Binius64 Blueprint

June 18, 2026

Abstract

This document specifies Binius64, a hash-based zero-knowledge succinct non-interactive argument of knowledge (ZK-SNARK) tailored to computations expressed over 64-bit machine words. Binius64 arithmetizes a computation as a circuit built from two native constraint forms—bitwise AND and 64-bit unsigned integer multiplication—acting on XOR-accumulations of bit-shifted words. This word-level constraint language matches the instruction set of modern processors and avoids the roughly 64-fold blowup that bit-level circuit encodings incur. The protocol works entirely in characteristic 2 and draws its verifier challenges from the GHASH field $\mathbb{F}_{2^{128}}$, so that the prover's field arithmetic is carried by the carryless-multiplication instructions (`CLMUL`, `PMULL`) already present on x86-64 and ARM64 processors for the AES-GCM and GMAC modes. Binius64 has a transparent setup, rests only on the collision resistance of a hash function modeled as a random oracle, and is plausibly post-quantum secure. Its proofs are succinct in size. We obtain zero knowledge by composing the core protocol with a Spartan-based outer argument, and treat the construction in full. This document is intended as the canonical protocol specification: it develops the construction in detail while deferring the proofs of standard cryptographic primitives to the peer-reviewed literature.

Contents

1 Introduction

1.1 Motivation

A *succinct non-interactive argument of knowledge*, or SNARK, lets a verifier check the result of a computation far more cheaply than re-executing it, while remaining convinced—up to a negligible soundness error—that the prover holds a witness for which the computation was carried out correctly. SNARKs are the central primitive of *verifiable computing* [Set20]: a small, trusted device can audit the output of an untrusted, possibly faulty, and far more powerful one. When the argument additionally reveals nothing about the witness beyond the truth of the statement, it is *zero-knowledge*, the property that underpins privacy-preserving applications such as verifiable credentials and confidential transactions.

Many of the computations one most wants to prove—hash functions such as SHA-256, SHA-3, and Blake3, block ciphers, and the bit manipulation at the core of cryptographic primitives—are naturally

expressed as operations on machine words: 64-bit XOR, AND, shifts, rotations, and integer multiplication. General-purpose SNARKs built over large prime fields can prove such operations only after an expensive bit-by-bit decomposition, and even most binary-field systems pay a cost per individual bit. Binius64 instead takes the 64-bit word as its unit of arithmetization. The constraint system (§3) offers exactly two native constraint forms—*BitAnd*, a bitwise AND between XOR-accumulations of shifted words, and *IntMul*, an unsigned 64-bit integer multiplication—and derives the remaining common operations (XOR, NOT, OR, integer addition and subtraction; §A) from them. The key device is the *shifted value index*, which fuses a reference to a prover word with a shift operation, so that shifts and rotations are absorbed into the operands of the AND and multiplication constraints rather than charged as separate constraints. Operating on words rather than bits reduces the number of constraints by up to a factor of 64 relative to a bit-level encoding, while retaining the efficiency of binary-field arithmetic.

That efficiency is rooted in the choice of field. Binius64 works entirely in characteristic 2. Binary-field addition is bitwise XOR; the prime field $\mathbb{F}_2$ is the natural home of a single bit; and—decisively for performance—multiplication in the GHASH field $\mathbb{F}_{2^{128}}$ is accelerated in hardware on precisely the processors we target. The carryless-multiplication instructions `CLMUL` on x86-64 and `PMULL` on ARM64, shipped to support the AES-GCM authenticated-encryption mode and the GMAC message-authentication code, supply exactly the field multiplication the prover needs. We accordingly take $K \coloneqq \mathbb{F}_{2^{128}}$, with security parameter $\lambda = 128$, as the field from which the verifier draws its random challenges (§2.1). The protocol is designed throughout for high throughput on commodity x86-64 and ARM64 CPUs, including the SIMD units of modern mobile processors, which makes it well suited to local proving.

The construction is hash-based. Its only cryptographic assumption is the collision resistance of a hash function, modeled in the security analysis as a random oracle; there is no trusted setup and no structured reference string, and the scheme is plausibly secure against quantum adversaries. These are conservative assumptions by the standards of the SNARK literature, and they are inherited directly from the polynomial commitment layer rather than bolted on.

At present, Binius64 is succinct in *proof size*: a proof is polylogarithmic in the size of the computation being proved. Verification is not yet succinct—the verifier reads the circuit description in the course of checking a proof—and reducing verification cost is ongoing work. We are careful to state this distinction precisely where it matters (§5.5); the generic compiler of §5 is succinct in both proof size and verifier time whenever the source protocol's queries are succinct, a condition the Binius64 verifier does not currently meet because it evaluates the constraint matrices directly.

1.2 Why Not Binary Spartan?

A natural way to build a multilinear SNARK over binary fields would be to adapt Spartan [Set20] to $\mathbb{F}_2$. The witness becomes a single multilinear polynomial over $\mathbb{F}_2$, which the binary-field polynomial commitment of Diamond and Posen [DP24] commits efficiently, and the rank-one constraints are checked by a sumcheck. The approach is appealing, but it founders on the *wiring check*. A rank-one constraint system over n witness bits has matrices A, B, C with $O(n)$ nonzero entries, and the prover's partial evaluation of these matrices entails $O(n)$ multiplications between elements of the large challenge field K and bits of $\mathbb{F}_2$. With $\lambda = \log_2 |K|$ the security parameter, this is $O(\lambda n)$ bit operations, and proving the sparse-matrix openings incurs a second factor of λ. The large field infects what ought to be a bit-level computation.

Binius64 removes this bottleneck by working at word granularity. Packing 64 bits into a word shrinks the constraint-matrix entry count by up to 64×; modeling the word width as scaling with λ, the prover's wiring cost falls from $O(\lambda n)$ back to $O(n)$ bit operations. The native BitAnd and IntMul constraints, together with the shifted value indices, are what let the system express realistic computations at this granularity without reintroducing per-bit overhead.

We do nonetheless return to Spartan, in a different guise. The zero-knowledge construction of §7 uses *IronSpartan* (§6), a commit-and-prove adaptation of Spartan, as an *outer* argument—applied not to the word-level witness but to the small circuit that describes the Binius64 verifier. There the assignment is already polylogarithmic in the size of the original computation, so the $O(\lambda n)$ bottleneck above never arises, and Spartan's simplicity and direct support for zero knowledge become exactly the right tools.

1.3 Outline of the Construction

The document develops Binius64 in layers, separating the information-theoretic protocol from its cryptographic compilation.

§2 fixes notation and recalls the building blocks: multilinear polynomials and the equality indicator, the binary fields used (the prime field $\mathbb{F}_2$, the Rijndael field $\mathbb{F}_{2^8}$, and the challenge field $K = \mathbb{F}_{2^{128}}$), the sumcheck protocol and its variants (the MLE-check, the zerocheck, batching, and the univariate skip), and the abstraction of a *linear interactive oracle proof* (LIOP) that organizes the rest of the document.

§3 defines the constraint system as a self-contained combinatorial object: prover data laid out as 64-bit words, the eight shift operations and the shifted value indices that name shifted words, and the BitAnd and IntMul constraints built from XOR-accumulations of these. Appendix A shows how the everyday operations of a 64-bit instruction set reduce to this language.

§4 specifies the *Binius64 LIOP*, the heart of the protocol. The prover commits a single oracle—the packed witness—and a sequence of sumcheck-based reductions (the BitAnd reduction, the GKR-based IntMul reduction, the shift reduction, ring-switching to the packed witness, and the public-input check) collapse satisfaction of the entire constraint system into a single linear query on that oracle.

§5 gives a generic compiler turning any LIOP into a SNARK, through the chain LIOP $\xrightarrow{\text{BaseFold}}$ IOP $\xrightarrow{\text{BCS}}$ IP $\xrightarrow{\text{Fiat–Shamir}}$ SNARK. Each LIOP linear query is discharged by a BaseFold [ZCF24] proximity test on a constrained Reed–Solomon code over K, the oracles are replaced by Merkle commitments via the BCS transform [BCS16], and the interaction is removed by Fiat–Shamir. Applied to the Binius64 LIOP, this yields the (non-zero-knowledge) Binius64 SNARK of §5.5.

§6 specifies the *IronSpartan LIOP*, a second, zero-knowledge LIOP for rank-one constraint systems, recast as a commit-and-prove argument. §7 then assembles the zero-knowledge SNARK by *proof composition*: the Binius64 LIOP is run as an inner protocol with its messages encrypted under one-time pads, and IronSpartan certifies in zero knowledge that the inner verifier would have accepted. The accompanying *Batched ZK BaseFold* compiler masks the polynomial-commitment layer, so that the composed argument is zero-knowledge end to end.

This is a protocol specification rather than a self-contained cryptographic treatise. Our aim is to pin down the construction precisely enough to serve as the reference for an implementation, and to fill in the design choices and engineering details that sit between the academic results we build on and a working system. Where a security property follows from established, peer-reviewed work, we cite it and state how it is used rather than reprove it; the novel reductions are developed in full.

2 Preliminaries

This section fixes notation and recalls the polynomial-algebra and protocol building blocks used throughout the document.

Boolean hypercube For $n \geq 0$, we write $\mathcal{B}_n := \{0,1\}^n$ for the n-dimensional Boolean hypercube. We will frequently identify $\mathcal{B}_n$ with the set $\{0, \ldots, 2^n - 1\}$ of nonnegative integers via the little-endian bit decomposition $i \mapsto (i_0, \ldots, i_{n-1})$, where $i = \sum_{j=0}^{n-1} i_j \cdot 2^j$.

Constant vectors For a length $n \geq 0$, we write 0^n and 1^n for the all-zeros and all-ones vectors, identified with the corresponding vertices of $\mathcal{B}_n$. We use the same string-power notation for any constant block of n identical coordinates within a longer argument list.

Functions For sets S and V, the set of functions $S \to V$ is denoted V^S. We use the same symbol f both for a function $f \in V^{\mathcal{B}_n}$ and for its corresponding length-2^n list of values; context will distinguish.

Inner product For any field K and vectors $u, v \in K^n$, we write $\langle u, v \rangle := \sum_{i=0}^{n-1} u_i \cdot v_i$ for the standard inner product.

