Flock: Fast Proving for Batch Boolean Computations

Benedikt Bünz* Ron D. Rothblum† William Wang‡

June 25, 2026

Abstract

For many applications of SNARKs, a key bottleneck is proving large batches of standard cryptographic hash evaluations, such as SHA-256, Keccak, or BLAKE3. We introduce Flock, a hash-based SNARK for extremely fast proving of such batched Boolean computations. Flock proves batches of the same R1CS circuit (plus input/output relations between them), can prove hash-chains and Merkle path openings, and in principle can be extended to full-fledged hash-based signature verification. At its core, Flock combines new optimizations for the lincheck and zerocheck protocols with an aggressively optimized proof-of-concept implementation co-designed by coding agents.

On a single core of an M4 Max processor, Flock proves 82k evaluations of the BLAKE3 compression function, 42k SHA-256 compressions, and 30k Keccak permutations per second — less than a 250× overhead over native execution. On ten cores, throughput exceeds 660k BLAKE3 compressions per second; in proving SHA-256, Flock is more than 9× faster than Binius64, the prior state of the art, and 1000× faster than Spartan.

*NYU, Espresso Systems. Email: bb@nyu.edu
†Succinct. Email: rothblum@gmail.com
‡NYU. Email: ww@priv.pub

Contents

1 Introduction

Succinct non-interactive arguments of knowledge (SNARKs) let an untrusted prover convince a verifier that a computation was executed correctly, by sending a short certificate that the verifier checks in time that is *much* faster than actually performing the computation.

The key bottleneck in SNARKs is the complexity of *proving* correctness of the computation. The prover's work scales with the complexity of the computation being proved, and the overhead can be quite large. State-of-the-art provers for general-purpose computations (aka zkVMs) are four to six orders of magnitude slower than native execution [Tha25]. While there are other important resources such as proof-size and verification time, these can usually be solved using recursive proof composition.[1] Thus, the prover time is the key remaining limiting factor for most applications.

This motivates the central question of this paper:

What is the smallest concrete overhead that proving can have over native execution?

Continuing a line of theoretical [RR24, RR25, HR22, ARR25] and more recently also applied [DP25, DP26, Sou24, Irr25] work, we focus on proving correctness of computations expressed as *Boolean circuits* (i.e., with bit-wise AND and XOR gates). This is in contrast to the vast majority of SNARK research, which focuses on proving correctness of arithmetic computation over large finite fields.

Boolean circuits are a natural target for two reasons. First, they serve as a good proxy for the actual cost of general computation. Second, they are perfectly suited to expressing the standard cryptographic primitives that dominate key applications of SNARKs. More generally, the same thing that makes a computation efficient natively — fewer bit operations — is what makes it cheap to prove in our system. Application designers optimize for efficiency anyway, and that optimization carries straight through to the prover.

Flock: Fast proving for batch computations. We introduce Flock, a proof-system designed and implemented to prove *batch* computations: that is, workloads consisting of many independent invocations of the same Boolean circuit, tied together by (simple) relations on their inputs and outputs. Concretely, fix a Boolean circuit F and an arbitrary auxiliary input/output (IO) circuit G; Flock proves statements of the form

$$F(\mathbf{x}_i, \mathbf{y}_i) = 0 \quad \forall\, i \in [K], \text{ and}$$
$$G\big((\mathbf{x}_i)_{i\in[K]}, (\mathbf{y}_i)_{i\in[K]}\big) = 0.$$

The idea is for the repeated relation F to capture the bulk of the work; the "glue circuit G" is in principle unrestricted but should be simple. For example, the F circuits can be hash evaluations and G checks that the inputs and outputs correspond to a hash-chain or a set of Merkle paths.

The batch setting is motivated both theoretically and practically. Thaler [Tha13] demonstrated the efficiency of batch proving using a variant of the GKR [GKR15] protocol. Proving a batch is significantly simpler than proving arbitrary circuits, as it obviates the need for expensive permutation or memory checking arguments [Set20, CBBZ23, CHM+20] and preprocessing. The verifier can be linear in the evaluation of one computation but sublinear in the entire batch size. For this reason,

[1] Recursive proof composition [Val08, CT10], combines a fast but *largish* SNARK system with one that has short proofs but a more expensive prover. This composition delivers the best-of-both-worlds (i.e., fast provers *and* short proofs) and is common in practice, see, e.g., [KST22, Zca22, Pol22].

batch computation has been well studied in theory in a line of work which showed that for sufficiently uniform computations one can in principle prove "as fast as one computes" [RR25, HR22].

Batch computation is also of immense practical interest: Almost every deployed SNARK workload is dominated by batched copies of a single small computation: hash chains in verifiable delay functions, Merkle openings, aggregation of hash-based signatures, and the inner layer of recursive proving are all collections of many invocations of the same primitive (with some small logic on top). Even general machine computation contains repeated steps of the same primitive (a single instruction execution). Most recently, the Ethereum Foundation specified Lean VM [lea25], with the specific goal of supporting their post-quantum transition through SNARKs. Lean VM is a read-only memory virtual machine with native support for hashing, and workloads such as signature aggregation are dominated by the hashing instructions.

In our implementation, we focus on the batch computation of functions F corresponding to hash function evaluations and a glue circuit G that supports basic linear IO relations. This suffices for hash-chains, Merkle trees and independent hashing. These circuits are sufficiently rich to capture many applications, e.g. VDFs, and demonstrate the power of our proof system. Our protocol can be easily extended by implementing more complex G, or can be tied together with a compatible multilinear proof system over binary fields (e.g., simple adaptations of PIOPs such as [Set20, CBBZ23])

Why hashes, and why standard hashes. We benchmark Flock on the most ubiquitous batch workload, cryptographic hashing, and we deliberately target *standard*, hardware-friendly hash functions—Keccak, SHA-256, and BLAKE3—rather than the "SNARK-friendly" primitives (e.g. Poseidon [GKR+21, GKS23, GKK+26], Rescue [AABS+20], MiMC [AGR+16]) that much of the literature co-designs with its proof systems. Standard hashes are what most real-world applications already use, from Merkle trees and verifiable delay functions to the hash-based signatures proposed for post-quantum Bitcoin and Ethereum, and they are the most heavily scrutinized primitives in deployment; SNARK-friendly designs are far less battle-tested, and their algebraic structure has repeatedly enabled attacks [ABM23, Ste24, GKR25, BBB+25, ZSVD25, MG26]. More fundamentally, a SNARK-friendly hash is cheap to prove only inside a proof system built over the single field it was designed for: it *enshrines* that field, so only SNARKs over the same field can exploit it. This severely limits composability and upgradability — swapping the proof system, or composing two systems over different fields, forfeits the speedup — and it forces the application designer into a false choice: optimize the hash for the prover and pay for it in native execution, or optimize for native speed and pay for it in the prover.

Targeting a standard hash removes this choice. The hash is fixed by the application and runs identically whether or not a proof is ever produced, so the only remaining question is how cheaply it can be proved. By showing that Flock's prover throughput essentially tracks the native speed of standard hashes — while remaining best-in-class on what are arguably the three most popular hash functions — we let designers pick a hash on its own merits and obtain efficient proving regardless. This also makes "prover time vs. native time" a fair comparison: standard hashes cost only a handful of cycles per byte on modern hardware, so any inefficiency in the proof system is immediately visible against a single-digit-nanoseconds-per-byte baseline. SNARK-friendly hashes, by contrast, flatter the prover by making the native baseline itself slow.

1.1 Our results

We design and implement Flock, a hash-based post-quantum SNARK for batch Boolean R1CS computations. Our headline number, measured on a single core of an Apple M4 Max:

$$\frac{\text{prover time per hash}}{\text{native time per hash}} \approx 250,$$

where the native time measures independent *computations* of the underlying permutation/compression function.[2]

On the same computer Flock proves 82k BLAKE3 compressions, 42k SHA-256 compressions, and 30k Keccak permutations per second on a single core (this includes witness generation which benchmarks sometimes exclude). The implementation parallelizes well: on ten cores throughput exceeds 660k BLAKE3 compressions per second.

On proving SHA-256, Flock is roughly 9× faster than Binius64 [Irr25], the prior state-of-the-art, and over 1000× faster than Spartan [Set20]. It is about 1.8× faster than Hashcaster [Sou24] in proving the Keccak-f permutation, and 14× faster than Binius64 and Plonky3 [Plo24] in proving BLAKE3. See Section 5 for the full breakdown. Flock's proofs are less than 450 KB for computations with 2^{30} AND gates and take only a few milliseconds to verify.

From hashes to signatures. These throughputs translate into application-level rates. Post-quantum hash-based signatures [Lam79, Mer89, BDH11] are dominated by hashing: a single signature in the multi-signature scheme proposed for Ethereum's quantum transition costs about 160 hash invocations [DKKW25], and aggregating many such signatures is exactly the batched-hashing workload Flock targets. Were Ethereum to instantiate these signatures with BLAKE3, Flock's ten-core throughput of 660k compressions per second could prove hashing corresponding to 4,000 transactions per second. Vitalik Buterin recently estimated [But24] that proving 200k hashes per second would suffice for a post-quantum transition. Flock, therefore, puts BLAKE3 and other standard hash functions as feasible targets for Ethereum's roadmap. For Bitcoin, which is extremly conservative, and avoids novel cryptographic assumptions, a post-quantum transition will likely involve SHA256-based signatures. Flock can prove hashing corresponding to 5,000 signatures, the current upper limit of a Bitcoin block, in less than 2.5 seconds on consumer hardware. We stress that these figures count only the hashing inside signature verification — not transaction execution, state access, or any other logic.

Security. Flock targets 100 bits of soundness (for details see Section 5). We rely only on proximity-gap results in the proven regime [BCH+25]; the remaining soundness assumption is the security of SHA-256, used internally for both Merkle commitments and Fiat-Shamir.

Limitations. Flock is a research prototype and we do not recommend using it in production systems yet. It supports only batched Boolean-circuit computations expressed as R1CS. We give specific instantiations (hash-chaining and Merkle trees) for the IO circuit G, but do not support general circuits yet (in particular, supporting signature aggregation would require additional work). The implementation was heavily AI-assisted and was targeted at demonstrating optimal performance

[2] ARM has special instructions for acceleration of SHA-256 and SHA3 compressions. Since we are interested in an apples-to-apples comparison, we do not enable these instructions when computing the native execution time.

on a specific system (Apple ARM processors), rather than generality. We expect that similar optimizations carry over to other architectures.

Lastly, we remark that the system currently only offers succinctness and not zero-knowledge but we expect that zero-knowledge could be added at a relatively low cost (e.g., via techniques from the recent works [CFW26, DHRR26]).

1.2 Overview of techniques

The starting point for Flock is a Spartan-like [Set20] (cf. [Tha22, Chapter 8]) PIOP for batch-R1CS, over the binary field $\mathbb{F}_2$. Compared to Plonkish [GWC19] arithmetization, R1CS has a significantly smaller witness as it does not commit to the output of addition gates.

Our concrete gains come from multiple ingredients, including multiple known optimizations combined with several novel ones that we introduce. We define a simple yet powerful zerocheck formulation for batch-R1CS that can be combined with a single small lincheck across the A, B, C matrices. The lincheck amortizes the verifier's matrix evaluation cost over the batch dimension, and is extremely cheap. It takes less than 10% of our total prover time. The batch R1CS optimization is based on [BCG+19, HR22] and is quite simple, but already buys us significant mileage — in particular making the zerocheck protocol the key remaining bottleneck, and the focus of most of our optimization efforts.

Focusing on zerocheck (which, in our context, is simply checking that the pointwise product between bit-vectors a and b is equal to c) we first and foremost utilize many known optimizations from the literature. These include the univariate skip [Gru24], partially deterministic sumcheck challenges [DT24], deferred modular reduction and evaluation at infinity [DBE+26].

We push these optimizations further. First, building on Dao and Thaler's [DT24] partially deterministic zerocheck challenges, we identify a new set of challenge points that can process $\log(|\mathbb{F}|)$ of the individual zerocheck bit claims using roughly 2 bit-shifts and XORs per input, and without relying on tower-fields, which are less efficient than direct extensions of $\mathbb{F}_2$.

Second, similarly to Binius64 [Irr25], we utilize a degree 64 univariate skip [Gru24]. We introduce a new cache-friendly lookup-based low-degree-extension kernel that is the main workhorse of the zerocheck protocol. Additionally, we show how the zerocheck claim over the vector c can be *elided*, removing roughly a third of the invocations of our most expensive procedure, namely, the batch univariate low degree extension for the univariate skip. This step requires us to evaluate our witness at two independent points. We further optimize the batch evaluation and the ring-switching techniques for this setting. See Section 4 for details.

Our PCS is based on the Binius [DP26] ring-switching technique combined with Ligerito [NA25] as the underlying dense PCS (the system is modular and one could equally instantiate the dense PCS with a different Boolean commitment scheme such as Blaze [BCF+25] or Bolt [GNR26b] or use Blaze as a Boolean-PCS and avoid the ring-switching altogether). We provide a self-contained overview of the ring-switching technique in Appendix B, replacing one of the steps with a more modular, and arguably simpler, approach. We also give an overview of Ligerito in Appendix C, and extend its analysis to the list-decoding regime (following [ACFY25]).

1.3 Related works

Binius. Flock is most closely related to the Binius64 proof-system [Irr25] (which builds on [DP25, DP26]). Like Binius, we work over binary fields and we build our $\mathbb{F}_2$ polynomial commitment

using their packing and ring-switching technique [DP26], which reduces building a commitment to a small-field ($\mathbb{F}_2$) witness to building one over the large field $\mathbb{F}_{2^{128}}$. Unlike the Binius academic papers, which use tower fields, but similarly to the Binius64 implementation, we instantiate $\mathbb{F}_{2^8}$ as the AES field and $\mathbb{F}_{2^{128}}$ as GHASH, as they have much faster implementations. Some high-level differences between our approach and that of Binius64 are that we commit only to the inputs, outputs and multiplication gates, whereas Binius64 commits to all wire values. This leads to our trace being significantly smaller. Also, by focusing on the batch setting our lincheck is drastically cheaper than the more general one that they support. Lastly, our new techniques significantly improve zero-check, a key bottleneck for both systems but especially for Flock.

Hashcaster. Hashcaster [Sou24] is a GKR-based system using binary fields that specifically targets batch proving of Keccak permutations. Extending it to SHA-256 or BLAKE3 seems difficult as these hash functions perform u32-addition operations that are hard to handle via GKR.[3]

For proving Keccak, Hashcaster's performance is closest to ours ($\sim$1.8$\times$ slower on a single core, 4$\times$ slower multi-threaded). Additionally, and in contrast to Flock, its proof size and verifier time grow linearly in the depth of the computation. The main advantage of the GKR approach is that the commitment is to a *much* smaller witness. Interestingly, some of our key optimizations do not immediately translate to the GKR setting. In particular, our deterministic challenges rely on performing a single zerocheck-to-sumcheck conversion: because we move from a small $\mathbb{F}_2$ witness to an $\mathbb{F}_{2^{128}}$ sumcheck in one shot, the challenges for that step can be made partially deterministic. GKR-style systems such as Hashcaster instead run many rounds of sumcheck, with the challenges in each round derived from the previous one, so this optimization does not seem to carry over.

Prime field SNARKs. A large body of work builds SNARKs over larger prime fields (and extensions thereof). Plonky3 [Plo24], for example, combines a Plonk-style PIOP with FRI as its polynomial commitment. Arithmetizing Boolean computation — such as the bit-level operations underlying standard hash functions — is costly over these fields because of their high *embedding overhead*: the gap between the number of bits a field element occupies and the number of bits of useful information it carries. While several of our techniques transfer to other fields, we focus on binary fields precisely because of their minimal embedding overhead. Overall, Flock is around 10-40x faster than Plonky3 [Plo24], a popular proof library which operates over a 31-bit prime field.

Group based SNARKs. Finally, there is a large family of group-based SNARKs, including Groth16 [Gro16] and Spartan [Set20]. These systems are not post-quantum secure and require arithmetizing the computation over an exponentially large prime field. While their proofs are very small, we show that their prover performance is simply not competitive with Flock. Spartan is 300/1000$\times$ (single/multi-threaded) slower than Flock.

1.4 Organization

Section 2 contains preliminaries. Section 3 presents a baseline Spartan-like proof-system for batch-R1CS over $\mathbb{F}_2$ that serves as our starting point. Section 4 is the technical core of the paper, developing the optimizations that make Flock fast. Section 5 reports our experimental evaluation

[3] While logarithmic-depth linear-size circuits for addition are known [BK82], using them would introduces an impractical depth blowup.

and comparison against prior systems. Appendix A gives the full protocol specification, Appendix B a self-contained overview of the ring-switching technique. Appendix C gives an overview of the Ligerito polynomial commitment scheme. Lastly, Appendix D proves the Cauchy-shift identity.

2 Preliminaries

Notation. Throughout, $\mathbb{F}$ denotes a finite field. For a positive integer n, we write $[n] := \{1, \ldots, n\}$ and identify the Boolean hypercube $\{0,1\}^n$ with the set of binary vectors of length n. Vectors are written in bold (e.g. $\mathbf{x}$) and indexed as $\mathbf{x} = (x_1, \ldots, x_n)$. We use log for the base-2 logarithm.

2.1 Multilinear extensions

Every function $f : \{0,1\}^m \to \mathbb{F}$ has a unique multilinear extension $\hat{f} : \mathbb{F}^m \to \mathbb{F}$ that agrees with f on $\{0,1\}^m$, given explicitly by

$$\hat{f}(\mathbf{x}) \;=\; \sum_{\mathbf{b} \in \{0,1\}^m} f(\mathbf{b}) \cdot \mathrm{eq}(\mathbf{b}, \mathbf{x}),$$

where the equality polynomial

$$\mathrm{eq}(\mathbf{b}, \mathbf{x}) \;:=\; \prod_{i=1}^{m} \big(b_i\, x_i + (1 - b_i)(1 - x_i)\big)$$

is the unique multilinear polynomial that takes value 1 on $\mathbf{b}$ and 0 elsewhere on $\{0,1\}^m$. The following equivalent representation will also be useful:

Fact 1. *For $\mathbf{b} \in \{0,1\}^m$ and $\mathbf{x} \in \mathbb{F}^m$ it holds that:*

$$\mathrm{eq}(\mathbf{b}, \mathbf{x}) = \prod_{i=1}^{m} \left(x_i^{b_i} \cdot (1 - x_i)^{1 - b_i} \right).$$

2.2 Rank-1 constraint systems (R1CS)

A *rank-1 constraint system* (R1CS) over $\mathbb{F}$ is specified by three matrices $A, B, C \in \mathbb{F}^{N \times M}$. A vector $\mathbf{z} \in \mathbb{F}^M$ *satisfies* the system if

$$(A\mathbf{z}) \circ (B\mathbf{z}) \;=\; C\mathbf{z},$$

where $\circ$ denotes the entrywise (Hadamard) product. Splitting $\mathbf{z} = (\mathbf{x}, \mathbf{w})$ into a public statement $\mathbf{x} \in \mathbb{F}^\ell$ and a private witness $\mathbf{w} \in \mathbb{F}^{M-\ell}$ yields the standard R1CS satisfiability relation: $\big((A, B, C), \mathbf{x}\big)$ is in the relation iff there exists $\mathbf{w}$ such that $\mathbf{z} = (\mathbf{x}, \mathbf{w})$ satisfies the above.

