. The prover computes the cross terms T_P and T_Q exactly as in Section 5. Expanding the folded scalar and group values gives

$$\begin{aligned} P^* &= P_1 \cdot T_P^r \cdot P_2^{r^2}, \\ Q^* &= Q_1 \cdot T_Q^r \cdot Q_2^{r^2}. \end{aligned}$$

The error terms are folded with the identical quadratic pattern:

$$\begin{aligned} E_P^* &= E_{P,1} \cdot T_P^r \cdot E_{P,2}^{r^2}, \\ E_Q^* &= E_{Q,1} \cdot T_Q^r \cdot E_{Q,2}^{r^2}. \end{aligned}$$

Consequently,

$$\begin{aligned} &e(P^*, [x]_2) \cdot e(Q^*, [1]_2)^{-1} \cdot e(E_P^*, [x]_2)^{-1} \cdot e(E_Q^*, [1]_2) \\ &\quad = \Delta_1 \cdot \Delta_2^{r^2}. \end{aligned}$$

If both inputs are satisfying, this value is the identity, proving perfect completeness. The use of W^{λ}, rather than W^{γ}, is essential in the expansion because it is the term inherited from Equation 7.

Knowledge soundness. Consider an adversarially chosen pair of relaxed quadratic Plonk instances. The witness of the folded target relation is $\perp$; the substantive obligation is therefore to show that an accepting fold implies satisfying endpoint instances. By Lemma 1, use a three-branch fork with the same pre-challenge pair (T_P, T_Q) and three distinct challenges r_1, r_2, r_3. For each branch i, acceptance gives

$$e(P_i^*, [x]_2) \cdot e(Q_i^*, [1]_2)^{-1} = e(E_{P,i}^*, [x]_2) \cdot e(E_{Q,i}^*, [1]_2)^{-1}. \tag{10}$$

After moving the right-hand side to the left and expanding the fold, the error has the form

$$\Delta(r) = \Delta_0 \cdot \Delta_T^r \cdot \Delta_2^{r^2} \in \mathbb{G}_T,$$

where Δ_0 and Δ_2 are exactly the two endpoint relation errors, and Δ_T is the cross-term coefficient determined by (T_P, T_Q). Because $\Delta(r_i) = 1_{\mathbb{G}_T}$ for three distinct challenges, the three equations form an invertible Vandermonde system over $\mathbb{Z}_p$. Hence

$$\Delta_0 = \Delta_T = \Delta_2 = 1_{\mathbb{G}_T}.$$

In particular, both endpoint instances satisfy Definition 6. The extracted witnesses are $\perp$.

For a fresh Plonk pair, the recursive circuit separately certifies

$$(\lambda, \gamma, v, \bar{a}, \bar{b}, \bar{c}, \boldsymbol{t}) = \mathsf{Preprocess}^{\mathrm{quad}}(\mathsf{vk}_{\mathsf{Plonk}}, u_{\mathsf{Plonk}}, \pi_{\mathsf{Plonk}})$$

and initializes $\mu = 1$ and $E_P = E_Q = [0]_1$. Thus the extraction argument for the running folded relation does not require the folded state itself to be the Fiat–Shamir transcript of a single Plonk proof. The Fiat–Shamir version follows in the random oracle model with the usual forking loss.

D Discussion

Limited IVC step (window size). Our approach is based on IVC, and knowledge extraction may take time polynomial in λ and exponential in n. Accordingly, all security claims for a single aggregation window should be read for any fixed $n \leq N_{\max}$. This limitation is inherited from IVC-based designs and is not introduced by our target-SNARK folding constructions. One possible way to scale across windows is to use a proof-carrying-data (PCD) architecture that organizes computations or aggregations as a tree rather than a single linear chain. In the current SnarkMux structure, a recursive step can take as an additional input either a new proof to aggregate or an output from a previous step, allowing two previous steps or target-SNARK proofs to be linked. This yields a binary aggregation tree. If $\mathsf{Fold}_{\mathsf{SNARK}}$ supports k-folding, as in our linear Plonk construction, the additional input is handled naturally. For a two-input fold, one can instead execute two $\mathsf{Fold}_{\mathsf{SNARK}}.\mathsf{V}$ operations sequentially inside the recursive circuit. The