Tensor product expansion For vectors $u \in K^a$ and $v \in K^b$, the *tensor product* $u \bigotimes v \in K^{a \cdot b}$ is the vector whose entry indexed by $(i, j) \in \{0, \ldots, a-1\} \times \{0, \ldots, b-1\}$ is $u_i \cdot v_j$, under the

lexicographic enumeration of pairs. For $n \geq 0$ and $x = (x_0, \ldots, x_{n-1}) \in K^n$, the *tensor product expansion* is the map $\mathsf{TensorExpand}_n \colon K^n \to K^{2^n}$ defined by

$$\mathsf{TensorExpand}_n(x) \coloneqq \bigotimes_{i=0}^{n-1} (1 - x_i, x_i),$$

where each factor $(1 - x_i, x_i)$ is in K^2. The entry of $\mathsf{TensorExpand}_n(x)$ indexed by $w \in \mathcal{B}_n$ equals $\prod_{i=0}^{n-1} \left((1-x_i)^{1-w_i} \cdot x_i^{w_i}\right)$, which is precisely $\widetilde{\mathsf{eq}}_n(x, w)$ for the equality indicator defined below. Thus $\mathsf{TensorExpand}_n(x) = \left(\widetilde{\mathsf{eq}}_n(x, w)\right)_{w \in \mathcal{B}_n}$ is the table of values of the partial evaluation $\widetilde{\mathsf{eq}}_n(x, \cdot)$ on the hypercube. The expansion can be computed in $O(2^n)$ field operations [Tha22, Lem. 3.8].

Multilinear polynomials For a ring R and indeterminates $X_0, \ldots, X_{n-1}$, we write $R[X_0, \ldots, X_{n-1}]^{\preceq 1}$ for the subspace of polynomials of individual degree at most one in each indeterminate; these are the *multilinear* polynomials. The evaluation map $t \mapsto (t(u))_{u \in \mathcal{B}_n}$ is a K-linear bijection between $K[X_0, \ldots, X_{n-1}]^{\preceq 1}$ and $K^{\mathcal{B}_n}$; for a function $f \in K^{\mathcal{B}_n}$, we write $\widetilde{f} \in K[X_0, \ldots, X_{n-1}]^{\preceq 1}$ for the unique multilinear whose restriction to $\mathcal{B}_n$ equals f, the *multilinear extension* of f. When f takes values in the subfield $\mathbb{F}_2 \subset K$, $\widetilde{f}$ has $\mathbb{F}_2$-coefficients. See [Tha22, Fact 3.5] for an alternative direct proof of uniqueness.

Equality indicator The *equality indicator* is the bivariate multilinear $\widetilde{\mathsf{eq}}(X, Y) \coloneqq (1 - X)(1 - Y) + XY$, which in characteristic 2 simplifies to $1 + X + Y$. For $n \geq 1$ its *n-fold* version is the $2n$-variate multilinear

$$\widetilde{\mathsf{eq}}_n(X, Y) \coloneqq \prod_{i=0}^{n-1} \widetilde{\mathsf{eq}}(X_i, Y_i), \qquad X, Y \in K^n,$$

taking value 1 on the diagonal of $\mathcal{B}_n \times \mathcal{B}_n$ and 0 elsewhere; the multilinear extension of any $f \colon \mathcal{B}_n \to K$ decomposes as $\widetilde{f}(X) = \sum_{u \in \mathcal{B}_n} f(u) \cdot \widetilde{\mathsf{eq}}_n(u, X)$. The characteristic-2 form uses only $n - 1$ multiplications, so $\widetilde{\mathsf{eq}}_n(r, r')$ can be evaluated in $O(n)$ field operations at any pair of points $r, r' \in K^n$. We abbreviate $\widetilde{\mathsf{eq}}_1$ as $\widetilde{\mathsf{eq}}$, and also write the bare symbol $\widetilde{\mathsf{eq}}$ when referring to the family generically; an *applied* indicator of arity $n \geq 2$ always carries its subscript.

Vector arguments We routinely flatten vector arguments into scalar ones: when a function g expects $a + b$ scalar inputs and $u \in K^a$, $v \in K^b$, we write $g(u, v)$ for $g(u_0, \ldots, u_{a-1}, v_0, \ldots, v_{b-1})$, listing the coordinates of each vector in order, and we group consecutive scalar arguments back into vectors when convenient. This abuse is most frequent for the equality indicator $\widetilde{\mathsf{eq}}_n$, whose two length-$n$ vector arguments are often written out coordinate-wise (as in $\widetilde{\mathsf{eq}}_3(\rho_0, \rho_1, \rho_2, u_0, u_1, u_2)$).

GKR The *GKR protocol* [GKR15] is an interactive reduction that, given a layered arithmetic circuit and an evaluation claim on the multilinear extension of its output layer at a point in K^n, recursively reduces, layer by layer via sumcheck on the wiring polynomials, to evaluation claims on the multilinear extension of the input layer.

2.1 Binary Fields

We work exclusively in characteristic 2. Three binary fields recur throughout: the prime field $\mathbb{F}_2$, in which the witness lives; the *Rijndael field* $\mathbb{F}_{2^8}$, used by the prover in the BitAnd reduction (§4.3); and a cryptographically large extension field $\mathbb{F}_{2^\lambda}$, for a power-of-two security parameter λ, from which the verifier draws random challenges. We abbreviate $K \coloneqq \mathbb{F}_{2^\lambda}$ in protocol specifications and elsewhere when convenient, and write K^* for its multiplicative group of units; in practice we take $\lambda = 128$. We require $8 \mid \lambda$, so that $\mathbb{F}_{2^8}$ embeds into K (see *Subfield embedding* below). The construction imposes two further divisibility conditions relating λ to the padded witness and public-segment sizes; these are stated, and arranged by padding, in §4.1.

Freshman's dream. In characteristic 2 the binomial cross-term vanishes, so squaring is additive: $(\alpha + \beta)^2 = \alpha^2 + \beta^2$ for all $\alpha, \beta \in \mathbb{F}_{2^n}$. Iterating, $(\alpha + \beta)^{2^j} = \alpha^{2^j} + \beta^{2^j}$ for every $j \geq 0$. This is the elementary form of the Frobenius linearity revisited below, and underlies the Frobenius-twisting manipulations of §4.4.

Construction. For any $n \geq 1$, fixing an irreducible polynomial $f(X) \in \mathbb{F}_2[X]$ of degree n, the quotient ring $\mathbb{F}_2[X]/(f(X))$ is a field of size 2^n, which we denote $\mathbb{F}_{2^n}$. Each $\alpha \in \mathbb{F}_{2^n}$ is represented as a polynomial of degree less than n, so the monomials $1, X, \ldots, X^{n-1}$ form an $\mathbb{F}_2$-basis of $\mathbb{F}_{2^n}$; concretely, elements of $\mathbb{F}_{2^n}$ are n-bit strings, and those of $K = \mathbb{F}_{2^\lambda}$ are λ-bit strings. Two binary fields from cryptographic practice serve as running examples: the *Rijndael field* $\mathbb{F}_{2^8} \cong \mathbb{F}_2[X]/(X^8+X^4+X^3+X+1)$, the base field of the AES block cipher, and the *GHASH field* $\mathbb{F}_{2^{128}} \cong \mathbb{F}_2[X]/(X^{128}+X^7+X^2+X+1)$, used in the AES-GCM authenticated-encryption mode and the GMAC message-authentication code.

Subfield embedding. When $m \mid n$, the field $\mathbb{F}_{2^m}$ occurs inside $\mathbb{F}_{2^n}$ as the unique subfield of size 2^m. We exploit this for the Rijndael field $\mathbb{F}_{2^8}$, which embeds into K under the assumption $8 \mid \lambda$ via a fixed embedding $\iota\colon \mathbb{F}_{2^8} \hookrightarrow K$. We refer to [DP24] for further background on representing and computing in binary fields.

Algebraic properties. The multiplicative group $\mathbb{F}_{2^n}^*$ is cyclic, so we may fix a generator $g \in \mathbb{F}_{2^n}^*$ whose powers g^a, for $a \in \{0, \ldots, 2^n - 2\}$, exhaust $\mathbb{F}_{2^n}^*$. The *Frobenius endomorphism* $\varphi\colon \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ sending $\alpha \mapsto \alpha^2$ is $\mathbb{F}_2$-linear, and in finite fields it is invertible. The n-fold iterate φ^n acts as the identity on $\mathbb{F}_{2^n}$; we use this in §4.4 to manipulate evaluation claims involving Frobenius twists on K.

2.2 Sumcheck

The *sumcheck protocol* [LFKN92; Tha22] is an interactive reduction that transforms a sum claim $s \stackrel{?}{=} \sum_{v \in \mathcal{B}_n} F(v)$ over a polynomial $F \in K[X_0, \ldots, X_{n-1}]$ into a single evaluation claim $s' \stackrel{?}{=} F(r)$ at a verifier-chosen point $r \in K^n$, with soundness error $n \cdot d/|K|$, where d is the maximum individual degree of F. In our applications F is a *composite of multilinears*,

$$F(X) = C\big(\widetilde{f}_0(X), \ldots, \widetilde{f}_{m-1}(X)\big),$$

for multilinears $\widetilde{f}_0, \ldots, \widetilde{f}_{m-1} \in K[X_0, \ldots, X_{n-1}]^{\preceq 1}$ and a *composition polynomial* C of total degree d (so F has individual degree at most d). For such F the prover runs in $O(2^n \cdot m \cdot d)$ field operations and the verifier in $O(n \cdot d)$ time [Gru24, §3]. The reduction is given in Protocol 2.1; several variants recur throughout the document. We bind variables from the highest index to the lowest—in round i binding X_{n-i-1} and sampling the challenge r_{n-i-1}—a convention we use throughout by default, as it lends itself to a more SIMD-efficient prover algorithm.

PROTOCOL 2.1 (Sumcheck). **Claim:** $\sum_{x \in \mathcal{B}_n} F(x) = s$, for $F \in K[X_0, \ldots, X_{n-1}]$ of individual degree $\leq d$.

Rounds: Set $s^{(0)} := s$. For $i = 0, \ldots, n-1$, write $k := n - i - 1$ (so round i binds X_k):

1. The prover sends the *round polynomial*

$$R^{(i)}(X) := \sum_{v \in \mathcal{B}_k} F(v, X, r_{k+1}, \ldots, r_{n-1}) \in K[X], \qquad \deg R^{(i)} \leq d,$$

   where v ranges over the low variables $X_0, \ldots, X_{k-1}$ and $r_{k+1}, \ldots, r_{n-1}$ are the previously sampled challenges; d of its $d+1$ coefficients are transmitted, the verifier recovering the last from the check below.