Throughout this paper we will be focusing on square R1CS matrices with $N = M = 2^m$ over the binary field $\mathbb{F}_2$.

2.3 Sumcheck

The sumcheck protocol [LFKN92] is an interactive reduction for claims of the form

$$S \;=\; \sum_{\mathbf{b} \in \{0,1\}^m} g(\mathbf{b}),$$

where $g : \mathbb{F}^m \to \mathbb{F}$ is a polynomial of individual degree at most d in each variable. The protocol runs for m rounds; in each round the prover sends a univariate polynomial of degree $\leq d$ and the verifier replies with a uniform challenge $r_i \in \mathbb{F}$. At the end the verifier is left with a single evaluation claim $g(\mathbf{r}) = v$ at a point $\mathbf{r} = (r_1, \ldots, r_m) \in \mathbb{F}^m$. The protocol has perfect completeness and soundness error at most $md/|\mathbb{F}|$.

Remark 2 (Batch sumcheck). Multiple sumcheck claims $\{v_t = \sum_{\mathbf{b}\in\{0,1\}^m} g_t(\mathbf{b})\}_{t=1}^T$ on the same hypercube can be folded into one: the verifier samples random $\alpha_1, \ldots, \alpha_T \in \mathbb{F}$, and the parties run sumcheck on $\sum_t \alpha_t v_t = \sum_{\mathbf{b}} \sum_t \alpha_t g_t(\mathbf{b})$. The combined sumcheck inherits the degree and round count; the soundness error increases additively by $1/|\mathbb{F}|$ due to the randomness of the α-combination.

2.3.1 Zerocheck

To check that a polynomial $f : \mathbb{F}^m \to \mathbb{F}$ vanishes on the Boolean hypercube, the verifier samples $\mathbf{r} \in \mathbb{F}^m$ and runs sumcheck on

$$\sum_{\mathbf{b}\in\{0,1\}^m} \mathrm{eq}(\mathbf{r}, \mathbf{b}) \cdot f(\mathbf{b}) \stackrel{?}{=} 0.$$

The summed expression equals the multilinear extension (in $\mathbf{r}$) of $f|_{\{0,1\}^m}$, so it vanishes identically iff f vanishes on $\{0,1\}^m$; a non-vanishing f is caught except with probability $m/|\mathbb{F}|$ over the choice of $\mathbf{r}$, plus the soundness error of the underlying sumcheck. We refer to this combined protocol as the *zerocheck* on f.

2.4 Multilinear polynomial commitments

A multilinear *polynomial commitment scheme* (PCS) over $\mathbb{F}$ lets a prover commit to a multilinear polynomial $\hat{f} : \mathbb{F}^m \to \mathbb{F}$, producing a short digest cm. Later, the prover can generate a short proof π certifying that $\hat{f}(\mathbf{r}) = v$ for a point $\mathbf{r} \in \mathbb{F}^m$ and value $v \in \mathbb{F}$.

Interactive oracle PCS. We describe the information-theoretic analogue of a PCS, known as an *interactive oracle PCS* (IOPCS) [DP26, BFRW25]. A multilinear IOPCS consists of two phases:

- *Commitment phase:* The prover, given a function $f\colon \{0,1\}^m \to \mathbb{F}$, interacts over multiple rounds with the verifier; the resulting transcript functions as a commitment to f.
- *Opening phase:* Given a point $\mathbf{r} \in \mathbb{F}^m$ and value $v \in \mathbb{F}$ claimed to be $\hat{f}(\mathbf{r})$, the prover interacts over multiple rounds with the verifier; the resulting transcript functions as a proof. At the end, the verifier either accepts or rejects.

Crucially, we require that the verifier only reads a few bits from the commitment and proof strings, which may be long (e.g., as long as the description of f). We additionally require the following properties:

- *Completeness:* Assuming the prover and verifier behave honestly, the verifier always accepts whenever $\hat{f}(\mathbf{r}) = v$.
- *Binding:* Every commitment transcript cm is associated with a subset S_{cm} of multilinear polynomials $\mathbb{F}^m \to \mathbb{F}$ such that the following holds. For any (possibly dishonest) prover, S_{cm} contains at most one element, with all but negligible probability (over the prover and verifier's randomness).

- *Soundness:* For any (possibly dishonest) prover, if there does not exist a $\hat{f} \in S_{\mathsf{cm}}$ such that $\hat{f}_{\mathsf{cm}}(\mathbf{r}) = v$, then the verifier rejects with all but negligible probability.

We remark that the opening phase (and soundness property) can be generalized to support multiple evaluation claims.

Throughout this work we refer to IOPCSs simply as PCSs.

Boolean PCS. In this work we rely on *Boolean* multilinear PCSs, which support committing to multilinear extensions of Boolean-valued functions $f : \{0,1\}^m \to \{0,1\}$. The verifier is guaranteed that any commitment is consistent with a function of this form.

Ring-switching [DP26] is a technique that enables constructing Boolean multilinear PCSs from (standard) multilinear PCSs. In more detail, suppose that $\mathbb{F} \supseteq \mathbb{F}_2$ is a large binary field with extension degree 2^k. Then, ring-switching transforms any "dense" $(m-k)$-variable multilinear PCS over $\mathbb{F}$ into a Boolean m-variable multilinear PCS over $\mathbb{F}$. In our constructions, we instantiate the dense PCS with Ligerito [NA25].

3 Baseline R1CS Protocol

We start by describing a baseline succinct argument for R1CS over the binary field $\mathbb{F}_2$, deferring all optimizations to Section 4. The protocol is a variant of Spartan [Set20] for the binary field setting (see also [Tha22, Chapter 8]). The construction is described as an interactive oracle proof [BCS16, RRR21], which can be compiled into a SNARK via standard transformations [Kil92, Mic00, BCS16].

Setup. Let $m \in \mathbb{N}$ and fix an R1CS instance (A, B, C) in which the matrices are square matrices over $\mathbb{F}_2$ of dimension $2^m \times 2^m$. We view them as functions $A, B, C : \{0,1\}^m \times \{0,1\}^m \to \mathbb{F}_2$. Consider a satisfying assignment $\mathbf{z}$ of the form $\mathbf{z} = (\mathbf{x}, \mathbf{w}) \in \mathbb{F}_2^{2^m}$, where $\mathbf{x}$ is the public input (known to the verifier) and $\mathbf{w}$ is the private input (i.e., the witness). Let $\mathbf{a} := A\mathbf{z}$, $\mathbf{b} := B\mathbf{z}$, $\mathbf{c} := C\mathbf{z}$. We view $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{z}$ as functions (from $\{0,1\}^m$ to $\mathbb{F}_2$) and write $\hat{a}, \hat{b}, \hat{c}, \hat{z} : \mathbb{F}^m \to \mathbb{F}$ for their multilinear extensions over a large extension field $\mathbb{F} \supseteq \mathbb{F}_2$. Similarly, for each matrix $M \in \{A, B, C\}$ we denote its multilinear extension by $\hat{M} : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}$. The R1CS satisfiability condition is then equivalent to

$$\hat{a}(\mathbf{i}) \cdot \hat{b}(\mathbf{i}) - \hat{c}(\mathbf{i}) \;=\; 0 \qquad \forall\, \mathbf{i} \in \{0,1\}^m.$$

Protocol. The baseline protocol proceeds as follows:

1. **Commit to the trace.** The prover commits to $\hat{z}$ using a Boolean multilinear PCS (see Section 2.4), producing a commitment transcript cm.

2. **Zerocheck on $\hat{a} \cdot \hat{b} - \hat{c}$.** The verifier samples a random challenge $\mathbf{r} \in \mathbb{F}^m$, and the parties run sumcheck on
$$\sum_{\mathbf{i} \in \{0,1\}^m} \mathrm{eq}(\mathbf{r}, \mathbf{i}) \cdot \big(\hat{a}(\mathbf{i}) \cdot \hat{b}(\mathbf{i}) - \hat{c}(\mathbf{i})\big) \;\stackrel{?}{=}\; 0.$$
After m rounds the verifier holds a random point $\mathbf{r}_y \in \mathbb{F}^m$ and claimed values
$$v_a = \hat{a}(\mathbf{r}_y), \qquad v_b = \hat{b}(\mathbf{r}_y), \qquad v_c = \hat{c}(\mathbf{r}_y),$$
and checks locally that $\mathrm{eq}(\mathbf{r}, \mathbf{r}_y) \cdot (v_a \cdot v_b - v_c)$ matches the final sumcheck value.

3. **Lincheck: reduce the $\hat{a}, \hat{b}, \hat{c}$ claims to a single claim on $\hat{z}$.** We describe the reduction for $\hat{a}$ first. Since $\mathbf{a} = A\mathbf{z}$, the MLE evaluation of $\hat{a}$ at $\mathbf{r}_y$ can be expressed as:

$$\hat{a}(\mathbf{r}_y) \;=\; \sum_{\mathbf{j} \in \{0,1\}^m} \hat{A}(\mathbf{r}_y, \mathbf{j}) \cdot \hat{z}(\mathbf{j}).$$

   The parties can therefore check the claim $v_a = \hat{a}(\mathbf{r}_y)$ via a sumcheck, resulting in a verifier evaluation of $\hat{A}(\mathbf{r}_y, \mathbf{r}_x) \cdot \hat{z}(\mathbf{r}_x)$ at a random $\mathbf{r}_x \in \mathbb{F}^m$. For the moment let us assume that the verifier computes $\hat{A}(\mathbf{r}_y, \mathbf{r}_x)$ on its own and so we are left only with a claim on $\hat{z}(\mathbf{r}_x)$.[4] We refer to such a reduction—from an MLE claim on $\hat{a}$ to an MLE claim on $\hat{z}$ via the matrix MLE $\hat{A}$—as a *lincheck* on A.

   The same applies with B, C in place of A. Batching the three linchecks via Remark 2, the three lincheck claims reduce to a single claim $v_z = \hat{z}(\mathbf{r}_x)$ at a random point $\mathbf{r}_x \in \mathbb{F}^m$ as well as evaluation claims on $\hat{A}, \hat{B}, \hat{C}$.

4. **Consistency with public input.**[5] For simplicity, assume that the public input length is 2^k and $\mathbf{z}$ is arranged so that $\mathbf{z}(0^{m-k}, \cdot)$ should be equal to $\mathbf{x}$. The verifier tests consistency via an MLE claim $v' = \hat{z}(\mathbf{r}')$, where $\mathbf{r}' = (0^{m-k}, \mathbf{r})$ for randomly sampled $\mathbf{r} \in \mathbb{F}^k$ and $v' = \hat{x}(\mathbf{r})$ is computed directly by the verifier.

5. **Open the trace.** The prover and verifier run the PCS opening phase on cm with evaluations $(\mathbf{r}_x, v_z)$ and $(\mathbf{r}', v')$. The verifier accepts if and only if PCS verifier accepts and every check above passes.

Completeness. Let $\mathbf{x}$ be a public input and suppose the (honest) prover holds a witness $\mathbf{w}$ with $\mathbf{z} = (\mathbf{x}, \mathbf{w})$ being a satisfying assignment. Then $\mathrm{eq}(\mathbf{r}, \mathbf{i}) \cdot (\hat{a}(\mathbf{i}) \cdot \hat{b}(\mathbf{i}) - \hat{c}(\mathbf{i}))$ vanishes on $\{0,1\}^m$, and, by completeness of zerocheck and lincheck, it holds that $v_z = \hat{z}(\mathbf{r}_x)$. Since $\mathbf{z}(0^{m-k}, \cdot) \equiv \mathbf{x}$, it also holds that $v' = \hat{z}(\mathbf{r}')$. Finally, by completeness of the PCS, the verifier always accepts.

Soundness. Let $\mathbf{x}$ be a public input and suppose there does not exist a witness $\mathbf{w}$ with $(\mathbf{x}, \mathbf{w})$ being a satisfying assignment. With all but the PCS's binding error probability, the commitment transcript cm is associated with a subset S_{cm} containing at most one multilinear polynomial. If S_{cm} is empty, then the verifier rejects with all but the PCS's soundness error probability. Thus, we focus on the scenario where S_{cm} contains only the multilinear extension of some function $\mathbf{z}_{\mathsf{cm}} \colon \{0,1\}^m \to \mathbb{F}$. There are two cases:

- If $\mathbf{z}_{\mathsf{cm}}$ is a satisfying assignment, then it must be inconsistent with $\mathbf{x}$, i.e., $\hat{z}_{\mathsf{cm}}(\cdot, 0^{m-k}) \not\equiv \mathbf{x}$. By Schwartz-Zippel, with all but $k/|\mathbb{F}|$ probability, it holds that $\hat{z}_{\mathsf{cm}}(\mathbf{r}') \neq v'$. Assuming this, the verifier rejects with all but the PCS's soundness error probability.

- If $\mathbf{z}_{\mathsf{cm}}$ is not a satisfying assignment, then there exists $\mathbf{i} \in \{0,1\}^m$ such that $\hat{a}_{\mathsf{cm}}(\mathbf{i}) \cdot \hat{b}_{\mathsf{cm}}(\mathbf{i}) - \hat{c}_{\mathsf{cm}}(\mathbf{i}) \neq 0$, where $\hat{a}_{\mathsf{cm}}, \hat{b}_{\mathsf{cm}}, \hat{c}_{\mathsf{cm}}$ are the multilinear extensions of $A\mathbf{z}_{\mathsf{cm}}, B\mathbf{z}_{\mathsf{cm}}, C\mathbf{z}_{\mathsf{cm}}$. The

[4] Since A is a fixed matrix, this computation can be done independently by the verifier. However, for a general matrix A it can be costly. In the actual protocol we will leverage the fact that the matrix is structured to reduce the verifier's cost (see Section 4.1).

[5] This step will be replaced with input/output constraints (Section 4.6) when we move to Batch R1CS in the next section.

zerocheck and lincheck protocols are ran with respect to $\mathbf{z}_{\mathsf{cm}}$; with all probability $O(m/|\mathbb{F}|)$, it holds that $\hat{z}_{\mathsf{cm}}(\mathbf{r}_x) \neq v_z$ (or one of the verifier's other checks already fails). Assuming this, the verifier rejects with all but the PCS's soundness error probability.

We conclude that the overall soundness error is at most $O(m/|\mathbb{F}|)$, plus the PCS's binding and soundness errors.

Remark 3. A straightforward implementation of the baseline protocol over $\mathbb{F}_2$ is quite inefficient. In particular, when used over $\mathbb{F}_2$, the zerocheck protocol introduces an exorbitant $O(2^m \cdot \log(|\mathbb{F}|))$ prover cost. Indeed, this is why the Spartan protocol is designed for R1CS over large fields. In the next section we discuss how to efficiently prove R1CS over $\mathbb{F}_2$.

4 Flock

We describe the different optimizations that we utilize in Flock to make the baseline protocol extremely efficient and practical. These optimizations build on, and improve, a long line of prior research in the field. For completeness, we describe the full protocol in Appendix A.

Since some of the optimizations are related to each other, we do not list them in order of significance but rather based on a topological sort of the dependency graph.

4.1 Batch R1CS

The motivating workloads for Flock are *batch computations*: K independent invocations of the same primitive — e.g., multiple Keccak permutations or SHA2 compressions. Each individual computation can be described by the same R1CS instance over $\mathbb{F}_2$. Rather than proving the K instances separately, we stack them into a single R1CS with dimension that is K times larger. Utilizing a technique from [BCG+19] (further extended in [HR22]), such a batch-R1CS can be handled *much* more efficiently than a generic instance.

Concretely, let the per-invocation (*base*) matrices be $A_0, B_0, C_0 : \{0,1\}^{m_0} \times \{0,1\}^{m_0} \to \mathbb{F}_2$ and let $K = 2^k$ denote the number of instances. These matrices correspond to verification of a single copy. The batched witness is

$$\mathbf{z} : \{0,1\}^k \times \{0,1\}^{m_0} \to \mathbb{F}_2$$

where we denote by $\mathbf{z}_i = \mathbf{z}(i, \cdot)$. The batched matrices are the block-diagonal matrices

$$A \;=\; I_K \otimes A_0, \quad B \;=\; I_K \otimes B_0, \quad C \;=\; I_K \otimes C_0,$$

where I_K denotes the $K \times K$ identity matrix and $\otimes$ is the tensor product. To see this observe that $(I_K \otimes A_0) \cdot z$ simply applies the matrix A_0 to each component of z separately.

These are the matrices fed to the baseline protocol of Section 3, with the role of m played by $m := k + m_0$. Satisfaction of $(A\mathbf{z}) \circ (B\mathbf{z}) = C\mathbf{z}$ is equivalent to simultaneous satisfaction of all K underlying instances. We refer to $A\mathbf{z}$, $B\mathbf{z}$, and $C\mathbf{z}$ as the *A-side*, *B-side*, and *C-side* of the constraint, respectively.

Lincheck via block structure. Decompose each hypercube index $\mathbf{i} \in \{0,1\}^m$ as $\mathbf{i} = (\mathbf{i}_{\text{out}}, \mathbf{i}_{\text{in}})$ with $\mathbf{i}_{\text{out}} \in \{0,1\}^k$ the batch index and $\mathbf{i}_{\text{in}} \in \{0,1\}^{m_0}$ the within-instance index. The MLE of a block-diagonal matrix factors as

$$\hat{A}\big((\mathbf{r}_{\text{out}}, \mathbf{r}_{\text{in}}), (\mathbf{j}_{\text{out}}, \mathbf{j}_{\text{in}})\big) \;=\; \text{eq}(\mathbf{r}_{\text{out}}, \mathbf{j}_{\text{out}}) \cdot \hat{A}_0(\mathbf{r}_{\text{in}}, \mathbf{j}_{\text{in}}),$$

and similarly for $\hat{B}, \hat{C}$. Substituting into the lincheck-on-A identity from Section 3 and pulling the $\mathbf{j}_{\text{out}}$-only factor out of the inner sum,

$$\begin{aligned}\hat{a}(\mathbf{r}_y) &= \sum_{\mathbf{j}_{\text{out}}} \text{eq}(\mathbf{r}_{y,\text{out}}, \mathbf{j}_{\text{out}}) \cdot \sum_{\mathbf{j}_{\text{in}}} \hat{A}_0(\mathbf{r}_{y,\text{in}}, \mathbf{j}_{\text{in}}) \cdot \hat{z}(\mathbf{j}_{\text{out}}, \mathbf{j}_{\text{in}}) \\ &= \sum_{\mathbf{j}_{\text{in}}} \hat{A}_0(\mathbf{r}_{y,\text{in}}, \mathbf{j}_{\text{in}}) \cdot \hat{z}(\mathbf{r}_{y,\text{out}}, \mathbf{j}_{\text{in}}),\end{aligned}$$

a sumcheck over only m_0 variables (rather than $m = k + m_0$). Two key consequences:

- The verifier evaluates only the base MLEs $\hat{A}_0, \hat{B}_0, \hat{C}_0$ at a single point $(\mathbf{r}_{y,\text{in}}, \mathbf{r}_{x,\text{in}}) \in \mathbb{F}^{m_0} \times \mathbb{F}^{m_0}$; the outer k rounds contribute only a cheap extra eq-factor evaluation.