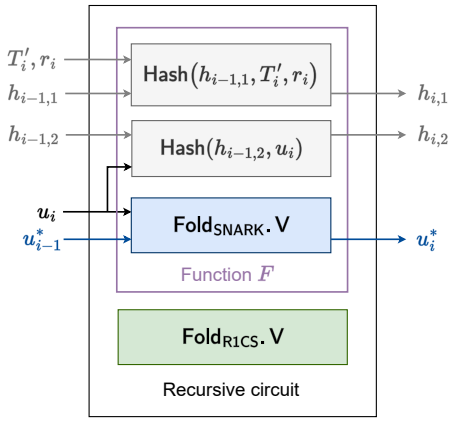


Fig. 8: Alternative design: off-circuit $\mathbb{G}_T$ computation with in-circuit hash binding.

resulting security argument would rely on the underlying PCD rather than on a longer linear IVC execution. We do not implement this variant because the additional target-fold verification increases recursive-circuit cost, particularly for our Groth16 constructions.

Security assumptions. SnarkMux inherits the random-oracle assumptions used to make its folds non-interactive and the collision-resistance assumption used for instance binding. It does not remove the setup assumptions of the target proof system: our Groth16 instantiation retains a circuit-specific SRS, and our Plonk instantiations retain Plonk's universal SRS. The Nova/Spartan recursion layer does not add a target-specific GIPA setup. These choices follow the same recursion setting as Nova-like designs [19,18,28,5,11].

Decoupling $\mathbb{G}_T$ operations from the recursive circuit. When SnarkMux aggregates Groth16 proofs, the recursive circuit computes

$$E^* \leftarrow E_1 \cdot (T')^r \cdot E_2^{r^2}$$

in every IVC step, where $T' \in \mathbb{G}_T$ and the fresh second input has $E_2 = [0]_T$. This requires one $\mathbb{G}_T$ scalar multiplication in the circuit per round. An alternative is to treat each step's (T'_i, r_i) as witnesses and bind them in-circuit by hashing. Such a design must enforce both (i) consistency between the off-circuit and in-circuit values (T'_i, r_i), and (ii) the correct update of each E^*_i. A corresponding relation is

$$\mathcal{R} = \left[\perp, \perp, (h_{n,1}, E^*_n), (\boldsymbol{T'}, \boldsymbol{r}) \;\middle|\; \begin{array}{l} \forall i \in [n], \quad E^*_i \leftarrow E_{i,1} \cdot (T'_i)^{r_i} \cdot E_{i,2}^{r_i^2}, \\ h_{i,1} \leftarrow \mathrm{Hash}(h_{i-1,1}, T'_i, r_i) \end{array} \right].$$

The final value $h_{n,1}$ is included in π_{AGG}. A straightforward verifier can receive all pairs $(T'_i, r_i)_{i=1}^n$, recompute E^*_n and the hash chain, and compare the resulting digest with $h_{n,1}$. This, however, makes the verifier-received output $O(n)$.

There is a possible compression route. Because the initial error contribution is the identity, the final value can be written as

$$E_n^* = (T_1')^{r_1} \cdot (T_2')^{r_2} \cdots (T_n')^{r_n} = (\boldsymbol{T}')^{\boldsymbol{r}}.$$

An inner-product argument such as Bulletproofs [4] could prove this relation compactly. However, an additional glue argument would still be needed to show that the vectors $\boldsymbol{T}'$ and $\boldsymbol{r}$ used by the inner-product argument are exactly those bound by the IVC execution. Even before accounting for such glue, a standard Bulletproofs-style argument incurs $O(\log n)$ communication, which conflicts with our constant-size-output objective.

Given these tradeoffs, we retain the construction from Section 4, in which the recursive circuit performs one $\mathbb{G}_T$ operation per IVC step. In our accounting, this is approximately the cost of 12 $\mathbb{G}_1$ scalar multiplications.