2. The verifier checks $R^{(i)}(0) + R^{(i)}(1) = s^{(i)}$, samples $r_k \leftarrow K$, and sets $s^{(i+1)} := R^{(i)}(r_k)$.

Output: the evaluation claim $F(r) = s^{(n)}$ at the point $r := (r_0, \ldots, r_{n-1}) \in K^n$.

MLE-check. A *multilinear extension check*, or *MLE-check*, is a sumcheck specialized to summands of the form $\widetilde{\mathsf{eq}}_n(\tau, \cdot) \cdot g(\cdot)$ for a challenge $\tau \in K^n$; since $\sum_{x \in \mathcal{B}_n} \widetilde{\mathsf{eq}}_n(\tau, x) \cdot g(x) = \widetilde{g}(\tau)$, an MLE-check reduces an evaluation claim on $\widetilde{g}$ at τ to an evaluation claim on g (Protocol 2.2). The tensor structure of $\widetilde{\mathsf{eq}}$ enables a more efficient prover algorithm [Gru24, §3].

PROTOCOL 2.2 (Multilinear-extension check)**. Claim:** $\sum_{x\in\mathcal{B}_n} \widetilde{\mathsf{eq}}_n(\tau,x)\cdot g(x) = s$, for a point $\tau \in K^n$ and $g \in K[X_0,\dots,X_{n-1}]$ of individual degree $\le d$.

Rounds: Set $s^{(0)} \coloneqq s$. For $i = 0,\dots,n-1$, write $k \coloneqq n-i-1$ (so round i binds X_k):

1. The prover sends the *round polynomial*

$$R^{(i)}(X) \coloneqq \sum_{v\in\mathcal{B}_k} \widetilde{\mathsf{eq}}_k\big((\tau_0,\dots,\tau_{k-1}),v\big)\cdot g(v,X,r_{k+1},\dots,r_{n-1}) \in K[X], \qquad \deg R^{(i)} \le d,$$

   where v ranges over the low variables $X_0,\dots,X_{k-1}$ and the equality-indicator weight on those unbound variables has been pulled out; d of its $d+1$ coefficients are transmitted.

2. The verifier checks $s^{(i)} = (1-\tau_k)\,R^{(i)}(0) + \tau_k\,R^{(i)}(1)$, samples $r_k \leftarrow K$, and sets $s^{(i+1)} \coloneqq R^{(i)}(r_k)$.

Output: the evaluation claim $g(r) = s^{(n)}$ at the point $r \coloneqq (r_0,\dots,r_{n-1}) \in K^n$.

Zerocheck. A *zerocheck* reduces a claim that a polynomial $t \in K[X_0,\dots,X_{n-1}]$ vanishes on $\mathcal{B}_n$ to an MLE-check: the verifier samples $\tau \leftarrow K^n$ and the parties run an MLE-check on $\widetilde{\mathsf{eq}}_n(\tau,\cdot)\cdot t(\cdot)$ with claimed sum 0. Soundness error is $n/|K|$ above the underlying MLE-check.

Batching. Several sumcheck claims $s_i \stackrel{?}{=} \sum_{v\in\mathcal{B}_n} F_i(v)$ over polynomials $F_0,\dots,F_{k-1} \in K[X_0,\dots,X_{n-1}]$ on a *common* number of variables can be discharged by a single sumcheck [CFS17]: the verifier samples a batching challenge $\alpha \leftarrow K$, and the parties run sumcheck on

$$\sum_{i=0}^{k-1} \alpha^i s_i \stackrel{?}{=} \sum_{v\in\mathcal{B}_n} \sum_{i=0}^{k-1} \alpha^i F_i(v).$$

Soundness of the reduction follows from Schwartz–Zippel applied to the batching polynomial in α.

Variable padding. The batched sumcheck above assumes the F_i share a common number of variables; when they do not, we pad each up to the common count. To raise an n-variate claim $\sum_{x\in\mathcal{B}_n} F(x) = s$ to $n+\nu$ variables, the parties instead sumcheck the padded summand $F(X_0,\dots,X_{n-1})\cdot\widetilde{\mathsf{eq}}_\nu\big(0^\nu,(X_n,\dots,X_{n+\nu-1})\big)$ over $\mathcal{B}_{n+\nu}$, with the same target s. The extra factor $\widetilde{\mathsf{eq}}_\nu(0^\nu,\cdot)$ is supported on the single point $X_n = \cdots = X_{n+\nu-1} = 0$, where it equals 1, so summing over the ν padding variables contributes a factor of 1 and the padded claim holds iff the original one does.

Since we bind variables from the highest index down, the ν padding variables $X_n,\dots,X_{n+\nu-1}$ are bound first—in rounds $i = 0,\dots,\nu-1$—and the original variables only afterwards. The padding rounds are essentially free, and the remaining rounds reproduce the unpadded sumcheck up to a constant:

- In a padding round $i < \nu$ (binding $X_{n+\nu-1-i}$), every original variable is still unbound, so summing over them returns the original target s, and the round polynomial is

$$R^{(i)}(X) = s\cdot\widetilde{\mathsf{eq}}(0,X)\cdot\widetilde{\mathsf{eq}}_i\big(0^i,(r_{n+\nu-i},\dots,r_{n+\nu-1})\big),$$

  the bivariate $\widetilde{\mathsf{eq}}(0,X) = 1-X$ for the current variable times the constant accumulated from the already-bound padding challenges. The prover touches F not at all.

- In a round $i \ge \nu$, all padding variables are bound and contribute the constant $c \coloneqq \widetilde{\mathsf{eq}}_\nu\big(0^\nu,(r_n,\dots,r_{n+\nu-1})\big)$, so the round polynomial is the unpadded one scaled by c,

$$R^{(i)}(X) = c\cdot R_F^{(i-\nu)}(X),$$

  where $R_F^{(i-\nu)}$ is the round polynomial of the original n-variate sumcheck on F in its round $i-\nu$.

A polynomial entering the batch below the common variable count therefore costs only a string of trivial padding rounds before its ordinary sumcheck begins.

2.2.1 Univariate Skip

The *univariate skip* [Gru24, §5] cuts the round count, and lets the prover work over a subfield, when the composite's inputs are defined over a subfield of the challenge field.

The long axis. Fix a k-dimensional $\mathbb{F}_2$-linear subspace $D \subset K$, so $|D| = 2^k$, and choose an $\mathbb{F}_2$-basis $(\zeta_0, \ldots, \zeta_{k-1})$ identifying $\mathcal{B}_k \cong D$ via $i \mapsto \widehat{i} \coloneqq i_0\zeta_0 + \cdots + i_{k-1}\zeta_{k-1}$. We call this distinguished, size-$|D|$ dimension the *long axis*, and decorate quantities living on it with hats.

Oblong multilinears. An *oblong-multilinear* polynomial is multilinear in all but one variable, in which it instead has univariate degree below $|D| = 2^k$ over the domain D. The *oblong-multilinearization* of a multilinear t on $\mathcal{B}_k \times \mathcal{B}_\ell$ is

$$\widehat{t}(\widehat{I}, X) \coloneqq \sum_{i \in \mathcal{B}_k} \delta_D(\widehat{I}, \widehat{i}) \cdot t(i, X),$$

where δ_D is the bivariate equality-indicator on D, of individual degree below 2^k.

The reduction. The univariate skip runs sumcheck on a composite $C(\widehat{f}_0, \ldots, \widehat{f}_{m-1})$ of oblong multilinears. The first round, ranging over the size-$|D|$ domain in place of $\{0,1\}$, is discharged by a single univariate polynomial send rather than k binary rounds; the partial specialization

$$\widehat{t}(r_{\widehat{i}}, X) = \sum_{i \in \mathcal{B}_k} \delta_D(r_{\widehat{i}}, \widehat{i}) \cdot t(i, X)$$

at the resulting challenge $r_{\widehat{i}} \in K$ recovers an ordinary multilinear for the remaining rounds. Since the first-round univariate has degree below $d \cdot |D|$, where d is the composition degree, the prover can carry out its arithmetic entirely within any subfield $\mathbb{F} \subseteq K$ with $|\mathbb{F}| \geq d \cdot |D|$, embedding into K only once $r_{\widehat{i}}$ is drawn; this is the source of the prover speedup when the inputs $\widehat{f}_j$ take values in a small subfield.

2.3 Linear Interactive Oracle Proofs

A *linear interactive oracle proof* (LIOP) is an interactive protocol between a prover P and a verifier V for a relation $R \subset \{0,1\}^* \times \{0,1\}^*$. Both parties take as input a public instance x; the prover additionally holds a witness w with $(x, w) \in R$. The protocol proceeds in rounds.

- In each round, the prover may send one or more *oracle messages* together with a possibly empty non-oracle message. The i-th oracle message is a function $\pi_i \colon \mathcal{B}_{n_i} \to K$ for some $n_i \geq 0$ fixed by the protocol; equivalently, π_i is a vector in $K^{2^{n_i}}$. The verifier does not access π_i directly during the protocol. The protocol additionally fixes, for each oracle, a *randomizable support* $I_i \subseteq \mathcal{B}_{n_i}$, a (possibly empty) subset of coordinates whose role is explained below; unless stated otherwise $I_i = \emptyset$.
- The verifier responds with a challenge sampled uniformly from K (or, when convenient, from some fixed challenge space).
- After all rounds, the verifier issues a set of *linear queries*. The j-th linear query targets some oracle π_{i_j} and is specified by an n_{i_j}-variate multilinear polynomial $t_j \in K[X_0, \ldots, X_{n_{i_j}-1}]^{\preceq 1}$ together with a target value $s_j \in K$. The query is *satisfied* if

  $$\sum_{w \in \mathcal{B}_{n_{i_j}}} t_j(w) \cdot \pi_{i_j}(w) = s_j;$$

  i.e., the inner product of the oracle with the table of hypercube evaluations of t_j equals the target. Every linear query must *vanish on the randomizable support* of its target oracle: $t_j(w) = 0$ for all $w \in I_{i_j}$. The verifier accepts iff all linear queries are satisfied. The list of queries is a deterministic function of the transcript.