  In particular, the matrices are sufficiently small that the verifier can compute their multilinear extension at the desired point by itself.

- Even more significantly, the prover only needs to "fold" z into $\hat{z}(\mathbf{r}_{y,\text{out}}, \cdot)$ and the remaining work is independent of k; only the inner m_0 rounds operate on the small base matrices A_0, B_0, C_0 (rather than the K-times-larger matrices A, B, C).

The latter point is crucial and drastically reduces the lincheck cost in Flock. In particular, we observe that when using this batching technique the dominant part in the protocol by far becomes zerocheck, and so most of the optimizations described below attempt to extensively optimize that part of the protocol.

4.2 Univariate skip

Motivation. The zerocheck of Section 3 runs sumcheck on a summand $\text{eq}(\mathbf{r}, \mathbf{i}) \cdot (\hat{a}(\mathbf{i})\hat{b}(\mathbf{i}) - \hat{c}(\mathbf{i}))$ whose constraint values $\hat{a}, \hat{b}, \hat{c}$ are bits. Recall that after the initial round of sumcheck, the verifier samples a challenge $r_1 \in \mathbb{F}$ from a large extension field (e.g. $|\mathbb{F}| = 2^{128}$), and the residual claim is a sumcheck over $\{0,1\}^{m-1}$ on the partial evaluations $\hat{a}(r_1, \cdot), \hat{b}(r_1, \cdot), \hat{c}(r_1, \cdot)$. The hypercube halves, but each cell is now an element of $\mathbb{F}$ rather than a bit: a size-$n = 2^m$ *bit*-valued claim has been replaced by a size-$n/2$ $\mathbb{F}$-valued claim — a $\log(|\mathbb{F}|)/2$ blowup of the working state. This is a massively unaffordable cost.

The univariate skip. We utilize a popular technique due to Gruen [Gru24], called *the univariate skip.*[6]

Fix a parameter $k_{\text{s}} \in [m]$ (concretely $k_{\text{s}} = 6$ in our implementation) and view each of $a, b, c : \{0,1\}^m \to \mathbb{F}_2$ as a matrix of shape $2^{m-k_{\text{s}}} \times 2^{k_{\text{s}}}$, indexed by a row coordinate $\mathbf{x} \in \{0,1\}^{m-k_{\text{s}}}$ (the variables *kept* for the residual sumcheck) and a column coordinate $\mathbf{y} \in \{0,1\}^{k_{\text{s}}}$ (the variables *skipped* in the first round). Fix a k_{s}-dimensional $\mathbb{F}_2$-subspace $S \subset \mathbb{F}$ of size $2^{k_{\text{s}}}$ and identify the $2^{k_{\text{s}}}$ columns of the matrix a with the points of S. We view each row $\mathbf{x}$ of a as a table of evaluations of a unique univariate polynomial $\alpha_{\mathbf{x}} : \mathbb{F} \to \mathbb{F}_2$ of degree $< 2^{k_{\text{s}}}$ at the points of S. We refer to $\alpha_{\mathbf{x}}$ as the *low-degree extension* (LDE) of the row; analogously $\beta_{\mathbf{x}}, \gamma_{\mathbf{x}}$ are the LDEs of the corresponding rows of b and c.

[6] A related technique, also geared at reducing the cost of sumcheck over $\mathbb{F}_2$, was proposed earlier by Holmgren and Rothblum [HR22, Lemma 6.4]. Their technique is asymptotically faster than the univariate skip, but seems concretely worse as it requires sending an additional oracle.

Instead of running k_{s} vanilla sumcheck rounds — each consuming one large-field challenge — we collapse them into a single round. The verifier samples a zerocheck challenge $\mathbf{r} \in \mathbb{F}^{m-k_{\mathrm{s}}}$ over the kept variables only, and the prover computes and sends the univariate

$$P(\lambda) \;:=\; \sum_{\mathbf{x}\in\{0,1\}^{m-k_{\mathrm{s}}}} \mathrm{eq}(\mathbf{r},\mathbf{x}) \cdot \big(\alpha_{\mathbf{x}}(\lambda)\cdot\beta_{\mathbf{x}}(\lambda) - \gamma_{\mathbf{x}}(\lambda)\big)$$

of degree $\leq 2\cdot(2^{k_{\mathrm{s}}}-1)$, transmitted as evaluations at the points of a disjoint output coset $\Lambda := S + \delta \subset \mathbb{F}_{2^{k_{\mathrm{s}}}}$, for fixed $\delta \in \mathbb{F}\backslash S$. (It suffices to send evaluations on $S+\delta$ since P is known to be identically 0 on S.)

The verifier samples a single challenge $\lambda \in \mathbb{F}$ and the residual sumcheck of $m-k_{\mathrm{s}}$ rounds proceeds over $\mathbb{F}$ on the partially evaluated $\hat{a}(\lambda,\cdot), \hat{b}(\lambda,\cdot), \hat{c}(\lambda,\cdot)$.

The win is that the prover crosses the $\mathbb{F}_2 \to \mathbb{F}$ boundary exactly once. The k_{s} rounds of bit-level $\mathbb{F}_2$ structure are preserved end-to-end via the row-as-polynomial view, and the field upgrade is paid in a single round message rather than in k_{s} separate witness blowups.

LDE via lookup tables. In the univariate skip round, the prover's hot loop computes, for each row $\mathbf{x}$, the LDE evaluations $\alpha_{\mathbf{x}}|_\Lambda, \beta_{\mathbf{x}}|_\Lambda, \gamma_{\mathbf{x}}|_\Lambda$ on the output coset. Since we extend 0/1 valued functions, by taking $S \cup \Lambda$ to be contained in a subfield, we can ensure that all of the computed values are in the subfield.

Nevertheless, the question is how to compute this massive number of small LDEs. One option is to just use the NTT for every row; or, more precisely, an inverse NTT to recover the coefficients of each polynomial from its evaluations on S, and then a forward NTT to extend to Λ. This turns out to be highly inefficient in practice due to two reasons. First, the $O(|S|\log|S|)$ asymptotic cost of the NTT does not work well on the very small input sizes that we operate on. Second, the NTT does not exploit the fact that each row is bit-valued: it pays the full large-field arithmetic at every butterfly round.[7]

Following [Irr25], we can instead precompute the entire LDE map as a lookup table. Fix a chunk size $\tau \in \mathbb{N}$ in bits, dividing $|S|$ (e.g., a good number to bear in mind is $\tau = 8$, corresponding to a single byte). The composite map on a single column,

$$M \;:=\; \mathrm{NTT}_\Lambda \circ \mathrm{iNTT}_S \;:\; \mathbb{F}_{2^\tau}^{|S|} \;\to\; \mathbb{F}_{2^\tau}^{|\Lambda|},$$

is $\mathbb{F}_2$-linear, so packing the row's bits into $|S|/\tau$ chunks $a_0(\mathbf{x}), \ldots, a_{|S|/\tau-1}(\mathbf{x}) \in [0, 2^\tau)$ and tabulating M's action per chunk position gives

$$\alpha_{\mathbf{x}}|_\Lambda \;=\; \bigoplus_b M_b\big[\,a_b(\mathbf{x})\,\big],$$

where $M_b[v] \in \mathbb{F}_{2^\tau}^{|\Lambda|}$ encodes the action of M on a chunk of value v at position b. Each LDE call is then a fixed number of chunk lookups and XORs, sidestepping the NTT entirely.

New optimization: unified lookup table. Notice that the above approach does not leverage any structure of the matrix M. While we do not know how leverage the full NTT structure, we observe that M still has useful structure that we can exploit, which reduces the size of the lookup

[7] One could potentially get some small savings by leveraging the small values in the first few butterfly rounds.

table used by an $|S|/\tau$ factor. This is significant as we observe that the unified table can be resident in the L1 cache, thereby giving a significant speedup in the hot loop.

Specifically, M satisfies the following *translation-invariance* property: shifting the column index by τb, for any b, is equivalent to XOR-shifting the row index by τb. Formally, for every $i \in [|\Lambda|]$, $b \in \{0, 1, \ldots, |S|/\tau - 1\}$, and $j \in \{0, 1, \ldots, \tau - 1\}$,

$$M\big[i,\, (\tau b) + j\big] \;=\; M\big[\, i \oplus (\tau b),\, j\,\big],$$

where the column index arithmetic $(\tau b) + j$ is over the integers (in particular τb stands for integer multiplication) whereas the row-index $\oplus$ is a bitwise XOR. The proof of this identity is deferred to Appendix D.

Define the *single* base table T where for $v \in \{0,1\}^\tau$:

$$T[v] \;:=\; M\begin{pmatrix} v \\ \mathbf{0} \end{pmatrix} \;=\; \bigoplus_{j : v_j = 1} M[\cdot, j] \;\in\; \mathbb{F}_{2^\tau}^{|\Lambda|},$$

where $\begin{pmatrix} v \\ \mathbf{0} \end{pmatrix}$ has v in its first τ entries and zeros in the remaining $|S| - \tau$ entries. The translation-invariance of M implies $M_b[v][i] = T[v]\big[i \oplus (\tau b)\big]$ for any chunk position b. Letting $r_b := T\big[a_b(\mathbf{x})\big] \in \mathbb{F}_{2^\tau}^{|\Lambda|}$ be the b-th chunk's row of the base table, the unified-lookup form of the LDE is

$$\alpha_{\mathbf{x}}|_\Lambda[i] \;=\; \bigoplus_{b=0}^{|S|/\tau - 1} r_b\big[\, i \oplus (\tau b)\,\big],$$

i.e. a single base-table access per chunk: instead of keeping $|S|/\tau$ position-specific tables, we read from the same table T for every chunk and just XOR τb into the index i to account for the chunk position, which costs one additional XOR per output entry.

Concretely, taking $\tau = 8$ (so each chunk is a byte) and $|S| = |\Lambda| = 64$, the per-chunk-table version occupies $(|S|/\tau) \cdot 2^\tau \cdot |\Lambda| = 8 \cdot 256 \cdot 64 = 128\,\text{KB}$, which may spill out of the L1 cache[8]; the unified table is just $2^\tau \cdot |\Lambda| = 256 \cdot 64 = 16\,\text{KB}$.

The two approaches do roughly the same arithmetic work per LDE call: $|S|/\tau$ chunk lookups returning length-$|\Lambda|$ vectors, accumulated by $|S|/\tau$ vector XORs of length $|\Lambda|$ — i.e. $|S|/\tau \cdot |\Lambda|$ byte loads and $|S|/\tau \cdot |\Lambda|$ byte XORs in both. The unified lookup adds only a constant cost per chunk to compute the index offset τb (a single XOR amortized over the $|\Lambda|$ output entries), which is dwarfed by the main loop. The win is therefore not in operation count but in cache locality: one memory-hot base table reused across all chunk positions instead of $|S|/\tau$ separately materialized tables that compete for cache.

4.3 Friendly challenges

The zerocheck protocol (as described in Section 2.3.1) weighs the constraint polynomial by an eq-factor $\text{eq}(\mathbf{r}, \cdot)$ for a challenge vector $\mathbf{r} \in \mathbb{F}^m$ that the verifier samples from the large extension field $\mathbb{F}$. Each coordinate r_i shows up as a multiplier inside the prover's per-round arithmetic.

Motivated by the fact that multiplication by a generic $\mathbb{F}$ element is expensive, Dao and Thaler [DT24] observed that a small number of coordinates of $\mathbf{r}$ can be *pinned* to fixed elements of $\mathbb{F}$.

[8] On our benchmark M4 processors the L1 data cache is exactly 128KB.

Specifically, Dao and Thaler work over a tower field $\mathbb{F}$ and show that one can pin coordinates of $\mathbf{r}$ to the 7 tower-level generators $x_1, \ldots, x_7$, each of which lives in a (progressively larger) subfield of $\mathbb{F}$.

While this avoids some of the expensive multiplications, in general the tower representation seems significantly less efficient than other representations. For this reason Binius64 [Irr25] uses a non-tower representation (specifically they use the GHASH representation) and leverages the [DT24] trick differently. Specifically they pin 3 of the coordinates of $\mathbf{r}$ to fixed elements of the subfield $\mathbb{F}_{2^8} \subset \mathbb{F}$, multiplication by which is significantly cheaper than by a generic element of the large field $\mathbb{F}$. Specifically, viewing elements of $\mathbb{F}_{2^8}$ as univariates over $\mathbb{F}_2$, Binius chooses the three fixed elements $r_1 = x$, $r_2 = x^2$, and $r_3 = x^4$.

Soundness. We briefly recall the soundness argument from [DT24]. Replacing seven coordinates of $\mathbf{r}$ with hardcoded constants restricts the zerocheck verifier's challenge from a uniformly random point in $\mathbb{F}^m$ to a uniformly random point in an $(m-7)$-dimensional affine subspace. Soundness still goes through, provided the seven pinned values are $\mathbb{F}_2$-*linearly independent*. To see why, recall that the zerocheck reduces a constraint of the form "$p(\mathbf{x}) = 0$ for all $\mathbf{x} \in \{0,1\}^m$" to a single evaluation claim at $\mathbf{r}$. If however p is known to take 0/1 values (which is the case for us), then p vanishes on $\{0,1\}^m$ if and only if $\sum_{\mathbf{b} \in \{0,1\}^7} \alpha_{\mathbf{b}} \cdot p(\mathbf{b}, \mathbf{x}) = 0$ for every $\mathbf{x} \in \{0,1\}^{m-7}$, for every fixed sequence $\alpha_{\mathbf{b}}$ that are $\mathbb{F}_2$-linearly independent.

Thus, one can set $\mathbf{r} = (r_1, \ldots, r_7) \in \mathbb{F}^7$ to be any fixed vector as long as the corresponding $(\mathrm{eq}(\mathbf{r}, \mathbf{b}))_{\mathbf{b} \in \{0,1\}^7}$ are linearly independent.

Our choice: a geometric progression. We pick a different set of constant points. Let d denote the extension degree of $\mathbb{F}$ (i.e., $\mathbb{F}$ is isomorphic to $\mathbb{F}_{2^d}$). For $i \in [d]$, we set

$$r_i \;:=\; \frac{x^{2^{i-1}}}{1 + x^{2^{i-1}}},$$

where both the numerator and denominator refer to the polynomial representation of elements in $\mathbb{F}$ (and the division is over the field). The reason we do so is that $r_i/(1+r_i) = x^{2^{i-1}}$. Using this fact, we observe that for $\mathbf{r} = (r_1, \ldots, r_d)$ and any $\mathbf{b} = (b_1, \ldots, b_d) \in \{0,1\}^d$, applying Fact 1:

$$\begin{aligned} \mathrm{eq}(\mathbf{r}, \mathbf{b}) &= \prod_{i \in [d]} (1 + r_i)^{1-b_i} \cdot r_i^{b_i} \\ &= C \cdot \prod_{i \in [d]} \left(x^{2^{i-1}}\right)^{b_i} \;=\; C \cdot x^{\mathrm{int}(\mathbf{b})}, \end{aligned}$$

where $C = \prod_{i \in [d]} (1 + r_i)$ is a fixed constant, and $\mathrm{int}(\mathbf{b}) := \sum_{i \in [d]} 2^{i-1} b_i$ is the integer corresponding to $\mathbf{b}$. Thus, the 2^d weights form the geometric progression:

$$\{C,\; Cx,\; Cx^2, \ldots,\; Cx^{2^d - 1}\}.$$

The benefit is structural: multiplication by x^k in the field is just a k-bit shift followed by a modular reduction. To exploit this, the prover splits the sum into outer and inner parts:

$$\sum_{\mathbf{b} \in \{0,1\}^m} \mathrm{eq}(\mathbf{r}, \mathbf{b}) \cdot f(\mathbf{b}) \;=\; \sum_{\mathbf{b}_{\mathrm{out}} \in \{0,1\}^{m-d}} \mathrm{eq}(\mathbf{r}_{\mathrm{out}}, \mathbf{b}_{\mathrm{out}}) \cdot g(\mathbf{b}_{\mathrm{out}}),$$

where the inner factor is

$$g(\mathbf{b}_{\text{out}}) \;:=\; \sum_{\mathbf{b}_{\text{in}} \in \{0,1\}^d} \text{eq}(\mathbf{r}_{\text{in}}, \mathbf{b}_{\text{in}}) \cdot f(\mathbf{b}_{\text{out}}, \mathbf{b}_{\text{in}}).$$

Using our fixed choice of $\mathbf{r}_{\text{in}}$, in computing g, the 2^d scalar multiplications by the $\text{eq}(\mathbf{r}_{\text{in}}, \cdot)$ weights are replaced by 2^d bit shifts, followed by a single modular reduction at the end of each 2^d-size chunk. The fixed scalar C commutes through the outer sum and is absorbed once at protocol startup — with zero cost in the hot loop.

Two-level implementation over $\mathbb{F}_{2^8}$ and $\mathbb{F}_{2^{128}}$. The discussion above takes place in a single extension field $\mathbb{F}$ for simplicity. In our implementation, following the heavily optimized implementation of [Irr25], we use the field $\mathbb{F} = \mathbb{F}_{2^{128}}$ in GHASH form $\mathbb{F}_{2^{128}} = \mathbb{F}_2[\gamma]/(\gamma^{128} + \gamma^7 + \gamma^2 + \gamma + 1)$, and we exploit the subfield containment $\mathbb{F}_{2^8} \subset \mathbb{F}_{2^{128}}$ to apply the trick *twice*. Three pinned coordinates are placed in $\mathbb{F}_{2^8}$ using its standard AES representation $\mathbb{F}_{2^8} = \mathbb{F}_2[\alpha]/(\alpha^8 + \alpha^4 + \alpha^3 + \alpha + 1)$ with generator α, and four more are placed in $\mathbb{F}_{2^{128}}$ using γ. Thereby we derive a total of $2^3 \cdot 2^4 = 128$ pinned-eq weights of the form $C \cdot \alpha^k \cdot \gamma^j$ for $k \in [0, 8)$ and $j \in [0, 16)$, which replace 7 coordinates of $\mathbf{r}$ in total.