The LIOP is *complete* if for every $(x, w) \in R$ the honest protocol execution makes V accept with probability 1, and *sound with error* ε if for every x such that $(x, w) \notin R$ for all w and every (possibly cheating) prover P^*, the probability that V accepts is at most ε. Knowledge soundness is defined analogously via an efficient extractor.

Randomizable support. The randomizable support I_i marks coordinates of π_i that the honest prover fills with independent uniform randomness, drawing $\pi_i(w) \leftarrow K$ for each $w \in I_i$; the remaining coordinates of π_i are determined by the protocol as usual. The well-formedness requirement that every query targeting π_i vanish on I_i makes this randomness *free*: for any such query,

$$\sum_{w \in \mathcal{B}_{n_{i_j}}} t_j(w) \cdot \pi_{i_j}(w) = \sum_{w \in \mathcal{B}_{n_{i_j}} \setminus I_{i_j}} t_j(w) \cdot \pi_{i_j}(w),$$

since the omitted terms all have $t_j(w) = 0$. The query targets therefore depend only on the values of π_i off its randomizable support, so the random fill affects neither completeness nor soundness, and an LIOP with $I_i \neq \emptyset$ inherits these properties from its $I_i = \emptyset$ restriction. The randomness carries no information about the witness through the LIOP's own queries; it instead perturbs the *encoded* oracle of the SNARK compiler (§5), where it serves to mask the codeword openings that would otherwise leak. This is the mechanism underlying the zero-knowledge compiler (§7); the basic compiler of §5 and the Binius64 LIOP of §4—until the latter is used as the inner LIOP of the zero-knowledge composition of §7—take $I_i = \emptyset$ throughout and recover the standard notion. Choosing I_i large enough, and with coordinates suitably placed, to make the codeword openings simulatable is a quantitative condition deferred to §7.

Succinct queries. A linear query specified by (t_j, s_j) is *succinct* if the polynomial t_j can be evaluated at an arbitrary point of $K^{n_{i_j}}$ in $\mathrm{poly}(n_{i_j})$ operations over K. All linear queries appearing in our LIOPs are succinct. The succinct queries used here are closely related to, but strictly more restrictive than, the *succinct linear forms* of Chiesa, Fenzi, and Weissenberg [CFW26, §5], which permit arbitrary K-linear functionals admitting polylogarithmic-time evaluation; in our setting the functional is always the inner product against the hypercube evaluation table of a succinctly evaluable multilinear polynomial.

The basic example is the *multilinear-evaluation query* at a point $r \in K^{n_i}$: taking $t(X) := \widetilde{\mathsf{eq}}_{n_i}(r, X)$, the constraint $\sum_{w \in \mathcal{B}_{n_i}} \widetilde{\mathsf{eq}}_{n_i}(r, w) \cdot \pi_i(w) = s$ is precisely the assertion that the multilinear extension $\widetilde{\pi}_i$ takes value s at r, and the succinctness of $\widetilde{\mathsf{eq}}$ makes the query succinct. LIOPs whose linear queries are all of this form are sometimes called polynomial IOPs (PIOPs); we use the broader term LIOP to admit other succinct multilinear t_j where convenient.

The compilation of an LIOP into a SNARK using a polynomial commitment scheme is recalled in §5.

3 The Constraint System

The Binius64 constraint system operates on 64-bit words and supports two native constraint forms: *BitAnd constraints*, which express bitwise-AND relations between XOR-accumulations of shifted prover words, and *IntMul constraints*, which express unsigned-integer multiplication on 64-bit operands. This section defines the constraint language as a combinatorial object, independent of any protocol that verifies it. Appendix A sketches how this language expresses common derived operations (bitwise XOR, NOT, OR; integer addition and subtraction).

3.1 Prover Data Layout

At runtime, the prover supplies a list of 64-bit words. The list is divided into three contiguous *stretches* of compile-time-fixed lengths n_{const}, n_{inout}, and n_{witness}, denoted respectively the *constant*, *input–output*, and *witness* parts. We write w for the concatenation of all three, so that w has total length

$$n_{\mathrm{words}} := n_{\mathrm{const}} + n_{\mathrm{inout}} + n_{\mathrm{witness}},$$

and we write $w[y] \in \mathcal{B}_{64}$ for the y-th word, indexed by $y \in \{0, \ldots, n_{\mathrm{words}} - 1\}$.

The constant and input–output stretches are public: the verifier holds them. The constant stretch is fixed at compile time and is shared across all proofs; we record its values via a map $C\colon \{0, \ldots, n_{\mathrm{const}} - 1\} \to \mathcal{B}_{64}$. The input–output stretch is the public statement for the current proof, supplied as a map $u\colon \{0, \ldots, n_{\mathrm{inout}} - 1\} \to \mathcal{B}_{64}$. The remaining witness stretch is private to the prover and impacts the verifier's runtime only polylogarithmically. Throughout, we use the term *prover data* (rather than *witness*) for w to emphasize that only part of w is private.

The prover data w *satisfies* the constraint system with respect to the statement u if all of the following hold:

- the constant stretch matches C: $w[y] = C(y)$ for each $y \in \{0, \dots, n_{\text{const}} - 1\}$;
- the input–output stretch matches u: $w[n_{\text{const}} + y] = u(y)$ for each $y \in \{0, \dots, n_{\text{inout}} - 1\}$;
- every BitAnd constraint is satisfied (§3.4);
- every IntMul constraint is satisfied (§3.5).

3.2 Shifted Value Indices

The constraints in this system act not on raw prover words but on *shifted* versions of them. We support eight shift operations:

$$\mathsf{ShiftOps} \coloneqq \{\mathsf{sll}, \mathsf{srl}, \mathsf{sra}, \mathsf{ror}, \mathsf{sll32}, \mathsf{srl32}, \mathsf{sra32}, \mathsf{ror32}\}.$$

The first four are full-width 64-bit operations. For $v \in \mathcal{B}_{64}$ and $s \in \{0, \dots, 63\}$:

- $\mathsf{sll}(v, s)$ shifts v towards the more-significant end by s bits, zero-filling the vacated low bits;
- $\mathsf{srl}(v, s)$ shifts v towards the less-significant end by s bits, zero-filling the vacated high bits;
- $\mathsf{sra}(v, s)$ shifts v towards the less-significant end by s bits, replicating the most-significant bit into the vacated high positions;
- $\mathsf{ror}(v, s)$ rotates v right by s bits, so that the s least-significant bits of v wrap into the most-significant positions of the result.

The remaining four operations are the 32-*bit parallel* variants $\mathsf{sll32}, \mathsf{srl32}, \mathsf{sra32}, \mathsf{ror32}$. Each treats v as a pair of independent 32-bit halves (the low and high halves of v) and applies the corresponding shift to both halves in parallel. The 32-bit operations use only the five least-significant bits of the 6-bit shift amount s; the most-significant bit of s is ignored.

A *shifted value index* is a triple

$$(y, \text{op}, s) \in \{0, \dots, n_{\text{words}} - 1\} \times \mathsf{ShiftOps} \times \{0, \dots, 63\}$$

naming a prover-data index y, a shift operation op, and a shift amount s. Each such triple represents the value $\text{op}(w[y], s) \in \mathcal{B}_{64}$. For example, $(55, \mathsf{sll}, 10)$ represents `w[55] << 10`.

3.3 Constraint Arrays

We use a notational device to package XOR-accumulations of shifted value indices. Fix some total count n_{cons} of constraints, and assume that for each $x \in \{0, \dots, n_{\text{cons}} - 1\}$ we are given a list L_x of shifted value indices. The *constraint array* associated with $(L_x)_{x=0}^{n_{\text{cons}}-1}$ is the length-n_{cons} array $z \in \mathcal{B}_{64}^{n_{\text{cons}}}$ defined by

$$z[x] \coloneqq \bigoplus_{(y, \text{op}, s) \in L_x} \text{op}(w[y], s).$$

We will define BitAnd and IntMul constraints in terms of arrays of this form.

3.4 BitAnd Constraints

The constraint system specifies a total number n_{and} of BitAnd constraints. The x-th BitAnd constraint is a triple of lists of shifted value indices

$$(L_x^a, L_x^b, L_x^c), \quad x \in \{0, \dots, n_{\text{and}} - 1\},$$

which together induce three constraint arrays $a, b, c \in \mathcal{B}_{64}^{n_{\text{and}}}$ as in §3.3: for each $x \in \{0, \dots, n_{\text{and}} - 1\}$,

$$a[x] \coloneqq \bigoplus_{(y, \text{op}, s) \in L_x^a} \text{op}(w[y], s), \quad b[x] \coloneqq \bigoplus_{(y, \text{op}, s) \in L_x^b} \text{op}(w[y], s), \quad c[x] \coloneqq \bigoplus_{(y, \text{op}, s) \in L_x^c} \text{op}(w[y], s).$$

The prover data w *satisfies* the BitAnd constraints if for each $x \in \{0, \dots, n_{\text{and}} - 1\}$,

$$a[x] \wedge b[x] = c[x],$$

where $\wedge$ denotes bitwise AND on 64-bit words.

3.5 IntMul Constraints

The constraint system specifies a total number n_{mul} of IntMul constraints. The x-th IntMul constraint is a tuple of four lists of shifted value indices

$$(L_x^a, L_x^b, L_x^{c_{\text{lo}}}, L_x^{c_{\text{hi}}}), \quad x \in \{0, \dots, n_{\text{mul}} - 1\},$$

inducing four length-n_{mul} constraint arrays $a, b, c_{\text{lo}}, c_{\text{hi}} \in \mathcal{B}_{64}^{n_{\text{mul}}}$ as in §3.3. The prover data w *satisfies* the IntMul constraints if for each $x \in \{0, \dots, n_{\text{mul}} - 1\}$,

$$a[x] \cdot b[x] = 2^{64} \cdot c_{\text{hi}}[x] + c_{\text{lo}}[x],$$

where the four words are interpreted as nonnegative integers in $\{0, \dots, 2^{64} - 1\}$ and the multiplication is taken over the integers. Equivalently, $c_{\text{hi}}[x]$ and $c_{\text{lo}}[x]$ are the high and low 64-bit halves of the 128-bit unsigned product $a[x] \cdot b[x]$.

4 The Binius64 LIOP

This section specifies the Binius64 LIOP: an interactive oracle protocol that, given a constraint system (§3) and a public statement u, verifies that the prover holds a witness w satisfying the constraint system with respect to u. The compilation of this LIOP into a SNARK is discussed in §5.

Throughout, we assume the constraint-system parameters are padded so that $n_{\text{words}} = 2^{\ell_{\text{words}}}$, $n_{\text{and}} = 2^{\ell_{\text{and}}}$, $n_{\text{mul}} = 2^{\ell_{\text{mul}}}$, and $n_{\text{public}} := n_{\text{const}} + n_{\text{inout}} = 2^{\ell_{\text{public}}}$ are all powers of two. We additionally pad both the witness and the public segment so that $\lambda \mid 64 \cdot n_{\text{words}}$ and $\lambda \mid 64 \cdot n_{\text{public}}$; equivalently, $\ell_{\text{words}} \geq \log_2(\lambda) - 6$ and $\ell_{\text{public}} \geq \log_2(\lambda) - 6$. With $\lambda = 128$, this means $\ell_{\text{words}}, \ell_{\text{public}} \geq 1$. The first condition is needed for the packed-witness construction of §4.6 and the second for the public input check of §4.7. When the LIOP is compiled to a zero-knowledge SNARK (§7), the packed-witness oracle additionally carries a randomizable support (§2.3); the compiler of §7.2 imposes a lower bound on its size, and the padding above must be enlarged to host it. Padding entries are taken to be zero (or, for constraint-array lists, the empty list). Where convenient we identify $\mathcal{B}_{\ell_{\text{words}}} \cong \{0, \dots, n_{\text{words}} - 1\}$ and similarly for the other domains.

4.1 Protocol Overview

The protocol commits to a single K-valued oracle obtained by packing the witness w word-by-word, runs a sequence of sumcheck-based reductions, and finally issues a single linear query on the packed oracle. Concretely, w is viewed as a function $w \colon \mathcal{B}_6 \times \mathcal{B}_{\ell_{\text{words}}} \to \mathbb{F}_2$, where $(j, y) \in \mathcal{B}_6 \times \mathcal{B}_{\ell_{\text{words}}}$ indexes bit j of word y; its multilinear extension is the $6 + \ell_{\text{words}}$-variate multilinear

$$\widetilde{w}(J_0, \dots, J_5, Y_0, \dots, Y_{\ell_{\text{words}}-1}) \in K[J, Y]^{\preceq 1}.$$

The prover does not commit to w directly: it instead commits to the packed function $\mathsf{pack}_\lambda(w) \colon \mathcal{B}_{\ell_{\text{pack}}} \to K$, defined in §4.6, which encodes λ consecutive bits of w as a single K-element. The packed witness is the actual LIOP oracle, in keeping with the K-valued-oracle convention of §2.3.

The LIOP proceeds in seven phases:

1. **Packed witness commitment.** The prover sends the packed witness oracle $\pi := \mathsf{pack}_\lambda(w) \colon \mathcal{B}_{\ell_{\text{pack}}} \to K$, where $\ell_{\text{pack}} := \ell_{\text{words}} - (\log_2 \lambda - 6)$ and pack_λ is the packing map defined in §4.6. (Implementations commit to π via the polynomial commitment scheme of §5, but the LIOP itself treats π as an oracle of length $2^{\ell_{\text{pack}}}$.)

2. **BitAnd reduction (§4.3).** A sumcheck-based reduction transforms the satisfaction of the n_{and} BitAnd constraints into evaluation claims on the three oblong-multilinearizations $\widehat{a}, \widehat{b}, \widehat{c}$ of the BitAnd constraint arrays.

3. **IntMul reduction (§4.4).** A separate GKR-based reduction transforms the satisfaction of the n_{mul} IntMul constraints into evaluation claims on the four oblong-multilinearizations $\widehat{a}, \widehat{b}, \widehat{c}_{\text{lo}}, \widehat{c}_{\text{hi}}$ of the IntMul constraint arrays.

4. **Shift reduction (§4.5).** The seven evaluation claims produced by the two reductions are batched, and a two-phase sumcheck reduces them to a single evaluation claim on $\widetilde{w}$ at a point $(r_j, r_y) \in K^6 \times K^{\ell_{\text{words}}}$.

5. **Ring-switching (§4.6).** The evaluation claim $\widetilde{w}(r_j, r_y) = t$ on the $\mathbb{F}_2$-valued multilinear $\widetilde{w}$ is converted into a linear query on the packed K-valued oracle π, via the protocol of Diamond and Posen [DP24].

6. **Public input check (§4.7).** A separate linear query on π certifies that the packing of the public segment $C \,\|\, u$ agrees with the prefix of π that contains it. This query is then random-combined with the ring-switching query into a single linear query.

7. **Witness query.** The verifier issues the combined linear query on π.

Phases 2 and 3 may be executed in either order; their outputs are independent until they are batched in Phase 4.

4.2 Shift Indicator Polynomials

Each of the eight shift operations in $\mathsf{ShiftOps}$ (§3.2) acts on $\mathcal{B}_{64}$ at the bit level. To verify claims about shifted prover words via sumcheck, we lift each shift operation to a multilinear polynomial identity. The key tool is the *shift indicator polynomial*, a multilinear in 18 variables whose values on $\mathcal{B}_6^3$ describe which bit of the original word ends up in which bit of the shifted word. Following Diamond and Posen [DG25, §4.3], the existence of such an identity reduces shifted-witness evaluation claims to single witness evaluation claims, provided the shift indicator admits an efficient evaluation algorithm.

Recall (§4.1) that the prover data w is viewed as a function $w\colon \mathcal{B}_6 \times \mathcal{B}_{\ell_{\text{words}}} \to \mathbb{F}_2$ with multilinear extension $\widetilde{w}(J, Y)$. For each $\text{op} \in \mathsf{ShiftOps}$, the *shifted witness function* is

$$\mathsf{shift}_{\text{op}}(w)\colon \mathcal{B}_{\ell_{\text{words}}} \times \mathcal{B}_6 \to \mathcal{B}_{64}, \quad \mathsf{shift}_{\text{op}}(w)(y, s) \coloneqq \text{op}(w[y], s).$$

Composing with bit extraction gives a function on $\mathcal{B}_6 \times \mathcal{B}_{\ell_{\text{words}}} \times \mathcal{B}_6$ sending (i, y, s) to the i-th bit of $\text{op}(w[y], s)$; we write $\widetilde{\mathsf{shift}}_{\text{op}}(\widetilde{w})(I, Y, S)$ for its multilinear extension.

For each op we define the *shift indicator* function $\mathsf{shift\text{-}ind}_{\text{op}}\colon \mathcal{B}_6 \times \mathcal{B}_6 \times \mathcal{B}_6 \to \mathbb{F}_2$ by

$$\mathsf{shift\text{-}ind}_{\text{op}}(i, j, s) \coloneqq \begin{cases} 1 & \text{if the } i\text{-th bit of } \text{op}(v, s) \text{ equals the } j\text{-th bit of } v \text{ for every } v \in \mathcal{B}_{64}, \\ 0 & \text{otherwise.} \end{cases}$$

For example, $\mathsf{shift\text{-}ind}_{\mathsf{sll}}(30, 20, 10) = 1$ because $\mathsf{sll}(v, 10)_{30} = v_{20}$ for all v, whereas $\mathsf{shift\text{-}ind}_{\mathsf{sll}}(9, j, 10) = 0$ for all j because $\mathsf{sll}(v, 10)_9 = 0$ for all v. We write $\widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(I, J, S)$ for the corresponding multilinear extension, an 18-variate multilinear with $\mathbb{F}_2$-coefficients.

The defining identity of the shift indicator polynomial is, for each op,

$$\widetilde{\mathsf{shift}}_{\text{op}}(\widetilde{w})(I, Y, S) = \sum_{j \in \mathcal{B}_6} \widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(I, j, S) \cdot \widetilde{w}(j, Y). \tag{1}$$

To verify (1), it suffices to check it on the cube: fix $i, s \in \mathcal{B}_6$ and $y \in \mathcal{B}_{\ell_{\text{words}}}$. Either some unique $j^* \in \mathcal{B}_6$ has $\mathsf{shift\text{-}ind}_{\text{op}}(i, j^*, s) = 1$, in which case the right-hand side reduces to $\widetilde{w}(j^*, y) = w[y]_{j^*} = \text{op}(w[y], s)_i$; or no such j^* exists, in which case $\text{op}(w[y], s)_i = 0$ by definition of op on zero-fill bits, and the right-hand side is zero.

Identity (1) enables the standard reduction: given an evaluation claim on $\widetilde{\mathsf{shift}}_{\text{op}}(\widetilde{w})$ at some point $(r_i, r_y, r_s) \in K^{6+\ell_{\text{words}}+6}$, a 6-round sumcheck binds the index j, leaving evaluation claims on $\widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(r_i, r_j, r_s)$ (which the verifier evaluates directly) and on $\widetilde{w}(r_j, r_y)$ (a single query to the witness oracle).

Succinctness. The shift indicator $\widetilde{\mathsf{shift\text{-}ind}}_{\mathrm{op}}$ is *succinct* for each of the eight $\mathrm{op} \in \mathsf{ShiftOps}$: any point of K^{18} can be evaluated in $O(1)$ field operations (concretely, at most 84 multiplications in K plus a 6-way product). The explicit arithmetizations, with auxiliary functions $\mathsf{s}_k, \mathsf{s}'_k, \mathsf{a}_k$ for the recursive constructions of the 64-bit and 32-bit-parallel variants, are deferred to Appendix B.

4.3 The BitAnd Reduction

The BitAnd reduction transforms the satisfaction of the n_{and} BitAnd constraints (§3.4) into evaluation claims on the oblong-multilinearizations of the BitAnd constraint arrays. It is a univariate-skip variant of the sumcheck-based zerocheck [Gru24], in which the first sumcheck round is replaced by a univariate polynomial send. To minimize prover work, we also use deterministic challenges sampled from $\mathbb{F}_{2^8}$ in the first three sumcheck rounds, adapting an idea of Dao and Thaler [DT24].