The two levels are handled differently in the hot loop. The $\mathbb{F}_{2^8}$ level uses the shift-then-reduce of the previous paragraph, producing one $\mathbb{F}_{2^8}$ scalar y_j per medium index $j \in [0, 16)$. The $\mathbb{F}_{2^{128}}$ level, in contrast, is absorbed entirely into a precomputed lookup table $T[j][v] := \gamma^j \cdot \varphi_8(v) \in \mathbb{F}_{2^{128}}$ for $j \in [0, 16)$ and $v \in [0, 256)$, where $\varphi_8 \colon \mathbb{F}_{2^8} \hookrightarrow \mathbb{F}_{2^{128}}$ is the subfield embedding. Built once at protocol startup, this 16×256 table (a one-time 64 KB precomputation) fuses the $\mathbb{F}_{2^8}$-to-$\mathbb{F}_{2^{128}}$ conversion with the 16-fold γ-progression. The constants commute through the outer sum and are absorbed once at protocol startup.

Importantly, we have verified that our fixed choice of constants does indeed generate linearly independent vectors over $\mathbb{F}_2$.

4.4 Skipping c

Recall that the goal in our use of the zerocheck protocol is to reduce checking that $\hat{a} \cdot \hat{b} - \hat{c}$ is identically zero on $\{0,1\}^m$ into individual multilinear evaluation claims on $\hat{a}$, $\hat{b}$ and $\hat{c}$. In this section we describe an optimization that reduces the cost of processing c in this protocol.

Skipping c: First Attempt. A trivial observation is that we can rewrite the zerocheck expression

$$\sum_{\mathbf{i} \in \{0,1\}^m} \text{eq}(\mathbf{r}, \mathbf{i}) \cdot \big(\hat{a}(\mathbf{i}) \cdot \hat{b}(\mathbf{i}) - \hat{c}(\mathbf{i})\big) \;\stackrel{?}{=}\; 0$$

as

$$\sum_{\mathbf{i} \in \{0,1\}^m} \text{eq}(\mathbf{r}, \mathbf{i}) \cdot \hat{a}(\mathbf{i}) \cdot \hat{b}(\mathbf{i}) \stackrel{?}{=} \sum_{\mathbf{i} \in \{0,1\}^m} \text{eq}(\mathbf{r}, \mathbf{i}) \cdot \hat{c}(\mathbf{i}), \tag{1}$$

and that the RHS is simply equal to $\hat{c}(\mathbf{r})$.

Thus, the prover computes $v = \hat{c}(\mathbf{r})$, sends it to the verifier and they run a sumcheck only on the LHS, avoiding any further processing of c. (Indeed, the claim $v = \hat{c}(\mathbf{r})$ is part of the output of the protocol.) At first glance this seems to reduce the overall cost of zerocheck by nearly 33%.

While this idea has merit, it also has a downside. Usually in the zerocheck protocol, when sending the first round sumcheck polynomial, the verifier knows a priori that its value on 0 and 1 have to be 0 and so the prover can compute one less evaluation. When employing the univariate skip (see Section 4.2), with parameter k_s, the zerocheck optimization means that we only need to report roughly 2^{k_s} evaluations rather than twice that. Concretely, the original zerocheck round-1 message $P(\lambda)$ has degree $2(2^{k_s}-1)$ and vanishes on S, so the prover transmits only its 2^{k_s} evaluations on the disjoint coset Λ. Once c is split off as in Eq. (1), the LHS sumcheck is no longer a zerocheck (its target value is $\hat{c}(\mathbf{r})$ rather than 0), so this saving is lost and the prover has to send all $2 \cdot 2^{k_s} - 1$ evaluations.

An improved strategy. To allow both optimizations to live side-by-side (at a minimal cost) we revisit Eq. (1). Two observations make this work.

First, we keep the zerocheck saving by running just the first round of sumcheck on the RHS as well: instead of splitting $\hat{c}$ off up-front, we have the prover send the round-1 messages $P^{ab}(\lambda)$ *and* $P^{c}(\lambda)$ separately, where P^{ab} is the univariate skip message for the LHS and P^{c} is the corresponding message for the RHS. The verifier checks that $P^{ab} - P^{c}$ vanishes on S (this is the original zerocheck identity, now applied to the difference) and samples a single challenge $\lambda \in \mathbb{F}$. The $\hat{c}$-claim is then read off directly as $P^{c}(\lambda) = \hat{c}(\lambda, \mathbf{r})$, so c still drops out of the residual sumcheck.

Second, computing P^{c} is itself cheap. At first glance, the univariate skip on the RHS would require computing the LDEs of all 2^{m-k_s} rows of c (viewing c as a $2^{m-k_s} \times 2^{k_s}$ matrix over $\mathbb{F}_2$, as in Section 4.2) — which would in fact be *more* expensive than processing c alongside a, b in the original protocol. But by linearity, we can first aggregate the rows of c under the eq-weights $\mathrm{eq}(\mathbf{r}, \cdot)$ to produce a single $\mathbb{F}_{2^{128}}$-valued vector of length 2^{k_s}, and then take a single LDE of that vector to obtain P^{c} on Λ. This makes P^{c} extremely cheap to compute.

One more optimization. The way we instantiate the R1CS from our circuits has a structural property: the constraint matrix C_0 is simply the identity matrix, so $c = z$. Thus, there is never a need to materialize c separately — the existing z-buffer plays the role of c wherever needed. Combined with the optimizations above, the $\hat{c}$-claim $\hat{c}(\lambda, \mathbf{r})$ is a direct claim on $\hat{z}$ at the same point. Thus, the vector c also does not need to be "lin-checked".

Remark 4. One caveat of these optimizations is that the claim that we get on $\hat{c}$ is at a different point than the one we get on $\hat{a}$ and $\hat{b}$. We deal with this using the known batching techniques of [RR24, CBBZ23], but note that there is a small cost associated with that.

4.5 Circuit walking

We focus throughout this subsection on the constraint matrix A_0; the same construction applies to B_0 (and the optimization of Section 4.4 fixes $C_0 = I$ so no further optimizations are needed for it). In a nutshell, we describe an optimization that enables us to avoid ever materializing the matrix A_0 by following the underlying circuit structure.

R1CS for circuits, and the cost of substitution. There are multiple ways to map a (Boolean) circuit into an R1CS instance. We highlight two extremes:

- *All wires.* Commit every internal wire of the circuit as a separate entry of z. Each row of A_0 has $O(1)$ nonzeros (only the wires immediately feeding the gate's A-side), but the witness is large.
- *Only ANDs.* Commit only the AND-gate outputs (and the inputs/outputs of the whole circuit). Every internal wire is then a fixed $\mathbb{F}_2$-linear combination of these committed values — evaluate the linear cascades symbolically, substitute everywhere, and the witness shrinks (often by a factor of two or more). The cost is that each row of A_0 now contains nonzeros at every committed column reachable, through the substitution, from the gate's A-side — typically a dense linear combination whose fan-in scales with the depth of intervening linear computation.

The substituted form is much better for almost everything — PCS, witness storage, witness generation, zerocheck. The catch is that any algorithm that needs to iterate over A_0's nonzeros pays the substituted cost, which can be drastically more expensive than the all-wires equivalent. (In our actual implementation this is most evident in our Keccak arithmetization, which has an extremely dense matrix A_0.) Next, we show how to mitigate the cost of this representation.

Warmup: witness generation as a circuit walk. In the substituted encoding above, only the AND-gate outputs are committed in z (alongside the circuit's inputs and outputs); the XOR-gate outputs are intermediate wires that the prover holds only transiently.

In the protocol we need to generate the vector $a = Az$ (this is part of what is known as *trace* generation). Recall that in our setting $A = I_K \otimes A_0$ and so this can be done via K vector-matrix multiplications by A_0.

However, it is easy to see that doing so is wasteful — a much better option is to simply evaluate the circuit, gate by gate, in topological order. By the definition of A_0, the A-side input of each AND-gate is exactly the corresponding entry of $A_0 z$, and the B-side input is the corresponding entry of $B_0 z$. So this forward walk *is* the matrix–vector products $A_0 z$ and $B_0 z$ — interleaved with per-gate ANDs and writes back into z, and computed at the cost of one $\mathbb{F}_2$-evaluation of the hash circuit, with no A_0 or B_0 ever materialized.

Lincheck's hot path. Recall from Section 4.1 that after the block-diagonal collapse, the lincheck prover and verifier only ever interact with the small base matrix $A_0 : \{0,1\}^{m_0} \times \{0,1\}^{m_0} \to \mathbb{F}_2$. The lincheck identity for A requires evaluating, for each Boolean column index $\mathbf{j}_{\text{in}} \in \{0,1\}^{m_0}$, the eq-weighted column marginal

$$\eta^A[\mathbf{j}_{\text{in}}] \;:=\; \hat{A}_0(\mathbf{r}_{y,\text{in}}, \mathbf{j}_{\text{in}}) \;=\; \sum_{\mathbf{i}_{\text{in}} \,:\, A_0[\mathbf{i}_{\text{in}}, \mathbf{j}_{\text{in}}]=1} \text{eq}(\mathbf{r}_{y,\text{in}}, \mathbf{i}_{\text{in}}),$$

which is then paired against the partial fold of the witness $\hat{z}(\mathbf{r}_{y,\text{out}}, \cdot)$ in an m_0-round multilinear sumcheck. Letting $E[\mathbf{i}_{\text{in}}] := \text{eq}(\mathbf{r}_{y,\text{in}}, \mathbf{i}_{\text{in}})$, this is a single matrix–vector product

$$\eta^A \;=\; A_0^\top \cdot E$$

— the same matrix A_0 that appeared in witness generation, just transposed and applied to E in place of z. The naive way to compute it is to materialize A_0 as a sparse Boolean matrix and, for each row $\mathbf{i}_{\text{in}}$, scatter $E[\mathbf{i}_{\text{in}}]$ into every column appearing in that row, paying one $\mathbb{F}$-addition per nonzero — in the substituted encoding, exactly the cost we want to avoid.

Walking the circuit backwards. Since the forward walk produced $A_0 z$ by processing the circuit's XOR and AND gates in topological order, we can produce $A_0^\top E$ by the transposed walk: traverse the same gates in reverse, with $\mathbb{F}$-additions in place of $\mathbb{F}_2$-XORs. Concretely, maintain a running marginal M^A over the circuit's wires, initialized to zero, and process the gates in reverse topological order:

- At an AND-gate, the gate's R1CS row carries weight $\mathrm{eq}(\mathbf{r}_{y,\mathrm{in}}, \cdot)$ at the row's index. Add this weight into $M^A[w]$ for each wire w on the gate's A-side; if w is committed in z, deposit directly into the corresponding entry of η^A.
- At an XOR-gate $y = x_1 \oplus \cdots \oplus x_k$, add $M^A[y]$ into each of $M^A[x_1], \ldots, M^A[x_k]$. (This is the transpose of XOR: a single output weight is distributed back to all its inputs.)

When the walk reaches the circuit's inputs (which are committed in z), deposit the remaining M^A into the corresponding entries of η^A. The result is exactly the column marginal η^A that the sparse scatter on the substituted A_0 would have produced — but no substituted matrix ever appears.

Cost. Each XOR-gate's backward step distributes one $\mathbb{F}$-value to each of its inputs, so the backward pass over the XOR sub-circuit costs as many $\mathbb{F}$-additions as the forward pass costs $\mathbb{F}_2$-XORs during witness generation. The AND-gate work likewise mirrors the forward work — one scatter per AND-gate. The total walker cost is therefore proportional to the cost of *evaluating the underlying circuit once*, in $\mathbb{F}$-arithmetic instead of $\mathbb{F}_2$-arithmetic, and is independent of how aggressive the substitution is. The walker also keeps memory traffic small: only a small wire-window of $\mathbb{F}$-elements is live at any time, fitting comfortably in cache. By contrast, the sparse scatter on the substituted A_0 has nonzero count proportional to (circuit size) $\times$ (substitution fan-in), and at large 2^{m_0} its parallel form requires per-thread accumulators of length 2^{m_0}, whose reduction is memory-bandwidth-bound.

Example: Keccak. The Keccak-f permutation is 24 rounds of a 1600-bit state, each round consisting of a linear part $\theta \circ \rho \circ \pi$ (plus the round constant) followed by 1600 χ AND-gates — $24 \cdot 1600 = 38{,}400$ AND-gates in total, and so (roughly) this many rows in A_0. The two encoding choices are very different:

- *All wires.* Commit every intermediate state $s_0, s_1, \ldots, s_{24}$ and every χ-output $t_0, \ldots, t_{23}$ — $49 \cdot 1600 = 78{,}400$ bits per Keccak (forcing the R1CS dimension to $m_0 = 17$). Each row of A_0 has $O(1)$ nonzeros.
- *Only ANDs.* Drop $s_1, \ldots, s_{23}$ and commit only s_0, s_{24}, $t_0, \ldots, t_{23}$ — $26 \cdot 1600 = 41{,}600$ bits per Keccak ($m_0 = 16$, a nearly $2\times$ witness shrink). But each row of A_0 now spreads through the substitution across s_0 and $t_0, \ldots, t_{r-1}$, and on average is several hundred times denser than the all-wires counterpart.

The circuit walker lets us keep the substituted encoding without paying the materialization cost: lincheck's hot path is computed in roughly one Keccak evaluation per call — well under a million $\mathbb{F}$-additions — instead of iterating through the tens of millions of nonzeros in the materialized substituted A_0.

Verifier symmetry. The walker is run identically by the prover and the verifier. The verifier's call costs the same circuit-evaluation-equivalent number of field additions, which is small compared to the verifier's other per-instance work (it is also independent of the batch size K thanks to the block-diagonal collapse of Section 4.1).

4.6 Input/output constraints

So far we have only proved that each of the K batched instances satisfies the same base R1CS. To express the cross-instance statement that the user actually cares about — e.g. that the instances correspond to a hash chain, or consistency with Merkle paths — we need to bind the per-instance *input/output (IO) regions* of the witness to an auxiliary glue circuit G, as outlined in Section 1. This subsection describes the generic mechanism, and then the specific instantiation of a hash-chain.

Generic IO via slot-aligned regions. Recall from Section 4.1 that the witness $\mathbf{z} : \{0,1\}^k \times \{0,1\}^{m_0} \to \mathbb{F}_2$ is a $K = 2^k$-fold stack of per-instance blocks of size 2^{m_0}. We adopt a uniform layout convention: *each instance reserves a fixed, byte-aligned slot for each of its IO regions* (e.g. the input state and the output state of a hash compression), positioned at fixed coordinates within the block. Concretely, an IO region of 2^{m_r} bits is placed at an aligned offset, so the corresponding sub-cube of $\mathbf{z}$ is indexed by $(\mathbf{i}_{\text{out}}, \mathbf{s}, \mathbf{b})$ where $\mathbf{i}_{\text{out}} \in \{0,1\}^k$ is the instance, $\mathbf{s} \in \{0,1\}^{m_0 - m_r}$ are the fixed (constant) high coordinates that select the slot, and $\mathbf{b} \in \{0,1\}^{m_r}$ ranges over the bits of the region.

With this convention, any IO claim expressible as a multilinear evaluation of a region's MLE flows back to a multilinear evaluation of $\hat{\mathbf{z}}$ at a point whose high coordinates are pinned to the slot's $\mathbf{s}$. The glue circuit G is in turn handled by a small auxiliary protocol whose only output is such a claim (or a batch of them), which is then folded into the PCS opening for $\hat{\mathbf{z}}$ alongside the zerocheck/lincheck claims via the standard MLE-batching technique [RR24, CBBZ23]. The cost of G on the prover therefore reduces to (i) the auxiliary protocol itself and (ii) one additional opening claim against $\hat{\mathbf{z}}$ — both of which we will see are essentially free for the hash-chain.

Hash-chain: the goal. A *hash-chain* statement asks the prover to show, for public endpoints $x_0, x_{2^n} \in \{0,1\}^\ell$, that

$$x_{i+1} = h(x_i) \quad \text{for all } i \in \{0, 1, \ldots, 2^n - 1\},$$

where h is the per-instance hash circuit (here ℓ is the input/output width, e.g. $\ell = 1600$ for Keccak). The 2^n instances are already proved to internally enforce $\mathit{output}_i = h(\mathit{input}_i)$ by the base R1CS; the chain claim is the cross-instance assertion that $\mathit{output}_i = \mathit{input}_{i+1}$ for all $i < 2^n - 1$, together with the public-endpoint constraints $\mathit{input}_0 = x_0$ and $\mathit{output}_{2^n-1} = x_{2^n}$. The prover is given the full chain $x_0, x_1, \ldots, x_{2^n}$ in the clear, so trace generation remains fully parallel across i.

The naive options are unattractive. *Witness aliasing* (committing x_i once and pointing input_{i+1} and output_i to the same physical entry) breaks the block-diagonal structure that drives Section 4.1. A *permutation argument* via grand product (e.g. a Plookup-style or [CBBZ23]-style permutation check) works but is wasteful: the underlying relation is just a length-2^n shift, not an arbitrary permutation. We use instead a tailored *shift argument* that exploits this structure and reduces the chain to a single MLE-evaluation claim on $\hat{\mathbf{z}}$ via one short sumcheck.

The shift argument. Write $\mathrm{In}(\mathbf{i})$ and $\mathrm{Out}(\mathbf{i})$ for the input and output *scalars* of instance $\mathbf{i} \in \{0,1\}^n$, obtained by collapsing the bit dimension of each region at a shared verifier-sampled

random point in $\mathbb{F}^{m_r}$ (by Schwartz–Zippel, the bit-level chain reduces to this scalar chain with overwhelming probability). Let $\mathrm{shift}(\mathbf{a}, \mathbf{b}) : \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}$ be the multilinear extension of the successor relation $\mathbf{b} = \mathbf{a} + 1$ on n-bit integers (i.e. for $\mathbf{a}, \mathbf{b} \in \{0,1\}^n$, $\mathrm{shift}(\mathbf{a}, \mathbf{b}) = 1$ iff $\mathbf{b}$ is the integer successor of $\mathbf{a}$, and 0 otherwise). A closed form is easily derived by splitting on the MSB and tracking the carry, and is evaluable in $O(n)$ field operations. The (interior) chain relation $\mathrm{Out}(\mathbf{i}) = \mathrm{In}(\mathbf{i} + 1)$ is then captured by the identity

$$\sum_{\mathbf{y} \in \{0,1\}^n} \mathrm{eq}(\boldsymbol{\tau}, \mathbf{y}) \cdot \mathrm{Out}(\mathbf{y}) \;=\; \sum_{\mathbf{y} \in \{0,1\}^n} \mathrm{shift}(\boldsymbol{\tau}, \mathbf{y}) \cdot \mathrm{In}(\mathbf{y}), \tag{2}$$

which the verifier checks via sumcheck after sampling a single challenge $\boldsymbol{\tau} \in \mathbb{F}^n$ — yielding a single MLE-evaluation claim on $\hat{\mathbf{z}}$ that is batched into the PCS opening alongside the zerocheck and lincheck claims.[9]

Crucially, because the input and output regions sit in two consecutive slots differing only in a single selector bit, the two MLE evaluations $\mathrm{In}(\boldsymbol{\tau}')$ and $\mathrm{Out}(\boldsymbol{\tau}')$ that the sumcheck would have produced at its challenge point $\boldsymbol{\tau}' \in \mathbb{F}^n$ can be merged into one MLE evaluation of $\hat{\mathbf{z}}$, by running a single extra sumcheck round over the selector bit.

Cost. The chain claim's evaluation point has a particularly favorable structure: its high coordinates contain $m - n - m_r - 1$ zero entries (the fixed slot-selector bits, all but s_0), each of which halves the live support of the eq-weighting in the PCS opening. Leveraging this fact, the chain claim adds well under 5% to the end-to-end prover wall-clock across BLAKE3, SHA-256, and Keccak.

Other IO circuits. The same template applies to other small glue circuits: a single auxiliary sumcheck that reduces G to one or a few $\hat{\mathbf{z}}$-evaluation claims at points whose high coordinates pin the IO slot. For instance, Merkle paths — the other IO circuit we instantiate — can be reduced analogously, with the auxiliary protocol expressing the parent/child relation as a sparse linear combination over the committed leaf and internal-node slots. We omit the details.

5 Evaluation

Our prototype Rust implementation of Flock and benchmarking scripts are available online[10]; see `BENCHMARKS.md` to reproduce benchmarks. The code was developed with the assistance of coding agents, specifically Claude Opus 4.7 and 4.8. The field arithmetic and ring-switching implementations are adapted from [Irr25], and the Ligerito implementation is adapted from [GNR26a] (all with significant modifications).

We evaluate Flock in terms of its ability to prove many independent executions of three hash function primitives: Keccak-f[1600] permutations, SHA-256 compressions, and BLAKE3 compressions. When possible, we compare its performance to other systems:

- Binius64 [Irr25] (pinned at `8f21b34`) for Keccak-f[1600], SHA-256, and BLAKE3. Since the original implementation does not support multi-threaded witness generation, we used coding agents to add this feature. Binius64 targets 96 bits of security.

[9] The public endpoint constraints $\mathrm{In}(\mathbf{0}) = \hat{x}_0$ and $\mathrm{Out}(\mathbf{1}^n) = \hat{x}_{2^n}$ are folded into Eq. (2) by standard tricks — the output endpoint via the boundary term that drops out of shift in characteristic 2, and the input endpoint via a random linear combination — without adding rounds or changing the sumcheck's structure. Details omitted.

[10] https://github.com/succinctlabs/flock

- Plonky3 [Plo24] (pinned at `109e95c`) for Keccak and BLAKE3. Plonky3's default configuration targets 113 bits under proximity gap *conjectures* that seem problematic following recent attacks [DG25, CS25, FS25]. That configuration only has 65 proven bits of security. We therefore adjusted the configuration to target 100 bits of proven security.
- Hashcaster [Sou24] for Keccak. The original implementation does not include a PCS, so we use the implementation from [han25] (pinned at `1af6fc5`). Hashcaster targets 100 bits of security.

Evaluation setup. All benchmarks were conducted on a single Apple M4 Max (36 GB RAM, 10 performance cores). Multi-threaded benchmarks use 10 threads, which matches the number of performance cores available. When measuring prover throughput, we take the best of three trials, following a warmup trial.

Security parameterization. We target 100-bits of security by default (but also briefly discuss a variant targeting 120-bits of security in Table 4). This means that the underlying IOP has round-by-round soundness error [CCH+18] at most 2^{-100} (unconditionally – without relying on any unproven conjectures). We rely on the cryptographic hardness of the SHA-256 compression function for collision resistance and Fiat-Shamir security.

We utilize the field $\mathbb{F}_{2^{128}}$ throughout. This more than suffices for protocols such as sumcheck, zerocheck and ring-switching. We use Ligerito [NA25] as our PCS but extend it to the list-decoding regime (see Appendix C). We set the initial code to be an (interleaved) Reed-Solomon code of rate $\rho = \frac{1}{2}$ and use distance and proximity gaps up to the Johnson bound [BCI+23, BCH+25] (in combination with a small amount of grinding to achieve the desired 2^{-100} error). One could additionally do grinding for the query phase. In our main benchmarks we do not do so, but in Table 4 we also give benchmarks for a "slim" variant of Flock that minimizes proof size and does introduce limited query grinding.

Concretely, for proving $\approx 2^{14}$ Keccak permutations the five levels use rates $1/2, 1/4, 1/8, 1/16, 1/32$ with $218, 106, 71, 53, 43$ queries and up to $17, 12, 9, 6, 4$ bits of proximity-gap grinding, respectively. The query counts follow the formula above (so they depend only on the rate), whereas the grinding scales mildly with the instance size.

5.1 Prover throughput

We measure prover throughput, i.e., the number of hash function primitives proven per second. For each system, we benchmark a range of batch sizes $(2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18})$, stopping once the peak memory usage roughly exceeds 10 GB.[11] We report the maximum prover throughput across the batch sizes, for both single-threaded and multi-threaded executions. For reference, we also report the native throughput of executing the hash function primitives on a single core, *without* specialized hardware instructions.

Keccak. Table 1 reports maximum proving throughputs for Keccak-f[1600] permutations. For padding reasons, the batch sizes for Flock and HashCaster are roughly 1.5× larger (e.g., $1.5 \cdot 2^{14}$ instead of 2^{14}) and the batch sizes for Plonky3 are roughly 1.33× larger. For single-threaded

[11] For Binius64, we limit the maximum batch size to be 2^{14}, since the offline circuit builder uses prohibitively large amounts of memory past this point.

	max throughput **keccak**/**s**	
system	single-thread	multi-thread
Flock	30.7k $_{(2^{14})}$	252k $_{(2^{16})}$
Hashcaster	17.2k $_{(2^{12})}$	66.2k $_{(2^{18})}$
Binius64	5.0k $_{(2^{12})}$	37.6k $_{(2^{14})}$
Plonky3	0.8k $_{(2^{10})}$	6.3k $_{(2^{12})}$
Native	7.51M	

Table 1: Proving vs. native computing throughputs for Keccak-f[1600] permutations. Subscripts indicate the approximate number of permutations at which proving throughput is maximized.

	max throughput, **sha-256**/**s**	
system	single-thread	multi-thread
Flock	42.1k $_{(2^{14})}$	338k $_{(2^{16})}$
Binius64	5.0k $_{(2^{12})}$	36.0k $_{(2^{13})}$
Spartan	0.1k $_{(2^{5})}$	0.3k $_{(2^{5})}$
Native	7.14M	

Table 2: Proving vs. native computing throughputs for SHA-256 compressions. Subscripts indicate the approximate number of compressions at which proving throughput is maximized.

performance, Flock reaches a maximum throughput of 30.7k permutations per second; this represents a 245× overhead over native execution. For multi-threaded performance, Flock reaches a maximum throughput of 252k permutations per second, which is roughly 4× faster than Hashcaster, 7× faster than Binius64, and 40× faster than Plonky3.

SHA-256. Table 2 reports maximum proving throughputs for SHA-256 compressions. For single-threaded performance, Flock reaches a maximum throughput of 42.1k compressions per second; this represents a 170× overhead over native execution. For multi-threaded performance, Flock reaches a maximum throughput of 338k compressions per second, which is roughly 9.4× faster than Binius64.

BLAKE3. Table 3 reports maximum proving throughputs for BLAKE3 compressions. For single-threaded performance, Flock reaches a maximum throughput of 82.1k compressions per second; this represents a 224× overhead over native execution. For multi-threaded performance, Flock reaches a maximum throughput of 661k compressions per second, which is roughly 14× faster than Binius64 and Plonky3.

S-two[12] [CGH+26] has benchmarks only for BLAKE2s and single-threaded. On the same machine it proves 2.9k BLAKE2s compressions per second. Since BLAKE3 has only 7 rounds as compared to BLAKE2s which has 10, we extrapolate that S-two should be able to prove about 4.1k BLAKE3 compressions per second – about 20× slower than Flock.

[12]Available at https://github.com/starkware-libs/stwo

	max throughput **blake3**/s	
system	single-thread	multi-thread
Flock	82.1k $_{(2^{16})}$	661k $_{(2^{18})}$
Binius64	6.1k $_{(2^{12})}$	47.1k $_{(2^{14})}$
Plonky3	5.9k $_{(2^{12})}$	45.9k $_{(2^{14})}$
Native	18.4M	

Table 3: Proving vs. native computing throughputs for BLAKE3 compressions. Subscripts indicate the approximate number of compressions at which proving throughput is maximized.

5.2 Additional benchmarks

Figure 1 plots Flock's BLAKE3 proving throughput at different batch sizes, both single-threaded and multi-threaded. Single-threaded throughput saturates at 2^{16} compressions, whereas multi-threaded throughput continues to rise, roughly approaching an 8× speedup.

Table 4 fixes the batch size to be $\approx 2^{14}$ Keccak permutations and reports the resulting prover throughput, proof size, and verifier time. We include three variants of Flock: *fast* (the default used elsewhere, PCS rate 1/2, list-decoding regime), *slim* (rate 1/4, which trades prover throughput for smaller proofs), and *120-security* (rate 1/2 in the unique-decoding regime, targeting 120-bit rather than 100-bit security). The exact number of permutations ($1.5 \cdot 2^{14}$ for Flock and Hashcaster, 2^{14} for Binius64, and $1.33 \cdot 2^{14}$ for Plonky3) depends on padding. Flock leads on throughput, produces the smallest proofs (386 KiB for the fast variant, 200 KiB for the slim variant), and has the fastest verifier (around 6 ms, single-threaded, across all variants).

system	m/t throughput	proof	verify
Flock (fast)	239k/s	386 KiB	5.8 ms
Flock (slim)	189k/s	200 KiB	5.3 ms
Flock (120-security)	241k/s	558 KiB	6.1 ms
Hashcaster	42.1k/s	664 KiB	28 ms
Binius64	38.0k/s	457 KiB	667 ms
Plonky3	5.8k/s	3.35 MiB	20.5 ms

Table 4: Throughputs, proof sizes, and verifier times for proving $\approx 2^{14}$ Keccak-f[1600] permutations.

Table 5 reports Flock's cost breakdowns for proving $\approx 2^{14}$ Keccak-f[1600] permutations, SHA-256 compressions, and BLAKE3 compressions, as a percentage of overall prover time. The shape is the same across all three hash function primitives. The PCS and zerocheck phases dominate, accounting for 84–86% of prover time, while witness generation and lincheck are comparatively cheap.

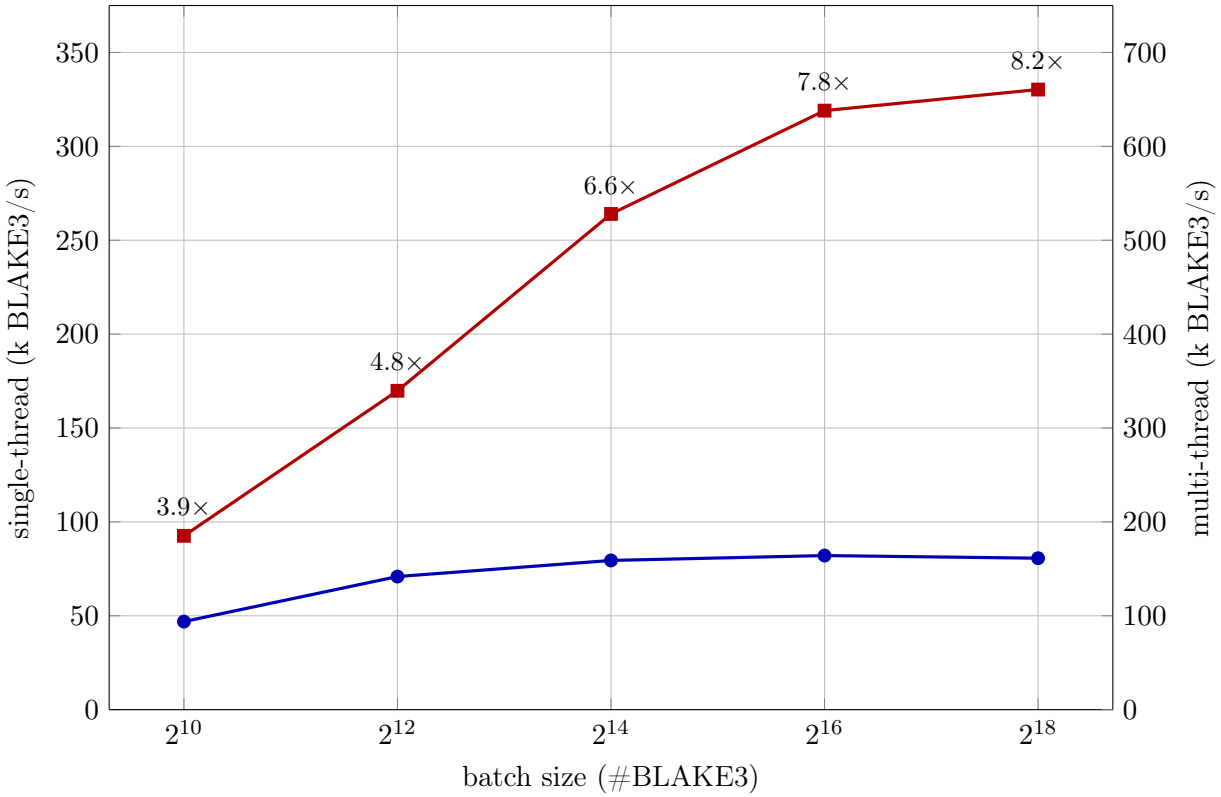

Figure 1: Flock BLAKE3 proving throughput vs. batch size. Blue (left axis) is single-threaded; red (right axis) is multi-threaded (10 threads). Each multi-threaded point is labeled with its speedup over the single-threaded prover at the same batch size.

phase	Keccak	SHA-256	BLAKE3
witness gen	8.0%	10.1%	5.0%
PCS commit	26.4%	26.1%	24.4%
zerocheck	42.3%	38.6%	37.3%
lincheck	6.4%	5.9%	9.1%
PCS open	16.8%	19.3%	24.3%

Table 5: Flock's cost breakdown for proving $\approx 2^{14}$ hash function primitives.

Acknowledgments

We thank Tamir Hemo and Lev Soukhanov for useful discussions.

References

[AABS+20] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR*

Transactions on Symmetric Cryptology, 2020(3):1–45, 2020.

[ABM23] Tomer Ashur, Thomas Buschman, and Mohammad Mahzoun. Algebraic cryptanalysis of the HADES design strategy: Application to Poseidon and Poseidon2. Cryptology ePrint Archive, Paper 2023/537, 2023.

[ACFY25] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. WHIR: Reed–Solomon proximity testing with super-fast verification. In *Proceedings of the 44th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT '25, 2025.

[AGR+16] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 191–219. Springer, 2016.

[ARR25] Noor Athamnah, Noga Ron-Zewi, and Ron D. Rothblum. Linear prover IOPs in log star rounds. In Benny Applebaum and Huijia (Rachel) Lin, editors, *Theory of Cryptography - 23rd International Conference, TCC 2025, Aarhus, Denmark, December 1-5, 2025, Proceedings, Part I*, Lecture Notes in Computer Science, pages 335–368. Springer, 2025.

[BBB+25] Antoine Bak, Augustin Bariant, Aurélien Boeuf, Pierre Briaud, Morten Øygarden, and Atharva Phanse. The algebraic CheapLunch: Extending FreeLunch attacks on arithmetization-oriented primitives beyond CICO-1. Cryptology ePrint Archive, Paper 2025/2040, 2025.

[BCF+25] Martijn Brehm, Binyi Chen, Ben Fisch, Nicolas Resch, Ron D. Rothblum, and Hadas Zeilberger. Blaze: Fast SNARKs from interleaved RAA codes. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV*, Lecture Notes in Computer Science, pages 123–152. Springer, 2025.

[BCG+19] Eli Ben-Sasson, Alessandro Chiesa, Lior Goldberg, Tom Gur, Michael Riabzev, and Nicholas Spooner. Linear-size constant-query IOPs for delegating computation. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019*, Lecture Notes in Computer Science, pages 494–521. Springer, 2019.

[BCH+25] Eli Ben-Sasson, Dan Carmon, Ulrich Haböck, Swastik Kopparty, and Shubhangi Saraf. On proximity gaps for Reed–Solomon codes. Cryptology ePrint Archive, Paper 2025/2055, 2025.

[BCI+23] Eli Ben-Sasson, Dan Carmon, Yuval Ishai, Swastik Kopparty, and Shubhangi Saraf. Proximity gaps for Reed–Solomon codes. *Journal of the ACM*, 70(5):31:1–31:57, 2023.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In Martin Hirt and Adam D. Smith, editors, *Theory of Cryptography - 14th International Conference, TCC 2016-B, Beijing, China, October 31 - November 3, 2016, Proceedings, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 31–60, 2016.

[BDH11] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - a practical forward secure signature scheme based on minimal security assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography (PQCrypto) 2011*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011.

[BFRW25] Benedikt Bünz, Giacomo Fenzi, Ron D. Rothblum, and William Wang. TensorSwitch: Nearly optimal polynomial commitments from tensor codes. Cryptology ePrint Archive, Paper 2025/2065, 2025.

[BGKS20] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: sampling outside the box improves soundness. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, Seattle, Washington, USA, January 12-14, 2020*, LIPIcs, pages 5:1–5:32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[BK82] Richard P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, 31(3):260–264, 1982.

[Blo25] Remco Bloemen. Succinct multi-linear extensions. https://xn--2-umb.com/25/succint-mle/, 2025.

[But24] Vitalik Buterin. Possible futures of the Ethereum protocol, part 4: The Verge. https://vitalik.eth.limo/general/2024/10/23/futures4.html#starked-binary-hash-trees, 2024.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023*, Lecture Notes in Computer Science, pages 499–530. Springer, 2023.

[CCH+18] Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, and Ron D. Rothblum. Fiat-shamir from simpler assumptions. *IACR Cryptol. ePrint Arch.*, 2018:1004, 2018.

[CFW26] Alessandro Chiesa, Giacomo Fenzi, and Guy Weissenberg. Zero-knowledge IOPPs for constrained interleaved codes. *IACR Cryptol. ePrint Arch.*, 2026:391, 2026.