Recall that satisfaction of the BitAnd constraints is the claim that, for each $x \in \mathcal{B}_{\ell_{\mathrm{and}}}$, $a[x] \wedge b[x] = c[x]$, where the 64-bit constraint arrays a, b, c are defined in §3.4 and the bitwise AND is just per-bit field multiplication. Equivalently, the polynomial

$$f(I, X) \coloneqq \widetilde{a}(I, X) \cdot \widetilde{b}(I, X) - \widetilde{c}(I, X)$$

on $\mathcal{B}_6 \times \mathcal{B}_{\ell_{\mathrm{and}}}$ vanishes identically. We work with the oblong-multilinearizations of the constraint arrays: writing $\widehat{a}(\widehat{I}, X), \widehat{b}(\widehat{I}, X), \widehat{c}(\widehat{I}, X)$ for the oblong-multilinearizations (§2.2.1) with $k = 6$, so each is of degree < 64 in $\widehat{I}$ and multilinear in $X_0, \ldots, X_{\ell_{\mathrm{and}}-1}$, the constraint reduces to the vanishing of

$$\widehat{f}(\widehat{I}, X) \coloneqq \widehat{a}(\widehat{I}, X) \cdot \widehat{b}(\widehat{I}, X) - \widehat{c}(\widehat{I}, X)$$

on $D \times \mathcal{B}_{\ell_{\mathrm{and}}}$.

4.3.1 The Rijndael Field and Deterministic Zerocheck

For the prover algorithm to run mostly over a small field, we fix the Rijndael field $\mathbb{F}_{2^8} \cong \mathbb{F}_2[X]/(X^8 + X^4 + X^3 + X + 1)$ (the AES base field) and embed it into K. We assume $D \subset \mathbb{F}_{2^8}$ (a 6-dimensional $\mathbb{F}_2$-subspace of the 8-dimensional $\mathbb{F}_2$-space $\mathbb{F}_{2^8}$); using the fixed embedding $\iota\colon \mathbb{F}_{2^8} \hookrightarrow K$ from §2.1, this gives the chain $D \subset \mathbb{F}_{2^8} \subset K$.

Tensor basis. We select three elements $\rho_0, \rho_1, \rho_2 \in \mathbb{F}_{2^8}$ such that

$$\left(\widetilde{\mathsf{eq}}_3(\rho_0, \rho_1, \rho_2, u_0, u_1, u_2)\right)_{u \in \mathcal{B}_3}$$

is an $\mathbb{F}_2$-basis of $\mathbb{F}_{2^8}$. The choices $\rho_0 \coloneqq X$, $\rho_1 \coloneqq X^2$, and $\rho_2 \coloneqq X^4$ work: their subproducts are $1, X, X^2, \ldots, X^7$, which form an $\mathbb{F}_2$-basis of $\mathbb{F}_{2^8}$, so the tensor expansion $\mathsf{TensorExpand}_3(\rho_0, \rho_1, \rho_2)$ (§2) differs from this monomial list by an invertible $\mathbb{F}_2$-linear map. We write $\sigma_j \coloneqq \iota(\rho_j) \in K$ for $j = 0, 1, 2$; since ι is an $\mathbb{F}_2$-linear injection that commutes with $\widetilde{\mathsf{eq}}$ (which has $\mathbb{F}_2$-coefficients), $\left(\widetilde{\mathsf{eq}}_3(\sigma, u)\right)_{u \in \mathcal{B}_3}$ is also an $\mathbb{F}_2$-linearly independent list in K.

Deterministic zerocheck. The standard zerocheck on an $\mathbb{F}_2$-valued polynomial $t(X)$ on $\mathcal{B}_\ell$ that is claimed to vanish samples $r_x \leftarrow K^\ell$ and sumchecks $0 \overset{?}{=} \sum_{x \in \mathcal{B}_\ell} t(x) \cdot \widetilde{\mathsf{eq}}_\ell(r_x, x)$, with soundness error $\ell/|K|$. The *deterministic-zerocheck* variant samples only $\ell - 3$ challenges and fixes the first three to the deterministic constants $\sigma_0, \sigma_1, \sigma_2$:

$$r_x \coloneqq (\sigma_0, \sigma_1, \sigma_2, \overline{r}_{x,0}, \ldots, \overline{r}_{x,\ell-4}), \qquad \overline{r}_x \leftarrow K^{\ell-3}.$$

A direct calculation gives

$$\sum_{x \in \mathcal{B}_\ell} t(x) \cdot \widetilde{\mathsf{eq}}_\ell(r_x, x) = \sum_{v \in \mathcal{B}_{\ell-3}} \overline{t}(v) \cdot \widetilde{\mathsf{eq}}_{\ell-3}(\overline{r}_x, v),$$

where

$$\overline{t}(X_0, \ldots, X_{\ell-4}) \coloneqq \sum_{u \in \mathcal{B}_3} t(u_0, u_1, u_2, X_0, \ldots, X_{\ell-4}) \cdot \widetilde{\mathsf{eq}}_3(\sigma, u).$$

Since t is $\mathbb{F}_2$-valued on $\mathcal{B}_\ell$ and $\left(\widetilde{\mathsf{eq}}_3(\sigma, u)\right)_{u \in \mathcal{B}_3}$ is $\mathbb{F}_2$-linearly independent, t vanishes on $\mathcal{B}_\ell$ iff $\overline{t}$ vanishes on $\mathcal{B}_{\ell-3}$, and the standard zerocheck soundness applied to $\overline{t}$ gives soundness error $(\ell - 3)/|K|$.

4.3.2 The Univariate-Skip Protocol

The BitAnd reduction runs a deterministic-zerocheck on $\widehat{f}(\widehat{I}, X)$ over $D \times \mathcal{B}_{\ell_{\text{and}}}$, but with the inner sum over $\widehat{i} \in D$ handled by a univariate polynomial send (Gruen [Gru24, §5.1]) rather than additional sumcheck rounds:

1. The verifier samples $\overline{r}_x \leftarrow K^{\ell_{\text{and}}-3}$ and sends it to the prover. Both parties set $r_x \coloneqq (\sigma_0, \sigma_1, \sigma_2, \overline{r}_{x,0}, \ldots, \overline{r}_{x,\ell_{\text{and}}-4})$.
2. The prover sends the univariate polynomial
$$g(\widehat{I}) \coloneqq \sum_{x \in \mathcal{B}_{\ell_{\text{and}}}} \widehat{f}(\widehat{I}, x) \cdot \widetilde{\mathsf{eq}}_{\ell_{\text{and}}}(r_x, x) \in K[\widehat{I}],$$
by sending its evaluations on 64 points outside D. The polynomial has degree at most $2 \cdot (2^6 - 1) = 126$, and these 64 values together with the 64 prescribed zeros on D determine g uniquely.
3. The verifier checks that g vanishes on D (using the prescribed zeros). It then samples $r_{\widehat{i}} \leftarrow K$ and sends it to the prover.
4. The parties run an ℓ_{and}-round sumcheck on the claim
$$g(r_{\widehat{i}}) \overset{?}{=} \sum_{x \in \mathcal{B}_{\ell_{\text{and}}}} \widehat{f}(r_{\widehat{i}}, x) \cdot \widetilde{\mathsf{eq}}_{\ell_{\text{and}}}(r_x, x).$$
At the end, the sumcheck produces a challenge $r'_x \in K^{\ell_{\text{and}}}$ and a target s, with the assertion that $s \overset{?}{=} \widehat{f}(r_{\widehat{i}}, r'_x) \cdot \widetilde{\mathsf{eq}}_{\ell_{\text{and}}}(r_x, r'_x)$.
5. The prover sends scalars $\alpha_a, \alpha_b, \alpha_c \in K$, claimed to equal $\widehat{a}(r_{\widehat{i}}, r'_x), \widehat{b}(r_{\widehat{i}}, r'_x), \widehat{c}(r_{\widehat{i}}, r'_x)$ respectively. The verifier evaluates $\widetilde{\mathsf{eq}}_{\ell_{\text{and}}}(r_x, r'_x)$ on its own and checks
$$s \overset{?}{=} \big(\alpha_a \cdot \alpha_b - \alpha_c\big) \cdot \widetilde{\mathsf{eq}}_{\ell_{\text{and}}}(r_x, r'_x).$$

The protocol's output is the triple of evaluation claims $\big(\widehat{a}(r_{\widehat{i}}, r'_x) = \alpha_a, \widehat{b}(r_{\widehat{i}}, r'_x) = \alpha_b, \widehat{c}(r_{\widehat{i}}, r'_x) = \alpha_c\big)$, to be discharged by the shift reduction (§4.5).

Soundness. A dishonest prover violating $a[x] \wedge b[x] = c[x]$ on some constraint induces a nonzero $\widehat{f}$ on $D \times \mathcal{B}_{\ell_{\text{and}}}$. By the deterministic-zerocheck analysis of §4.3.1, there exists $\widehat{i}^* \in D$ such that $\overline{g}(\widehat{i}^*) \neq 0$ except with probability $(\ell_{\text{and}} - 3)/|K|$ over $\overline{r}_x$, where $\overline{g}(\widehat{I}) \coloneqq \sum_x \widehat{f}(\widehat{I}, x) \cdot \widetilde{\mathsf{eq}}_{\ell_{\text{and}}}(r_x, x)$ is the polynomial the honest prover would send. Conditioned on this, the verifier's check that g vanishes on D forces $g \neq \overline{g}$ as polynomials, so by univariate Schwartz–Zippel, $g(r_{\widehat{i}}) = \overline{g}(r_{\widehat{i}})$ except with probability $2 \cdot (2^6 - 1)/|K|$ over $r_{\widehat{i}}$. The sumcheck on the (now-wrong) claim adds another $2\ell_{\text{and}}/|K|$ (round polynomials have individual degree ≤ 2). Soundness of the final triple-claim follows by univariate Schwartz–Zippel on $\alpha_a, \alpha_b, \alpha_c$, contributing $O(1)/|K|$. Total soundness error is $O(\ell_{\text{and}})/|K|$.

4.3.3 Prover Algorithm

The dominant cost is the univariate send: the prover must evaluate $g(\widehat{I})$ at 64 points outside D. We sketch the algorithm, called the *Rijndael zerocheck*, which performs most of the inner work entirely in $\mathbb{F}_{2^8}$ and embeds into K only once every 8 cube points. We refine the parameter D: we fix a 7-dimensional $\mathbb{F}_2$-subspace $D' \supset D$ with $D' \subset \mathbb{F}_{2^8}$, so $|D' \setminus D| = 64$. The prover evaluates g on $D' \setminus D$.

Lookup-based extrapolation. The core primitive is a precomputed map $\mathsf{Extrap}\colon \mathbb{F}_2^{64} \to \mathbb{F}_{2^8}^{64}$ that, on input a bit-vector $p = (p_0, \ldots, p_{63})$ viewed as the values on D of a univariate polynomial $P \in K[\widehat{I}]$ of degree < 64, returns $\big(\iota^{-1}(P(\widehat{i}))\big)_{\widehat{i} \in D' \setminus D}$ (which lies in $\mathbb{F}_{2^8}^{64}$ because $D' \subset \mathbb{F}_{2^8}$ and P is $\mathbb{F}_2$-valued on D). Since Extrap is $\mathbb{F}_2$-linear, it can be implemented by splitting the 64-bit input into eight 8-bit chunks, precomputing for each chunk a length-256 table of Extrap outputs on inputs nonzero on that chunk alone, and combining via 8 table lookups and 7 length-64 byte-wise XORs per invocation, following the lookup-table extrapolation technique of Hu et al. [Hu+25]. Total table size is $8 \cdot 256 \cdot 64$ bytes $= 128$ KiB.

Inner-outer structure. The prover tensor-expands $\overline{r}_x$ once into the table $\mathsf{TensorExpand}_{\ell_{\text{and}}-3}(\overline{r}_x) = \big(\widetilde{\mathsf{eq}}_{\ell_{\text{and}}-3}(\overline{r}_x, v)\big)_{v\in\mathcal{B}_{\ell_{\text{and}}-3}}$. Then it iterates over outer index $v \in \mathcal{B}_{\ell_{\text{and}}-3}$, and for each v accumulates a length-64 vector temp in $\mathbb{F}_{2^8}$ by iterating $u \in \mathcal{B}_3$ and computing, for $x := u \,\|\, v$ and each $\widehat{i} \in D' \setminus D$,

$$\mathsf{temp}[\widehat{i}] \mathrel{+}= \big(\mathsf{Extrap}(a[x])[\widehat{i}] \cdot \mathsf{Extrap}(b[x])[\widehat{i}] - \mathsf{Extrap}(c[x])[\widehat{i}]\big) \cdot \widetilde{\mathsf{eq}}_3(\rho, u).$$

At the end of the inner loop, the entries of temp are inverse-embeddings of $\sum_{u\in\mathcal{B}_3} \widehat{f}(\widehat{i}, u \,\|\, v) \cdot \widetilde{\mathsf{eq}}_3(\sigma, u)$. The prover then embeds via ι, scales by $\widetilde{\mathsf{eq}}_{\ell_{\text{and}}-3}(\overline{r}_x, v)$, and accumulates into a length-64 outer accumulator over K. The final accumulator contains $\big(g(\widehat{i})\big)_{\widehat{i}\in D'\setminus D}$. The asymptotic cost is dominated by the inner $\mathbb{F}_{2^8}$ work, with 64 $\mathbb{F}_{2^8} \to K$ embeddings and K-multiplications per 8 cube points. Further speedups are available via lookup-table-based additive-NTT techniques.

Preparing the sumcheck. For the sumcheck in step 4 of §4.3.2, the prover needs the value tables of $\widehat{a}(r_{\widehat{i}}, X), \widehat{b}(r_{\widehat{i}}, X), \widehat{c}(r_{\widehat{i}}, X)$ on $\mathcal{B}_{\ell_{\text{and}}}$. By the partial-specialization formula of §2.2.1, each table is obtained by precomputing the 64 Lagrange weights $\big(\delta_D(r_{\widehat{i}}, \widehat{i})\big)_{i\in\mathcal{B}_6}$ once and then, for each constraint $x \in \mathcal{B}_{\ell_{\text{and}}}$, taking the $\mathbb{F}_2$-coefficient subset sum of these weights with bits drawn from $a[x]$, $b[x]$, $c[x]$ respectively.

4.4 The IntMul Reduction

The IntMul reduction transforms the satisfaction of the n_{mul} IntMul constraints (§3.5) into four evaluation claims on the oblong-multilinearizations $\widehat{a}, \widehat{b}, \widehat{c}_{\text{lo}}, \widehat{c}_{\text{hi}}$ of the IntMul constraint arrays, all at a single point in $K^{1+\ell_{\text{mul}}}$. Unlike the BitAnd reduction, which exploits that bitwise AND is just per-bit field multiplication, IntMul concerns an integer-arithmetic relation that has no direct expression in K; we instead encode it as a multiplicative-group identity in K^* and reduce that identity via a tree of small-degree sumchecks.

4.4.1 Multiplying in the Exponent

Fix a generator $g \in K^*$, so g has order $2^\lambda - 1$. For each $x \in \mathcal{B}_{\ell_{\text{mul}}}$, the integer relation $a[x] \cdot b[x] = 2^{64} \cdot c_{\text{hi}}[x] + c_{\text{lo}}[x]$ can be converted into the multiplicative-group identity

$$\big(g^{a[x]}\big)^{b[x]} \stackrel{?}{=} \big(g^{2^{64}}\big)^{c_{\text{hi}}[x]} \cdot g^{c_{\text{lo}}[x]}, \tag{2}$$

both sides taken in K^*. By the basic property of cyclic groups, (2) holds iff the integer relation holds modulo the order $2^\lambda - 1$.

Wraparound. For $\lambda \geq 128$, the integer products in question are bounded:

$$a[x] \cdot b[x] \leq (2^{64}-1)^2 < 2^{128} - 1, \qquad 2^{64} \cdot c_{\text{hi}}[x] + c_{\text{lo}}[x] \leq 2^{128} - 1,$$

so $a[x] \cdot b[x]$ never wraps modulo $2^\lambda - 1$ (assuming $\lambda \geq 128$), but the right-hand side can equal $2^{128} - 1$ exactly when $c_{\text{hi}}[x] = c_{\text{lo}}[x] = 2^{64} - 1$. A dishonest prover could exploit this by setting $a[x] \cdot b[x] = 0$ and $c_{\text{hi}}[x] = c_{\text{lo}}[x] = 2^{64} - 1$, since both sides of (2) would equal $1 \in K^*$ even though the integer relation fails. In this case, $a[x] \cdot b[x]$ is even and $2^{64} \cdot c_{\text{hi}}[x] + c_{\text{lo}}[x]$ is odd, so the spurious case is ruled out by additionally requiring

$$a[x] \cdot b[x] \equiv c_{\text{lo}}[x] \pmod 2,$$

a condition on a single bit per constraint. This is expressible by adding to the constraint system, for each IntMul constraint, three auxiliary witness words a', b', c'_{lo} together with four BitAnd constraints enforcing $a = a'$, $b = b'$, $c_{\text{lo}} = c'_{\text{lo}}$, and

$$\mathsf{sll}(a', 63) \wedge \mathsf{sll}(b', 63) = \mathsf{sll}(c'_{\text{lo}}, 63).$$

The IntMul reduction below assumes these auxiliary constraints are part of the constraint system. (We henceforth assume $\lambda \geq 128$ so that no wraparound occurs on either side.)

4.4.2 Exponentiating Multilinears via GKR

Identity (2) is one of several exponentiations we need to verify. We isolate the generic problem: given a multilinear $\widetilde{V}(X) \in K[X_0, \ldots, X_{\ell_{\text{mul}}-1}]^{\preceq 1}$ and an $\mathbb{F}_2$-valued oblong-multilinear $\widehat{z}(\widehat{I}, X)$ (with $k = 6$) whose values on $D \times \mathcal{B}_{\ell_{\text{mul}}}$ form a constraint array, define the *exponentiated multilinear* $\widetilde{W}(X)$ by its values on the cube

$$W(x) \coloneqq \widetilde{V}(x)^{z[x]}, \quad z[x] \coloneqq \sum_{i=0}^{63} 2^i \cdot \widehat{z}(\widehat{i}, x), \quad x \in \mathcal{B}_{\ell_{\text{mul}}}.$$

The goal is to reduce an evaluation claim $s^{(0)} \stackrel{?}{=} \widetilde{W}(r^{(0)})$ to evaluation claims on $\widetilde{V}$ and $\widehat{z}$.

Product decomposition. Writing $z_i(x) \coloneqq \widehat{z}(\widehat{i}, x) \in \{0, 1\}$, on the cube

$$W(x) = \prod_{i=0}^{63} \big(\widetilde{V}(x)^{2^i}\big)^{z_i(x)} = \prod_{i=0}^{63} W_i(x), \quad W_i(x) \coloneqq \big(\widetilde{V}(x)^{2^i}\big)^{z_i(x)}.$$

Let $\widetilde{W_i}(X)$ denote the multilinear extension of W_i.

The GKR step. The product is balanced into a binary tree of depth 6 with the $\widetilde{W_i}$ at the leaves; each internal node at layer $k \in \{0, \ldots, 5\}$ is the multilinear extension of the pointwise product of its two children at layer $k+1$, so the root at layer 0 equals $\widetilde{W}$. Starting from the claim $s^{(0)} \stackrel{?}{=} \widetilde{W}^{(0)}(r^{(0)})$, the parties run a batched-sumcheck GKR loop for $k = 0, \ldots, 5$:

1. Verifier samples batching coefficients $\beta_i^{(k)} \leftarrow K$ for $i \in \{0, \ldots, 2^k - 1\}$.
2. Parties run an ℓ_{mul}-round sumcheck on

$$\sum_{i=0}^{2^k-1} \beta_i^{(k)} \cdot s_i^{(k)} \stackrel{?}{=} \sum_{x \in \mathcal{B}_{\ell_{\text{mul}}}} \widetilde{\mathsf{eq}}_{\ell_{\text{mul}}}(r^{(k)}, x) \cdot \sum_{i=0}^{2^k-1} \beta_i^{(k)} \cdot \widetilde{W}_{2i}^{(k+1)}(x) \cdot \widetilde{W}_{2i+1}^{(k+1)}(x),$$

   whose composing polynomials have individual degree ≤ 2 (excluding the equality indicator).
3. The sumcheck yields a challenge $r^{(k+1)} \in K^{\ell_{\text{mul}}}$. The prover sends claimed evaluations $s_i^{(k+1)} \coloneqq \widetilde{W}_i^{(k+1)}(r^{(k+1)})$ for $i \in \{0, \ldots, 2^{k+1} - 1\}$, and the verifier checks the resulting algebraic identity.