[CGH+26] Dan Carmon, Lior Goldberg, Ulrich Haböck, Leonardo Lerer, Ilya Lesokhin, Shahar Papini, and Shahar Samocha. S-two whitepaper. Cryptology ePrint Archive, Paper 2026/532, 2026.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020*, volume 12105 of *Lecture Notes in Computer Science*, pages 738–768. Springer, 2020.

[CS25] Elizabeth C. Crites and Alistair Stewart. On Reed-Solomon proximity gaps conjectures. *IACR Cryptol. ePrint Arch.*, 2025:2046, 2025.

[CT10] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In Andrew Chi-Chih Yao, editor, *Innovations in Computer Science (ICS) 2010*, pages 310–331. Tsinghua University Press, 2010.

[DBE+26] Quang Dao, Ari Biswas, Liam Eagen, Andrew Milson, Shahar Papini, and Justin Thaler. The sum-check protocol over the monomial basis, and other optimizations. Cryptology ePrint Archive, Paper 2026/762, 2026.

[DG25] Benjamin E. Diamond and Angus Gruen. On the distribution of the distances of random words. *IACR Cryptol. ePrint Arch.*, 2025:2010, 2025.

[DHRR26] Rahul Dalal, Tamir Hemo, Eugene Rabinovich, and Ron D. Rothblum. VEIL: Lightweight zero-knowledge for hash-based multilinear proof systems. Cryptology ePrint Archive, Paper 2026/683, 2026.

[DKKW25] Justin Drake, Dmitry Khovratovich, Mikhail Kudinov, and Benedikt Wagner. Hash-based multi-signatures for post-quantum Ethereum. Cryptology ePrint Archive, Paper 2025/055. Minor revision in CIC 2025, 2025.

[DP25] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Madrid, Spain, May 4-8, 2025, Proceedings, Part IV*, Lecture Notes in Computer Science, pages 93–122. Springer, 2025.

[DP26] Benjamin E. Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology - EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part VII*, Lecture Notes in Computer Science, pages 3–32. Springer, 2026.

[DT24] Quang Dao and Justin Thaler. Constraint-packing and the sum-check protocol over binary tower fields. Cryptology ePrint Archive, Paper 2024/1038, 2024.

[FS25] Giacomo Fenzi and Antonio Sanso. Small-field hash-based SNARGs are less sound than conjectured. *IACR Cryptol. ePrint Arch.*, 2025:2197, 2025.

[GKK+26] Lorenzo Grassi, Dmitry Khovratovich, Katharina Koschatko, Christian Rechberger, Markus Schofnegger, Verena Schröppel, and Zhuo Wu. Poseidon(2)b: Binary field versions of Poseidon/Poseidon2. *IACR Commun. Cryptol.*, 2(4):15, 2026.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *Journal of the ACM*, 62(4):27:1–27:64, 2015.

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegg. Poseidon: A new hash function for zero-knowledge proof systems. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 519–535. USENIX Association, 2021.

[GKR25] Lorenzo Grassi, Katharina Koschatko, and Christian Rechberger. Poseidon and Neptune: Gröbner basis cryptanalysis exploiting subspace trails. Cryptology ePrint Archive, Paper 2025/954, 2025.

[GKS23] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the Poseidon hash function. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *Progress in Cryptology - AFRICACRYPT 2023 - 14th International Conference on Cryptology in Africa, Sousse, Tunisia, July 19-21, 2023, Proceedings*, Lecture Notes in Computer Science, pages 177–203. Springer, 2023.

[GNR26a] Kobi Gurkan, Andrija Novakovic, and Ron D. Rothblum. Bolt. https://github.com/bcc-research/bolt-rs, 2026.

[GNR26b] Kobi Gurkan, Andrija Novakovic, and Ron D. Rothblum. Bolt: Faster SNARKs from sketched codes. *IACR Cryptol. ePrint Arch.*, 2026:310, 2026.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016*, volume 9666 of *Lecture Notes in Computer Science*, pages 305–326. Springer, 2016.

[GRS23] Venkatesan Guruswami, Atri Rudra, and Madhu Sudan. Essential coding theory. Draft textbook, available at https://cse.buffalo.edu/faculty/atri/courses/coding-theory/book/, 2023.

[Gru24] Angus Gruen. Some improvements for the PIOP for ZeroCheck. Cryptology ePrint Archive, Paper 2024/108, 2024.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[han25] han0110. bench-hash-in-snark. https://github.com/han0110/bench-hash-in-snark/, 2025.

[HJR+26] Tamir Hemo, Kevin Jue, Eugene Rabinovich, Gyumin Roh, and Ron D. Rothblum. Jagged polynomial commitments (or: How to stack multilinears). In Joan Daemen and Emmanuel Thomé, editors, *Advances in Cryptology - EUROCRYPT 2026 - 45th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Rome, Italy, May 10-14, 2026, Proceedings, Part VII*, volume 16547 of *Lecture Notes in Computer Science*, pages 121–147. Springer, 2026.

[HR18] Justin Holmgren and Ron D. Rothblum. Delegating computations with (almost) minimal time and space overhead. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 124–135. IEEE Computer Society, 2018.

[HR22] Justin Holmgren and Ron D. Rothblum. Faster sounder succinct arguments and IOPs. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022*, Lecture Notes in Computer Science, pages 474–503. Springer, 2022.

[Irr25] Irreducible Team. Binius64. https://github.com/binius-zk/binius64, 2025.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 723–732. ACM, 1992.

[KST22] Abhiram Kothapalli, Srinath T. V. Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022*, volume 13510 of *Lecture Notes in Computer Science*, pages 359–388. Springer, 2022.

[Lam79] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report CSL-98, SRI International, 1979.

[lea25] lean Ethereum. leanVM: A minimal zkVM for lean Ethereum. https://github.com/leanEthereum/leanVM, 2025.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM*, 39(4):859–868, 1992.

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2 edition, 1997.

[Mer89] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO 1989*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[MG26] Simon-Philipp Merz and Àlex Rodríguez García. Skipping class: Algebraic attacks exploiting weak matrices and operation modes of Poseidon2. Cryptology ePrint Archive, Paper 2026/306, 2026.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[NA25] Andrija Novakovic and Guillermo Angeris. Ligerito: A small and concretely fast polynomial commitment scheme. Cryptology ePrint Archive, Paper 2025/1187, 2025.

[Ope26] OpenVM Contributors. SWIRL: Stacked WHIR with interaction reductions via LogUp. https://openvm.dev/swirl.pdf, 2026.

[Plo24] Plonky3 Team. Plonky3. https://github.com/Plonky3/Plonky3, 2024.

[Pol22] Polygon Zero Team. Plonky2: Fast recursive arguments with PLONK and FRI. https://github.com/0xPolygonZero/plonky2, 2022.

[RR24] Noga Ron-Zewi and Ron D. Rothblum. Local proofs approaching the witness length. *J. ACM*, 71(3):18, 2024.

[RR25] Noga Ron-Zewi and Ron D. Rothblum. Proving as fast as computing: Succinct arguments with constant prover overhead. *J. ACM*, 72(2):15:1–15:54, 2025.

[RRR21] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *SIAM J. Comput.*, 50(3), 2021.

[Set20] Srinath T. V. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020*, volume 12172 of *Lecture Notes in Computer Science*, pages 704–737. Springer, 2020.

[Sou24] Lev Soukhanov. Hashcaster. https://github.com/morgana-proofs/hashcaster, 2024. GKR over binary field implementation.

[Ste24] Matthias Johann Steiner. A zero-dimensional Gröbner basis for Poseidon. Cryptology ePrint Archive, Paper 2024/310, 2024.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013*, volume 8043 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2013.

[Tha22] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. Foundations and Trends in Privacy and Security, vol. 4, no. 2–4. now Publishers, 2022.

[Tha25] Justin Thaler. The path to secure and efficient zkVMs: How to track progress. https://a16zcrypto.com/posts/article/secure-efficient-zkvms-progress/, 2025.

[Val08] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008.

[Zca22] Zcash Foundation and Electric Coin Company. The halo2 book. https://github.com/zcash/halo2, 2022.

[ZSVD25] Ziyu Zhao, Antonio Sanso, Giuseppe Vitto, and Jintai Ding. Graeffe-based attacks on Poseidon and NTT lower bounds. Cryptology ePrint Archive, Paper 2025/1916, 2025.

A Full protocol

We describe the Flock protocol with the optimizations discussed in Section 4. We focus on describing the verifier, since only its behavior can affect soundness.

Setup. Let $\mathbb{F} = \mathbb{F}_{2^{128}}$. Fix base matrices $A_0, B_0 \in \mathbb{F}_2^{2^{m_0} \times 2^{m_0}}$, and let $K = 2^k$ be the number of instances. We assume $m_0 \geq 6$. The batched matrices are the block-diagonal

$$A = I_K \otimes A_0 \ , \quad B = I_K \otimes B_0$$

square matrices in $\mathbb{F}_2^{2^m \times 2^m}$ where $m \coloneqq k + m_0$.

Recall that the prover's goal is to prove, for some assignment $\mathbf{z} \in \mathbb{F}^{2^m}$, that $(A\mathbf{z}) \circ (B\mathbf{z}) = C\mathbf{z}$ and moreover $G(\mathbf{z}) = 0$ for some auxiliary input/output circuit G.

Quirky extensions. Fix $k_{\mathrm{s}} = 6$ and a subset $S \subseteq \mathbb{F}$ of size $2^{k_{\mathrm{s}}}$.[13] Set $D_0 \coloneqq \{0,1\}^{m_0 - k_{\mathrm{s}}} \times S$ and $D \coloneqq \{0,1\}^{m - k_{\mathrm{s}}} \times S$. We view $\mathbf{z}, \mathbf{a}, \mathbf{b}$ as functions $z, a, b \colon D \to \mathbb{F}_2$. We write $\hat{z} \colon \mathbb{F}^{m-k_{\mathrm{s}}+1} \to \mathbb{F}$ to denote the *quirky extension*[14] of z over $\mathbb{F}$, namely, the unique multivariate polynomial that is linear in the first $m - k_{\mathrm{s}}$ variables, is of degree less than $2^{k_{\mathrm{s}}}$ in the last variable, and agrees with z over D. We also write $\hat{a}, \hat{b}$ to denote the quirky extensions of a, b over $\mathbb{F}$.

Similarly, we view A_0, B_0 as functions mapping from $D_0 \times D_0$ to $\mathbb{F}$, and we write $\hat{A}_0, \hat{B}_0$ to denote their quirky extensions over $\mathbb{F}$.

Quirky PCS. We rely on a Boolean *quirky* PCS, which supports committing to quirky extensions of Boolean-valued functions $f : \{0,1\}^m \to \{0,1\}$.

Protocol. The full protocol proceeds as follows.

1. **Commit to the trace.** The prover commits to $\hat{z}$ using a Boolean quirky PCS (without out-of-domain sampling, see Remark 11), producing a commitment transcript cm.

2. **Zerocheck.** The initial claim is that

$$a(\mathbf{x}, y) \cdot b(\mathbf{x}, y) - z(\mathbf{x}, y) = 0 \quad \text{for all } (\mathbf{x}, y) \in D \ .$$

   Friendly challenges (Section 4.3). The verifier chooses $\mathbf{r} \in \mathbb{F}^{m-k_{\mathrm{s}}}$ so that the first 7 coordinates are fixed constants (which are $\mathbb{F}_2$-linearly independent, see Section 4.3), and the remaining coordinates are randomly sampled. The resulting claim is that

$$\sum_{\mathbf{x} \in \{0,1\}^{m-k_{\mathrm{s}}}} \mathrm{eq}(\mathbf{r}, \mathbf{x}) \cdot \big(a(\mathbf{x}, y) \cdot b(\mathbf{x}, y) - z(\mathbf{x}, y)\big) = 0 \quad \text{for all } y \in S \ .$$

   Univariate skip (Section 4.2). The prover sends polynomials P^{ab}, P^z claimed to be

$$P^{ab}(Y) = \sum_{\mathbf{x} \in \{0,1\}^{m-k_{\mathrm{s}}}} \mathrm{eq}(\mathbf{r}, \mathbf{x}) \cdot \hat{a}_{\mathbf{x}}(Y) \cdot \hat{b}_{\mathbf{x}}(Y) \ ,$$
$$P^z(Y) = \sum_{\mathbf{x} \in \{0,1\}^{m-k_{\mathrm{s}}}} \mathrm{eq}(\mathbf{r}, \mathbf{x}) \cdot \hat{z}_{\mathbf{x}}(Y) = \hat{z}(\mathbf{r}, Y) \ ,$$

   where $\hat{a}_{\mathbf{x}}, \hat{b}_{\mathbf{x}}, \hat{z}_{\mathbf{x}}$ are the low-degree univariate extensions of the functions $a(\mathbf{x}, \cdot), b(\mathbf{x}, \cdot), z(\mathbf{x}, \cdot)$, respectively. Observe that the degree of P^{ab} is at most $2 \cdot (2^{k_{\mathrm{s}}} - 1)$, and the degree of P^z is at most $2^{k_{\mathrm{s}}} - 1$. Since $P^{ab} - P^z$ is supposed to vanish on S, it suffices for the prover to specify P^{ab}, P^z by sending their evaluations on a disjoint subset $\Lambda \subset \mathbb{F} \backslash S$ of size $2^{k_{\mathrm{s}}}$. The verifier samples $\lambda \leftarrow \mathbb{F}$. The resulting claims are that

$$P^{ab}(\lambda) = \sum_{\mathbf{x} \in \{0,1\}^{m-k_{\mathrm{s}}}} \mathrm{eq}(\mathbf{r}, \mathbf{x}) \cdot \hat{a}_{\mathbf{x}}(\lambda) \cdot \hat{b}_{\mathbf{x}}(\lambda) \ , \tag{3}$$
$$P^z(\lambda) = \hat{z}(\mathbf{r}, \lambda) \ , \tag{4}$$

[13] The specific choice of k_{s} is flexible, but we pick 6 for efficiency.

[14] These types of extensions have different names in the literature, e.g., *oblong extensions* [Irr25] and *prismalinear extensions* [Ope26].

where $P^{ab}(\lambda)$ and $P^{z}(\lambda)$ are computed by the verifier. Observe that Eq. (4) is an evaluation claim $v^{\dagger} := P^{z}(\lambda)$ for $\hat{z}$ at the point $\mathbf{r}^{\dagger} := (\mathbf{r}, \lambda)$; this will be handled directly in Step 5.

For the claim in Eq. (3), the prover and verifier run an $(m - k_{\mathrm{s}})$-round sumcheck, resulting in a random point $\mathbf{s}' \in \mathbb{F}^{m-k_{\mathrm{s}}}$ along with evaluation claims

$$v_a \stackrel{?}{=} \hat{a}(\mathbf{s}) \ , \quad v_b \stackrel{?}{=} \hat{b}(\mathbf{s}) \quad \text{for } \mathbf{s} := (\mathbf{s}', \lambda) \ .$$

We split $\mathbf{s} = (\mathbf{s}_{\mathrm{out}}, \mathbf{s}_{\mathrm{in}})$; note that $\mathbf{s}_{\mathrm{out}} \in \mathbb{F}^{k}$ does not contain λ (since $k_{\mathrm{s}} \leq m_0$).

3. **Lincheck.** We reduce the claims v_a, v_b to a single claim on $\hat{z}$. We describe the reduction for $\hat{a}$ first. The claim is that

$$v_a \stackrel{?}{=} \hat{a}(\mathbf{s}) = \sum_{(\mathbf{j},y)\in D_0} \hat{A}_0(\mathbf{s}_{\mathrm{in}}, \mathbf{j}, y) \cdot \hat{z}(\mathbf{s}_{\mathrm{out}}, \mathbf{j}, y) \ .$$

The prover and verifier run an $(m_0 - k_{\mathrm{s}} + 1)$-round sumcheck on this claim, resulting in a random point $\mathbf{t} \in \mathbb{F}^{m_0 - k_{\mathrm{s}} + 1}$ along with evaluation claims for $\hat{A}_0(\mathbf{s}_{\mathrm{in}}, \mathbf{t})$ and $\hat{z}(\mathbf{s}_{\mathrm{out}}, \mathbf{t})$.

The same applies with B_0 in place of A_0, and batching the linchecks via Remark 2 reduces to a single claim

$$v^{\ddagger} \stackrel{?}{=} \hat{z}(\mathbf{r}^{\ddagger}) \quad \text{for } \mathbf{r}^{\ddagger} := (\mathbf{s}_{\mathrm{out}}, \mathbf{t}) \ ,$$

together with evaluation claims on $\hat{A}_0$ and $\hat{B}_0$ at $(\mathbf{s}_{\mathrm{in}}, \mathbf{t})$ that the verifier directly checks.

4. **Consistency with auxiliary circuit.** The prover and verifier run an auxiliary protocol, which reduces the claim that $G(\mathbf{z}) = 0$ into the claim that $\hat{z}(\mathbf{r}^{\star}) = v^{\star}$ (both $\mathbf{r}^{\star}$ and $v^{\star}$ are known to the verifier).

5. **Open the trace.** The prover and verifier run the PCS opening phase on cm with the evaluation claims $(\mathbf{r}^{\dagger}, v^{\dagger})$, $(\mathbf{r}^{\ddagger}, v^{\ddagger})$, and $(\mathbf{r}^{\star}, v^{\star})$. The verifier accepts if and only if the PCS verifier accepts and every check above passes.

Security. Let $L_{\max}$ be the upper bound on the list size from Eq. (10) (or $L_{\max} = 1$ for the unique decoding regime). The round-by-round soundness errors are:

- **Commit to the trace:** 0, since there is no out-of-domain sample.
- **Zerocheck:** The initial round's soundness error is $L_{\max} \cdot \frac{m - k_{\mathrm{s}} - 7}{|\mathbb{F}|}$, the univariate skip round's soundness error is $L_{\max} \cdot \frac{2 \cdot (2^{k_{\mathrm{s}}} - 1)}{|\mathbb{F}|}$, and the remaining rounds have soundness error $L_{\max} \cdot \frac{2}{|\mathbb{F}|}$.
- **Lincheck:** The univariate skip round's soundness error is $L_{\max} \cdot \frac{2 \cdot (2^{k_{\mathrm{s}}} - 1)}{|\mathbb{F}|}$, and the remaining rounds have soundness error $L_{\max} \cdot \frac{2}{|\mathbb{F}|}$.
- **Consistency with auxiliary circuit:** according to the auxiliary protocol, times $L_{\max}$.
- **Open the trace:** discussed in Appendix C.

B Overview of ring-switching

In multilinear proof-systems operating over binary fields, such as Flock, it is convenient to arithmetize while assuming access to the underlying bits of the witness. In contrast, the underlying commitment schemes are typically designed to commit to data represented over a moderately large field.

The *ring-switching* technique, introduced by Diamond and Posen [DP26] (building also on [DP25]), converts a polynomial commitment scheme designed to work over a large extension field into one that works over the base field, and in particular over bits. In this section we give an overview of their technique, while replacing one of their steps with a more modular, and arguably simpler, approach.

For sake of simplicity we restrict our attention to the 128-bit extension field $\mathbb{F}_{2^{128}}$ but note that the same approach can be generalized to any power-of-two extension.

B.1 Setup

We use $\mathbb{F}_2$ to denote the two-element binary field and $\mathbb{F} = \mathbb{F}_{2^{128}}$ to denote its degree-128 extension. Recall that $\mathbb{F}$ is a 128-dimensional vector space over $\mathbb{F}_2$. Let $\mathbf{b} = \{b_v\}_{v\in\{0,1\}^7}$ denote a basis for $\mathbb{F}$ over $\mathbb{F}_2$; we view the b_v's as elements of $\mathbb{F}$.

Let $q : \{0,1\}^m \to \{0,1\}$ be a Boolean-valued function. Let $q_{\mathrm{pkd}} : \{0,1\}^{m-7} \to \mathbb{F}$ denote the packing of q relative to $\mathbf{b}$: for $y \in \{0,1\}^{m-7}$,

$$q_{\mathrm{pkd}}(y) = \sum_{v\in\{0,1\}^7} q(y,v)\cdot b_v.$$

Note that q and q_{pkd} carry *the same information*: q_{pkd} replaces each 128-bit chunk of q's truth table by the single $\mathbb{F}$-element it encodes. Hence committing to q_{pkd} via a large-field scheme incurs no "embedding overhead".

Let $\hat{q} : \mathbb{F}^m \to \mathbb{F}$ and $\hat{q}_{\mathrm{pkd}} : \mathbb{F}^{m-7} \to \mathbb{F}$ denote the multilinear extensions of q and q_{pkd}, respectively. The ring-switch protocol is an interactive reduction from a multilinear evaluation claim $\hat{q}(r) = \alpha$, for $r \in \mathbb{F}^m$ and $\alpha \in \mathbb{F}$, to a multilinear evaluation claim on the packed polynomial $\hat{q}_{\mathrm{pkd}}(r') = \alpha'$, for some $r' \in \mathbb{F}^{m-7}$ and $\alpha' \in \mathbb{F}$.

B.2 The ring-switch protocol

Decomposing the claim. Decompose $r \in \mathbb{F}^m$ as $r = (r_{\mathrm{hi}}, r_{\mathrm{lo}}) \in \mathbb{F}^{m-7} \times \mathbb{F}^7$, where r_{lo} corresponds to the 7 packed coordinates. Observe that,

$$\alpha \;=\; \hat{q}(r_{\mathrm{hi}}, r_{\mathrm{lo}}) \;=\; \sum_{v\in\{0,1\}^7} \mathrm{eq}(r_{\mathrm{lo}}, v)\cdot \hat{q}(r_{\mathrm{hi}}, v). \tag{5}$$

For $v \in \{0,1\}^7$, define the *partial evaluation*:

$$s_v \;:=\; \hat{q}(r_{\mathrm{hi}}, v) \;\in\; \mathbb{F}.$$

The prover generates and sends to the verifier the sequence $(s_v)_{v\in\{0,1\}^7}$. It therefore suffices for the prover to convince the verifier that the values $(s_v)_{v\in\{0,1\}^7}$ are correct; assuming they are, the verifier can check that $\alpha \stackrel{?}{=} \sum_v \mathrm{eq}(r_{\mathrm{lo}}, v)\cdot s_v$ via Eq. (5). Thus, the key task is certifying the s_v's.

Remark 5 (Why Naive Recombination is Insecure). It is tempting to try to certify the s_v's in one shot. By definition of packing,

$$\sum_{v \in \{0,1\}^7} s_v \, b_v \;=\; \sum_{v} \hat{q}(r_{\mathrm{hi}}, v) \cdot b_v \;=\; \hat{q}_{\mathrm{pkd}}(r_{\mathrm{hi}}),$$

so the verifier could obtain $\hat{q}_{\mathrm{pkd}}(r_{\mathrm{hi}})$ from one opening of the committed packed polynomial and simply compare. This is complete but *not sound*.

In more detail, the basis $\{b_v\}$ is linearly independent over $\mathbb{F}_2$, but *not* over $\mathbb{F}$. Writing $\delta_v := s_v - \hat{q}(r_{\mathrm{hi}}, v) \in \mathbb{F}$ for the prover's errors, the check above amounts to the single $\mathbb{F}$-equation $\sum_v \delta_v \, b_v = 0$, which has an enormous space of nonzero solutions $\delta \in \mathbb{F}^{128}$. Adjoining the verifier's other linear check $\sum_v \delta_v \mathrm{eq}(r_{\mathrm{lo}}, v) = 0$ leaves two $\mathbb{F}$-equations constraining 128 unknowns in $\mathbb{F}$. A cheating prover has ample room to send false s_v that pass. Abstractly, the recombination map $\sum_v s_v \, b_v$ is the multiplication map $h : \mathbb{F} \otimes_{\mathbb{F}_2} \mathbb{F} \to \mathbb{F}$, which is not injective.

The fix: descend to $\mathbb{F}_2$, recombine slice-wise. At a high-level, the fix is to perform the linear combination *where the basis is actually independent* — over $\mathbb{F}_2$ — and only then lift back to $\mathbb{F}$.

As $\mathbf{b}$ is a basis, we can decompose each partial evaluation s_v over $\mathbb{F}_2$ as $s_v = \sum_{u \in \{0,1\}^7} s_{u,v} \cdot b_u$ for $s_{u,v} \in \mathbb{F}_2$. Likewise decompose the equality weights: there exists a function $A : \{0,1\}^{m-7} \times \{0,1\}^7 \to \mathbb{F}_2$ such that for each $y \in \{0,1\}^{m-7}$:

$$\mathrm{eq}(r_{\mathrm{hi}}, y) = \sum_{u \in \{0,1\}^7} A(y, u) \cdot b_u. \tag{6}$$

Substituting Eq. (6) into the definition of s_v we have that $s_v = \sum_{u \in \{0,1\}^7} b_u \cdot \Big(\sum_y A(y,u) \cdot q(y,v) \Big)$. As the inner sum is strictly over $\mathbb{F}_2$, and the b_u's are linearly independent, each one of the summands must match the corresponding $s_{u,v}$. Thus, we have that for all $u, v \in \{0,1\}^7$,

$$s_{u,v} = \sum_{y \in \{0,1\}^{m-7}} A(y,u) \cdot q(y,v). \tag{7}$$

These claims are *purely over* $\mathbb{F}_2$. We may therefore safely recombine them *over* v using the basis $\mathbf{b}$. Setting $s_u := \sum_v s_{u,v} \, b_v$ we have,

$$s_u \;=\; \sum_v \Big(\sum_y A(y,u) \, q(y,v) \Big) \cdot b_v \;=\; \sum_y A(y,u) \sum_v q(y,v) \cdot b_v \;=\; \sum_{y \in \{0,1\}^{m-7}} A(y,u) \cdot q_{\mathrm{pkd}}(y), \tag{8}$$

one claim for each $u \in \{0,1\}^7$. The right-hand side depends only on the packed polynomial q_{pkd} and on the publicly determined coefficients $A(y,u)$.

To see that soundness holds, assume one of the s_v's sent by the prover was incorrect. By the uniqueness of the $\mathbb{F}_2$-decomposition, $s_{u,v}$ is then incorrect for at least one $u \in \{0,1\}^7$. For this u, the recombined value $s_u = \sum_v s_{u,v} \cdot b_v$ must also be incorrect: by the $\mathbb{F}_2$-linear independence of the b_v's, the recombination is injective in the $\mathbb{F}_2$-coefficients $(s_{u,v})_v$.

B.2.1 Batching and the Sumcheck

There are 128 claims captured by Eq. (8). The verifier batches them: it samples $r'' \leftarrow \mathbb{F}^7$ and computes by itself $\beta_0 = \sum_{u\in\{0,1\}^7} \mathrm{eq}(r'', u) \cdot s_u$. What is left is to check that

$$\sum_{y\in\{0,1\}^{m-7}} B(y) \cdot q_{\mathrm{pkd}}(y) = \beta_0,$$

where $B(y) := \sum_{u\in\{0,1\}^7} \mathrm{eq}(r'', u) \cdot A(y, u)$.[15]

The prover and verifier now run the standard sumcheck on $\sum_{y\in\{0,1\}^{m-7}} B(y) \cdot \hat{q}_{\mathrm{pkd}}(y)$. The result of the sumcheck is a claim on $\hat{B}(r')$ and one on $\hat{q}_{\mathrm{pkd}}(r')$ for some $r' \in \mathbb{F}^{m-7}$. The claim on $\hat{q}_{\mathrm{pkd}}(r')$ is the output of the ring-switching reduction. As for the claim on $\hat{B}(r')$, we next show that the verifier can compute it by itself.

Evaluating $\hat{B}$. Diamond and Posen [DP26] give a direct way to evaluate $\hat{B}$ via the perspective of tensor algebras. We give here a slightly more modular approach, which we find to be conceptually simpler.

To show how the verifier can compute $\hat{B}$, we start by giving a closed form expression for B. Let $\Phi : \mathbb{F} \to \mathbb{F}$ be the $\mathbb{F}_2$-linear map determined by $b_u \mapsto \mathrm{eq}(r'', u)$ on the basis $\mathbf{b}$.

Claim 6. *For all $y \in \{0,1\}^{m-7}$:*

$$B(y) = \Phi\big(\mathrm{eq}(r_{\mathrm{hi}}, y)\big).$$

Proof.

$$\Phi\big(\mathrm{eq}(r_{\mathrm{hi}}, y)\big) \;=\; \sum_{u\in\{0,1\}^7} A(y, u) \cdot \Phi(b_u) \;=\; \sum_{u\in\{0,1\}^7} A(y, u) \cdot \mathrm{eq}(r'', u) \;=\; B(y),$$

where the first equality is by the $\mathbb{F}_2$ linearity of Φ and since A takes values in $\mathbb{F}_2$. □

The equality polynomial factors over $\mathbb{F}$ as

$$\mathrm{eq}(r_{\mathrm{hi}}, y) \;=\; \prod_i g_i(y_i),$$

where $g_i : \{0,1\} \to \mathbb{F}$ is defined as $g_i(0) = 1 - r_{\mathrm{hi},i}$ and $g_i(1) = r_{\mathrm{hi},i}$. Recall that multiplication by a fixed element of $\mathbb{F}$ is an $\mathbb{F}_2$-linear map on $\mathbb{F} \cong \mathbb{F}_2^{128}$ (via the basis $\mathbf{b}$), and so can be expressed as a 128×128 matrix over $\mathbb{F}_2$. Hence, for each i there are two such matrices, one for each value of $y_i \in \{0,1\}$. Thus, $B(y) = \Phi(\mathrm{eq}(r_{\mathrm{hi}}, y))$, evaluated on the Boolean hypercube, can be expressed as a width-128, length-$(m-7)$ *matrix branching program* in y: the program reads the bits of y in sequence and multiplies a 128-bit vector by the matrix selected at each layer by the corresponding bit. After reading the last bit it further applies the linear transformation Φ.

As shown by Holmgren and Rothblum [HR18] (cf. [HJR+26, Lemma 4.1] and [Blo25]), if a function can be computed by a small width (matrix) branching program then there is an efficient algorithm to compute its multilinear extension. This yields an efficient algorithm for evaluating $\hat{B}$, as desired. For completeness, we reproduce the [HR18] result in Appendix B.4.

[15] This batching incurs soundness error $7/|\mathbb{F}|$ by Schwartz–Zippel, which is negligible.

B.3 Ring-switching a quirky claim

The reduction above assumes a *multilinear* input claim $\hat{q}(r) = \alpha$. In Flock, however — as in Binius64 — ring-switching is invoked on the zerocheck's output, which uses the univariate-skip optimization (Section 4.2). The committed witness is then a *quirky extension* (see Appendix A): linear in all but the last variable, in which it has degree $< 2^{k_s}$. We now explain how to extend ring-switching to handle such a claim.

The structure of the packed coordinates enters the reduction at the decomposition step (see Eq. (5)) and its corresponding verifier check $\alpha \stackrel{?}{=} \sum_v \mathrm{eq}(r_{\mathrm{lo}}, v)\, s_v$. For a quirky claim the partial evaluations $s_v = \hat{q}(r_{\mathrm{hi}}, v)$ are unchanged — the suffix r_{hi} stays multilinear — and the *only* difference is that the weights $\mathrm{eq}(r_{\mathrm{lo}}, v)$ become a more general vector $w \in \mathbb{F}^{128}$, giving $\alpha = \sum_v w_v\, s_v$.

These weights have a simple closed form. Recall that the 7 packed coordinates consist of the $k_s = 6$ skipped coordinates together with one extra coordinate, so we may split the packed index as $v = (\sigma, b)$, where $\sigma \in \{0,1\}^{k_s}$ ranges over the 2^{k_s} skip points and $b \in \{0,1\}$ is the extra coordinate. The quirky claim reads the skipped coordinates at a univariate point $\zeta \in \mathbb{F}$ and the extra coordinate at a multilinear point $\rho \in \mathbb{F}$. The weight splits along this decomposition,

$$w_{(\sigma,b)} \;=\; L_\sigma(\zeta) \cdot \mathrm{eq}(\rho, b),$$

where L_σ is the Lagrange polynomial selecting the σ-th skip point (so that $L_\sigma(\zeta)$ evaluates the degree-$< 2^{k_s}$ univariate extension at ζ), and $\mathrm{eq}(\rho, b)$ is the usual multilinear weight for the extra coordinate.

Every other ingredient of the reduction depends only on r_{hi} and fresh randomness, so we simply substitute w_v for $\mathrm{eq}(r_{\mathrm{lo}}, v)$ and leave the rest untouched — including soundness, which pins down each s_v via the injective $\mathbb{F}_2$-recombination of Eq. (8), regardless of w.

B.4 Multilinear extension of matrix branching programs

Intuitively, a width w matrix branching program maintains as its state a vector $s \in \mathbb{F}^w$. It reads its input from left-to-right and every input bit specifies a $w \times w$ linear transformation to apply to the state. At the end we take the inner product of the state with a "sink" vector to get the result.

More formally, a *width-w read-once matrix branching program (MBP)* over $\{0,1\}^m$ specifies, for every layer $i \in [m]$ two transition matrices $M_i^{(0)}, M_i^{(1)} \in \mathbb{F}^{w \times w}$, together with a source vector $u \in \mathbb{F}^w$ and sink vector $v \in \mathbb{F}^w$. The function $f : \{0,1\}^m \to \mathbb{F}$ computed by the program is

$$f(x_1, \ldots, x_m) \;:=\; v^T \cdot M_m^{(x_m)} \cdot M_{m-1}^{(x_{m-1})} \cdots M_1^{(x_1)} \cdot u.$$

As observed in [HR18], the multilinear extension of such a function can then be computed using the following identity.

$$\hat{f}(r_1, \ldots, r_m) = v^T \cdot \prod_{i=m}^{1} \Big((1 - r_i)\, M_i^{(0)} + r_i\, M_i^{(1)} \Big) \cdot u, \tag{9}$$

for every $\mathbf{r} = (r_1, \ldots, r_m) \in \mathbb{F}^m$. The identity follows by observing that by definition it holds for Boolean-valued inputs and both sides of the equation are multilinear in $\mathbf{r}$.

C Overview of Ligerito

Flock uses Ligerito [NA25], instantiated with Reed–Solomon codes, as the underlying (dense) multilinear PCS. Ligerito is closely related to the WHIR PCS [ACFY25] but uses interleaved codes rather than Reed-Solomon codes (but the specific instantiation we focus on uses *interleaved* Reed-Solomon codes).

This appendix gives a self-contained overview of Ligerito, presented as a recursive construction of an interactive oracle PCS (see Section 2.4). Ligerito as described in [NA25] works in the unique decoding regime. We give a straightforward extension to the list-decoding regime, which enables fewer queries (and hence smaller proofs).

C.1 Commitment phase

Our goal is to commit to the multilinear extension of a function $f\colon \{0,1\}^m \to \mathbb{F}$, where $\mathbb{F}$ is a sufficiently large finite field. The PCS has two key parameters: the *folding factor* 2^ℓ and the *rate* $\rho \in (0,1)$.[16] We view f as a $2^\ell \times 2^{m-\ell}$ dimensional matrix over $\mathbb{F}$. Rows are indexed by $\{0,1\}^\ell$, and for $\mathbf{i} \in \{0,1\}^\ell$ we write $f_{\mathbf{i}}$ to denote the $\mathbf{i}$-th row of f.

Fix a Reed–Solomon code of rate ρ over $\mathbb{F}$, which encodes messages of length $k := 2^{m-\ell}$ to codewords of length $n := 2^{m-\ell}/\rho$ via a linear map $\mathrm{Enc}\colon \mathbb{F}^k \to \mathbb{F}^n$. The prover commits to f by sending the *interleaved* codeword C, which is the $2^\ell \times n$ matrix obtained by encoding each row of f, i.e., $C[\mathbf{i}, \cdot] := \mathrm{Enc}(f_{\mathbf{i}})$ for each $\mathbf{i} \in \{0,1\}^\ell$.

Binding in the list decoding regime. How tightly this commitment binds the prover depends on the proximity radius $\gamma \in (0,1)$ used during the opening phase, which directly translates to the number of queries issued by the verifier. In the *unique decoding* regime, the proximity radius is such that the committed rows are jointly close to at most one interleaved codeword, so C determines a single polynomial $\hat{f}$.

To reduce the query count we consider also the *list decoding* regime, where C is only guaranteed to be close to a small list of at most L candidate codewords. To bind the prover to a single one, the verifier requests an *out-of-domain* (OOD) [BGKS20] evaluation: it samples a random point $\mathbf{z} \in \mathbb{F}^m$ and the prover answers with the claimed value $\hat{f}(\mathbf{z})$. Importantly, this is done immediately after the prover sends the interleaved codeword (i.e., in the commitment phase). Next, we show that, with high probability over the choice of $\mathbf{z}$, all of the candidate of the candidate codewords in the nearby list disagree on their evaluation on $\mathbf{z}$, and so this evaluation uniquely identifies one of them.

By the Schwartz-Zippel lemma, since $\mathbf{z}$ is chosen at random, the multilinears corresponding to any two distinct codewords in the list agree on $\mathbf{z}$ with probability at most $m/|\mathbb{F}|$. By union bounding over all pairs, the probability that there exist a pair of distinct codewords in the list whose underlying multilinears agree on $\mathbf{z}$ is at most $\binom{L}{2} \cdot \frac{m}{|\mathbb{F}|}$.

Thus, we need to bound the size of the list L. Here, we use the fact that an interleaved code inherits its distance from the underlying base code. This means that the interleaved Reed-Solomon code (which we used to encode f) has relative distance $1-\rho$. By the Johnson bound (see, e.g., [GRS23, Chapter 7]), for any slack parameter $\eta > 0$, at proximity radius $\gamma = 1 - \sqrt{\rho} - \eta$ the

[16] Looking ahead, we remark that the construction of the PCS is recursive and these two parameters can—and will—be independently set in each level of recursion.

committed word is close to a list of at most

$$L \leq \frac{1}{2\eta\sqrt{\rho}} \tag{10}$$

codewords. Combining with the union bound, we find that the the out-of-domain evaluation fails to bind the prover with probability at most

$$\binom{L}{2} \cdot \frac{m}{|\mathbb{F}|} \leq \frac{1}{8\eta^2\rho} \cdot \frac{m}{|\mathbb{F}|} \ .$$

The slack parameter η should be thought of as some small fixed constant (e.g., $\eta = 0.01$).

C.2 Opening phase

Our goal is to prove an evaluation of the form $\hat{f}(\mathbf{y}) = y$. However, to facilitate recursion (and also as it is more useful) we show how to prove more general *linear evaluations* of the form

$$v = \langle w, f \rangle \coloneqq \sum_{\mathbf{x} \in \{0,1\}^m} w(\mathbf{x}) \cdot f(\mathbf{x}), \tag{11}$$

where the weight $w \colon \{0,1\}^m \to \mathbb{F}$ is "MLE-friendly": its multilinear extension $\hat{w}$ can be evaluated at any point in $\text{poly}(m)$ time (a multilinear evaluation claim $\hat{f}(\mathbf{r})$ is the special case $w = \text{eq}(\mathbf{y}, \cdot)$).

Batching claims. Besides the input evaluation claim $\langle w, f \rangle = v$, the verifier must also check the out-of-domain evaluation $\hat{f}(\mathbf{z}) = v'$, which can be written as $\langle w', f \rangle = v'$ for $w' = \text{eq}(\mathbf{z}, \cdot)$. To do so, it batches them into a single evaluation claim $\langle w + \alpha \cdot w', f \rangle = v + \alpha \cdot v'$, where $\alpha \in \mathbb{F}$ is a random coefficient (this can be generalized to batch any number of evaluation claims). Observe that if w are w' are MLE-friendly, then so is $w + \alpha \cdot w'$. Thus, we can focus on proving a single evaluation claim below.

Sumcheck and folding. The prover and verifier start by running ℓ rounds of sumcheck on Eq. (11)'s claim. The residual claim is

$$\sum_{\mathbf{x} \in \{0,1\}^{m-\ell}} w'(\mathbf{x}) \cdot f'(\mathbf{x}) = v'', \tag{12}$$

where $v'' \in \mathbb{F}$ is held by the verifier and $f', w' \colon \{0,1\}^{m-\ell} \to \mathbb{F}$ are obtained by partially evaluating (aka *folding*) $\hat{f}, \hat{w}$ at a random point $\mathbf{s} \in \mathbb{F}^\ell$, i.e., $f'(\mathbf{x}') \coloneqq \hat{f}(\mathbf{s}, \mathbf{x})$ and $w'(\mathbf{x}') \coloneqq \hat{w}(\mathbf{s}, \mathbf{x}')$. Observe that if w is MLE-friendly, then so is w'.

Commit and recurse. After the ℓ sumcheck rounds, the prover commits to f' recursively, using a smaller instantiation of the IOPCS which supports $m - \ell$ variables. We can then run the smaller opening phase to prove Eq. (12).

The problem is that nothing so far forces the prover to actually commit to f', i.e., it may commit to an unrelated g that happens to satisfy Eq. (12). Hence, we must additionally check that the committed polynomial g is indeed f'.

Consistency check. If f' is not equal to g, then $\mathrm{Enc}(f')$ must be (coordinate-wise) far from $\mathrm{Enc}(g)$. The verifier can therefore check consistency by sampling t random coordinates $J = \{j_1, \ldots, j_t\} \subset [n]$, and testing

$$\mathrm{Enc}(g)[j] = \mathrm{Enc}(f')[j] \quad \forall j \in J \ .$$

As Enc corresponds to a linear code, this is a set of t linear evaluations of g; moreover, using the fact that it is a Reed–Solomon code, it can be shown that the corresponding weights are MLE-friendly [ACFY25, NA25] (cf. [BFRW25, Lemma 7.5]). Thus, the prover may recursively prove these claims alongside the main claim $\langle g, w' \rangle = v''$ (via batching).

It remains to show that the verifier can obtain $\mathrm{Enc}(f')[j]$ for an arbitrary coordinate $j \in [n]$, from the original commitment matrix C. Let $G \in \mathbb{F}^{n \times k}$ be the generator matrix of the Reed–Solomon code, i.e., the matrix defining $\mathrm{Enc}(u) \coloneqq Gu$, and let G_j denote the j-th row of G. We have the following identity for every coordinate j:

$$\mathrm{Enc}(f')[j] = \langle G_j, f' \rangle = \left\langle G_j, \sum_{\mathbf{i} \in \{0,1\}^\ell} \mathrm{eq}(\mathbf{s}, \mathbf{i}) \cdot f_i \right\rangle = \sum_{\mathbf{i} \in \{0,1\}^\ell} \mathrm{eq}(\mathbf{s}, \mathbf{i}) \cdot C[\mathbf{i}, j] \ .$$

Thus, the verifier can access the j-th coordinate of $\mathrm{Enc}(f')$ by reading the j-th column of C and computing $\sum_{\mathbf{i}} \mathrm{eq}(\mathbf{s}, \mathbf{i}) \cdot C[\mathbf{i}, j]$ itself.[17]

C.3 Security

We sketch the opening phase's soundness analysis, with a focus on deriving precise round-by-round soundness errors of the protocol.

Let $\mathrm{RS}[\mathbb{F}, n, k]$ denote the Reed–Solomon code used in the PCS; its minimum distance is known to be $\delta = 1 - \rho$, where $\rho = k/n$ is the rate of the code. Recall that a 2^ℓ-wise interleaved codeword is a matrix $U \in \mathbb{F}^{2^\ell \times n}$ where, for each $\mathbf{i} \in \{0,1\}^\ell$, the row $U[\mathbf{i}, \cdot]$ is a codeword of $\mathrm{RS}[\mathbb{F}, n, k]$. Let $C \in \mathbb{F}^{2^\ell \times n}$ be an interleaved word, and fix a proximity radius $\gamma \in (0, 1)$. We say:

- C agrees with an interleaved codeword U on a set of columns $A \subseteq [n]$ if, for every $\mathbf{i} \in \{0,1\}^\ell$ and $j \in A$, it holds that $C[\mathbf{i}, j] = U[\mathbf{i}, j]$.
- C agrees with the interleaved code $\mathrm{RS}[\mathbb{F}, n, k]^{2^\ell}$ on A if there exists an interleaved codeword U agreeing with C on A.
- C is γ-close to an interleaved codeword U if they agree on some A with $|A| \geq (1 - \gamma) \cdot n$.

We write $\Lambda_\gamma(C)$ to denote the list of interleaved codewords U which are γ-close to C. For a point $\mathbf{s}_i \coloneqq (s_1, \ldots, s_i) \in \mathbb{F}^i$, we write $C_{\mathbf{s}_i}$ to denote the $\mathbf{s}_i$-folding of C, i.e., for $\mathbf{i} \in \{0,1\}^{\ell - i}$, the entry $C_{\mathbf{s}_i}[\mathbf{i}, j] \coloneqq$ is defined to be $\hat{c}_j(\mathbf{s}_i, \mathbf{i})$, where c_j denotes the j-th column of C.

Mutual correlated agreement. Our soundness analysis leverages *mutual correlated agreement* (MCA) of linear codes [ACFY25]. In particular, we rely on the MCA analysis for Reed–Solomon codes from [BCH+25].

[17] When compiling into a succinct argument, we arrange the columns of C as leaves of a Merkle tree. Thus, opening a column only requires a single Merkle path.

Theorem 7 (MCA up to unique decoding, adapted from [BCH+25, Corollary 1.4])**.** *Let* $\mathrm{RS}[\mathbb{F}, n, k]$ *be a Reed–Solomon code with minimum distance* $\delta = 1 - \frac{k}{n}$ *satisfying* $\delta \geq \frac{3\sqrt{2}}{n}$*. Let* $\gamma \in \left[\frac{\delta}{3}, \frac{\delta}{2} - \frac{3}{\delta n}\right]$*. For any interleaved word* $C \in \mathbb{F}^{2\times n}$*, it holds that*

$$\Pr_{s\leftarrow\mathbb{F}}\left[\exists A \subseteq [n], |A| \geq (1-\gamma)\cdot n : \begin{array}{l} C_s \text{ agrees with } \mathrm{RS}[\mathbb{F}, n, k] \text{ on } A \\ \wedge\ C \text{ does not agree with } \mathrm{RS}[\mathbb{F}, n, k]^2 \text{ on } A \end{array}\right] \leq \frac{a}{|\mathbb{F}|} \ ,$$

where $a = \gamma \cdot n + 1$*.*

Theorem 8 (MCA up to Johnson bound, adapted from [BCH+25, Theorem 4.6])**.** *Let* $\mathrm{RS}[\mathbb{F}, n, k]$ *be a Reed–Solomon code. Denote* $\rho = k/n$*, the slightly reduced rate of the code. Let* $\gamma \in (0, 1 - \sqrt{\rho})$*,* $\eta \coloneqq 1 - \sqrt{\rho} - \gamma$*, and* $\mu = \max\left\{\left\lceil \frac{\sqrt{\rho}}{2\eta} \right\rceil, 3\right\}$*. For any interleaved word* $C \in \mathbb{F}^{2\times n}$*, it holds that*

$$\Pr_{s\leftarrow\mathbb{F}}\left[\exists A \subseteq [n], |A| \geq (1-\gamma)\cdot n : \begin{array}{l} C_s \text{ agrees with } \mathrm{RS}[\mathbb{F}, n, k] \text{ on } A \\ \wedge\ C \text{ does not agree with } \mathrm{RS}[\mathbb{F}, n, k]^2 \text{ on } A \end{array}\right] \leq \frac{a}{|\mathbb{F}|} \ ,$$

where $a = \frac{2(\mu+1/2)^5 + 3(\mu+1/2)\gamma\rho}{3\rho^{3/2}} \cdot n + \frac{\mu+1/2}{\sqrt{\rho}}$*.*

Lemma 9 (MCA commutes with list decoding)**.** *Let* $\mathrm{RS}[\mathbb{F}, n, k]$ *be a Reed–Solomon code. Let* $\gamma \in (0, 1)$*, and let* ϵ *be a corresponding MCA error from Theorem 7 or Theorem 8. For any* $\ell \in \mathbb{N}$ *and interleaved word* $C \in \mathbb{F}^{2^\ell \times n}$*, it holds that*

$$\Pr_{s\leftarrow\mathbb{F}}\left[\Lambda_\gamma(C_s) \neq \{U_s : U \in \Lambda_\gamma(C)\}\right] \leq 2^{\ell-1}\cdot\epsilon \ .$$

Proof. Follows from [ACFY25, Lemma 4.13] and a union bound over the $2^{\ell-1}$ rows of C_s. □

Round-by-round soundness errors. Let C be the interleaved word sent by the prover in the commitment phase. Let $L_{\max}$ be the upper bound on the list size $|\Lambda_\gamma(C)|$ from Eq. (10) (or $L_{\max} = 1$ for the unique decoding regime), and let ϵ be the MCA error probability from Theorem 8 (or Theorem 7 for the unique decoding regime).

At the start of the opening phase, suppose that the prover is bound to a polynomial, i.e., there is a unique codeword in $\Lambda_\gamma(C)$ (the list of nearby interleaved codewords) that decodes to a polynomial agreeing with the out-of-domain evaluation claim. Moreover, suppose this polynomial disagrees with the input evaluation claim. It follows that every codeword in $\Lambda_\gamma(C)$ decodes to a polynomial disagreeing with at least one of the two evaluation claims. We show that the verifier rejects with high probability:

- **Batching claims.** Suppose that every codeword in $\Lambda_\gamma(C)$ disagrees with an evaluation claim. By soundness of batching and a union bound over the list, we find that every codeword in $\Lambda_\gamma(C)$ disagrees with the batched evaluation claim, except with probability $L_{\max} \cdot \frac{1}{|\mathbb{F}|}$.
- **Sumcheck.** Before the i-th round of the sumcheck protocol, suppose that every codeword in $\Lambda_\gamma(C_{\mathbf{s}_{i-1}})$ does not satisfy the $(i-1)$-th round sumcheck claim (when $i = 1$, this is the batched evaluation claim). In the i-th round, the verifier samples $s_i \leftarrow \mathbb{F}$. By soundness of sumcheck and a union bound over the list, with all but probability $L_{\max} \cdot \frac{2}{|\mathbb{F}|}$ it holds that the the s_i-folding of any codeword in $\Lambda_\gamma(C_{\mathbf{s}_{i-1}})$ disagrees with the i-th round sumcheck claim. By Lemma 9, with all but probability $2^{\ell-i}\cdot\epsilon$ the set of s_i-foldings of codewords in $\Lambda_\gamma(C_{\mathbf{s}_{i-1}})$ is exactly $\Lambda_\gamma(C_{\mathbf{s}_i})$. Taking a union bound over both errors, we find that every codeword in $\Lambda_\gamma(C_{\mathbf{s}_i})$ disagrees with the i-th round sumcheck claim, except with probability $L_{\max} \cdot \frac{2}{|\mathbb{F}|} + 2^{\ell-i}\cdot\epsilon$.

- **Commit.** The prover uses a smaller instantiation of the PCS to commit to some function $g\colon \{0,1\}^{m-\ell} \to \mathbb{F}$; the probability that this step fails is bounded by the binding error of the smaller PCS's commitment phase.
- **Consistency check.** Suppose that g either does not satisfy the folded claim (i.e., the residual sumcheck claim), or its encoding $\mathrm{Enc}(g)$ is γ-far from $C_{\mathbf{s}}$. The former case will be handled by the next step. In the latter case, the verifier detects an inconsistency, i.e., $\mathrm{Enc}(g)[j] \neq C_{\mathbf{s}}[j]$ for some $j \in J$, with all but probability $(1-\gamma)^t$.
- **Recurse.** Suppose that g does not agree with the folded claim or one of the consistency check claims. The remaining round-by-round soundness errors follow from those of the smaller PCS's opening phase.

Remark 10. The number of queries made by the verifier is a direct function of the proximity radius γ; in particular, one must set $t := \frac{\lambda}{-\log(1-\gamma)}$ in order to get λ bits of soundness in the consistency check. Thus, it is beneficial to choose codes with lower rate (and hence higher γ) in the later rounds of recursion, where the prover overhead is minimal.

Remark 11 (No OOD sampling in the initial commitment). OOD sampling is usually preferable but requires the prover to perform an additional MLE evaluation, which is actually somewhat costly for the prover. We observe that the initial commitment's OOD sampling can be *dropped*, at the cost of the outer protocol (which uses the commitment) increasing its soundness error by a factor of the list size L (security follows from a union bound over the nearby codewords).

D Proof of the Cauchy-shift identity

We prove the Cauchy-shift identity used in the Unified Lookup optimization (Section 4.2). The proof rests on two standard facts about vanishing polynomials of $\mathbb{F}_2$-linear subspaces:

Claim 12 (Linearized vanishing polynomial; see e.g. [LN97, Ch. 3])**.** *Let S be an $\mathbb{F}_2$-linear subspace of $\mathbb{F}_{2^\tau}$ and set $Z_S(x) := \prod_{s\in S}(x+s)$. Then:*

1. *Z_S is* additive*: $Z_S(x+y) = Z_S(x) + Z_S(y)$ for all $x, y \in \mathbb{F}_{2^\tau}$.*
2. *The formal derivative Z'_S is constant on S: $Z'_S(s) = D$ for every $s \in S$, where $D := \prod_{s'\in S\setminus\{0\}} s'$.*

Lemma 13 (Translation-invariance of the LDE matrix)**.** *Let τ be a power of two, let $S \subseteq \mathbb{F}_{2^\tau}$ be an $\mathbb{F}_2$-linear subspace with $|S| \geq \tau$, let $\delta \in \mathbb{F}_{2^\tau} \setminus S$, and let $\Lambda := \delta + S$. Fix any $\mathbb{F}_2$-basis of S and use it to index S (resp. Λ) by $[|S|]$ via the natural $\mathbb{F}_2$-linear bijection (resp. its δ-translate). Let $M := \mathrm{NTT}_\Lambda \circ \mathrm{iNTT}_S$. Then for every $i \in [|S|]$, $b \in \{0,1,\ldots,|S|/\tau - 1\}$, and $j \in \{0,1,\ldots,\tau-1\}$,*

$$M\big[i,\, \tau b + j\big] \;=\; M\big[\, i \oplus \tau b,\, j\,\big].$$

Proof. Write $\{e_0, \ldots, e_{k_s-1}\}$ for the chosen basis, and for $k \in [|S|]$ with binary expansion $k = \sum_r k_r 2^r$, let $s_k := \sum_{r:k_r=1} e_r \in S$ and $\lambda_i := \delta + s_i \in \Lambda$ denote the corresponding field elements. By Lagrange interpolation on S (using $a - b = a + b$ in characteristic 2),

$$M[i,k] \;=\; \frac{Z_S(\lambda_i)}{(\lambda_i + s_k)\cdot Z'_S(s_k)},$$

where $Z_S(x) := \prod_{s \in S}(x+s)$ is the vanishing polynomial of S. By Claim 12(1), $Z_S(\lambda_i) = Z_S(\delta + s_i) = Z_S(\delta) + Z_S(s_i) = Z_S(\delta)$ for every i (the second term vanishes as $s_i \in S$). By Claim 12(2), $Z'_S(s_k) = D$ for every k. Setting $\mu := Z_S(\delta)$, we obtain

$$M[i,k] \;=\; \frac{\mu}{(\lambda_i + s_k)\cdot D}. \tag{13}$$

Finally, τ being a power of 2 with $\tau \le |S|$ gives $(\tau b) + j = (\tau b) \oplus j$ (no carry), and the $\mathbb{F}_2$-basis indexing gives $s_{(\tau b)\oplus j} = s_{\tau b} + s_j$ and $\lambda_{i \oplus (\tau b)} = \lambda_i + s_{\tau b}$. Substituting into (13),

$$\begin{aligned} M[i,(\tau b)+j] &= \frac{\mu}{(\lambda_i + s_{\tau b} + s_j)\cdot D} \\ &= \frac{\mu}{(\lambda_{i\oplus(\tau b)} + s_j)\cdot D} \;=\; M[i \oplus (\tau b), j]. \end{aligned}$$

□