After 6 rounds, the parties have 64 evaluation claims $s_i^{(6)} \stackrel{?}{=} \widetilde{W}_i(r^{(6)})$, each at the common point $r^{(6)} \in K^{\ell_{\text{mul}}}$. Soundness error is the standard GKR sum over layers, $O(\ell_{\text{mul}} \cdot 2^6)/|K|$.

4.4.3 Twisting via Frobenius

The remaining task is to reduce the 64 claims $s_i^{(6)} \stackrel{?}{=} \widetilde{W}_i(r^{(6)})$ to a single claim on $\widetilde{V}$ and a single claim on $\widehat{z}$. Set $r_x \coloneqq r^{(6)}$ and unfold the definition of $\widetilde{W}_i$:

$$\widetilde{W}_i(X) = \sum_{x \in \mathcal{B}_{\ell_{\text{mul}}}} \widetilde{\mathsf{eq}}_{\ell_{\text{mul}}}(X, x) \cdot \big(1 + z_i(x) \cdot (\widetilde{V}(x)^{2^i} - 1)\big).$$

The exponent 2^i in the right-hand side makes the polynomial degree blow up. We sidestep this using the Frobenius endomorphism $\varphi(\alpha) = \alpha^2$ (§2.1). Applying φ^{-i} (well-defined because φ is invertible on K, with inverse $\varphi^{\lambda-1}$) to both sides:

$$\varphi^{-i}(s_i^{(6)}) \stackrel{?}{=} \sum_{x \in \mathcal{B}_{\ell_{\text{mul}}}} \widetilde{\mathsf{eq}}_{\ell_{\text{mul}}}\big(\varphi^{-i}(r_x), x\big) \cdot \big(1 + z_i(x) \cdot (\widetilde{V}(x) - 1)\big),$$

using that φ^{-i} is a ring automorphism that fixes the $\mathbb{F}_2$-coefficients of $\widetilde{\mathsf{eq}}$ and of z_i, and that $\varphi^{-i}(\widetilde{V}(x)^{2^i}) = \widetilde{V}(x)$.

The right-hand side is now of degree ≤ 2 in $\widetilde{V}, z_i$. The verifier samples batching coefficients $\gamma_i \leftarrow K$ for $i \in \{0, \ldots, 63\}$ and the parties run a batched ℓ_{mul}-round sumcheck:

$$\sum_{i=0}^{63} \gamma_i \cdot \varphi^{-i}(s_i^{(6)}) \stackrel{?}{=} \sum_{x \in \mathcal{B}_{\ell_{\text{mul}}}} \sum_{i=0}^{63} \gamma_i \cdot \widetilde{\mathsf{eq}}_{\ell_{\text{mul}}}\big(\varphi^{-i}(r_x), x\big) \cdot \big(1 + z_i(x) \cdot (\widetilde{V}(x) - 1)\big).$$

At the end, the parties hold a final sumcheck challenge $r'_x \in K^{\ell_{\text{mul}}}$ and claimed evaluations $\alpha_V \stackrel{?}{=} \widetilde{V}(r'_x)$ and $\alpha_{z,i} \stackrel{?}{=} \widetilde{z}_i(r'_x)$ for $i \in \{0, \ldots, 63\}$. The verifier evaluates each $\widetilde{\mathsf{eq}}_{\ell_{\text{mul}}}(\varphi^{-i}(r_x), r'_x)$ on its own, costing at most λ squarings per i to compute $\varphi^{-i}(r_x)$, and closes out the batched sumcheck.

Oblong-multilinearization. The 64 claims on $\widetilde{z}_i(r'_x)$ are reduced to a single claim on $\widehat{z}$ by a Lagrange-extrapolation argument. The verifier samples $r_{\widehat{i}} \leftarrow K$ and forms

$$\alpha_z := \sum_{i=0}^{63} \alpha_{z,i} \cdot \delta_D(r_{\widehat{i}}, \widehat{i}),$$

where δ_D is the bivariate equality-indicator on D from §2.2.1. By oblong-multilinearity, $\widehat{z}(\widehat{I}, r'_x) = \sum_{i=0}^{63} \delta_D(\widehat{I}, \widehat{i}) \cdot \widetilde{z}_i(r'_x)$ as univariate polynomials in $\widehat{I}$; specializing at $r_{\widehat{i}}$ gives the reduced claim

$$\alpha_z \stackrel{?}{=} \widehat{z}(r_{\widehat{i}}, r'_x).$$

If any individual α_{z,i^*} disagrees with $\widehat{z}(\widehat{i^*}, r'_x)$, then the two degree-63 polynomials in $\widehat{I}$ disagree, and univariate Schwartz–Zippel bounds the probability of accidental agreement by $64/|K|$.

4.4.4 Combined Protocol

The IntMul reduction orchestrates four exponentiations to verify identity (2) for all x simultaneously, with all four final oblong-multilinear claims pertaining to the *same* point. Let $G(X), G_{64}(X)$ be the constant multilinears equal to g and $g^{2^{64}}$, respectively. The four exponentiations are:

- base G, exponent $\widehat{a}$, output $\widetilde{A}(X) := G(X)^{a[X]} = g^{a[X]}$;
- base $\widetilde{A}$, exponent $\widehat{b}$, output $\widetilde{B}(X) := \widetilde{A}(X)^{b[X]} = g^{a[X] \cdot b[X]}$;
- base G, exponent $\widehat{c}_{\text{lo}}$, output $\widetilde{C}_{\text{lo}}(X) := g^{c_{\text{lo}}[X]}$;
- base G_{64}, exponent $\widehat{c}_{\text{hi}}$, output $\widetilde{C}_{\text{hi}}(X) := g^{2^{64} \cdot c_{\text{hi}}[X]}$.

Identity (2) then says $\widetilde{B}(x) = \widetilde{C}_{\text{hi}}(x) \cdot \widetilde{C}_{\text{lo}}(x)$ for each $x \in \mathcal{B}_{\ell_{\text{mul}}}$.

The protocol verifies this pointwise identity and unrolls it carefully so that all final oblong-claims end up at the same evaluation point:

1. Verifier samples $r \leftarrow K^{\ell_{\text{mul}}}$ and sends it to the prover. The prover sends $s \in K$ claimed to equal both $\widetilde{B}(r)$ and $\sum_{x \in \mathcal{B}_{\ell_{\text{mul}}}} \widetilde{\mathsf{eq}}_{\ell_{\text{mul}}}(r, x) \cdot \widetilde{C}_{\text{hi}}(x) \cdot \widetilde{C}_{\text{lo}}(x)$.

2. The parties run *only the GKR step* of the exponentiation protocol on $s \stackrel{?}{=} \widetilde{B}(r)$. This reduces the claim to 64 intermediate claims $s'_{B,i} \stackrel{?}{=} \widetilde{B}_i(r')$ at a common point $r' \in K^{\ell_{\text{mul}}}$.

3. The parties batch the following two ℓ_{mul}-round sumchecks:

   - The Frobenius step (§4.4.3) for the 64 claims $s'_{B,i} \stackrel{?}{=} \widetilde{B}_i(r')$, which reduces to claims $s_{\widetilde{A}} \stackrel{?}{=} \widetilde{A}(r'')$ and $s_{b,i} \stackrel{?}{=} \widetilde{b}_i(r'')$, plus the constant base contributions.
   - The deferred sumcheck $s \stackrel{?}{=} \sum_x \widetilde{\mathsf{eq}}_{\ell_{\text{mul}}}(r, x) \cdot \widetilde{C}_{\text{hi}}(x) \cdot \widetilde{C}_{\text{lo}}(x)$, which reduces to claims $s_{\widetilde{C}_{\text{hi}}} \stackrel{?}{=} \widetilde{C}_{\text{hi}}(r'')$ and $s_{\widetilde{C}_{\text{lo}}} \stackrel{?}{=} \widetilde{C}_{\text{lo}}(r'')$.

   All five output claims pertain to the same point $r'' \in K^{\ell_{\text{mul}}}$.

4. The parties run, in batched fashion, the GKR steps for the three remaining root claims $s_{\widetilde{A}} \stackrel{?}{=} \widetilde{A}(r'')$, $s_{\widetilde{C}_{\mathrm{hi}}} \stackrel{?}{=} \widetilde{C}_{\mathrm{hi}}(r'')$, $s_{\widetilde{C}_{\mathrm{lo}}} \stackrel{?}{=} \widetilde{C}_{\mathrm{lo}}(r'')$. At the last GKR layer the parties splice in a rerandomization sumcheck on the previously-obtained $s_{b,i} \stackrel{?}{=} \widetilde{b}_i(r'')$, producing all bit-level claims at a common point $r_x \in K^{\ell_{\mathrm{mul}}}$.

5. The three remaining root claims involve fixed bases $(g, g^{2^{64}}, g)$, so their Frobenius steps simplify: each $\widetilde{B}_i$-style decomposition reduces, by inspection, to a linear function of $\widetilde{a}_i(r_x)$ (resp. $\widetilde{c}_{\mathrm{hi},i}(r_x), \widetilde{c}_{\mathrm{lo},i}(r_x)$). Concretely,

$$s'_{\widetilde{A},i} = 1 + (g^{2^i} - 1) \cdot \widetilde{a}_i(r_x),$$

giving $s_{a,i} \coloneqq \widetilde{a}_i(r_x)$ by local algebra, and similarly for $\widetilde{c}_{\mathrm{hi},i}, \widetilde{c}_{\mathrm{lo},i}$.

6. The parties run the oblong-multilinearization step (§4.4.3) *once* across all four bit-level vectors $(\widetilde{a}_i(r_x))_i$, $(\widetilde{b}_i(r_x))_i$, $(\widetilde{c}_{\mathrm{lo},i}(r_x))_i$, $(\widetilde{c}_{\mathrm{hi},i}(r_x))_i$, using a single verifier challenge $r_{\widehat{i}} \leftarrow K$. This yields four output evaluation claims at the common point $(r_{\widehat{i}}, r_x) \in K^{1+\ell_{\mathrm{mul}}}$:

$$\widehat{a}(r_{\widehat{i}}, r_x) = \alpha_a, \quad \widehat{b}(r_{\widehat{i}}, r_x) = \alpha_b, \quad \widehat{c}_{\mathrm{lo}}(r_{\widehat{i}}, r_x) = \alpha_{c_{\mathrm{lo}}}, \quad \widehat{c}_{\mathrm{hi}}(r_{\widehat{i}}, r_x) = \alpha_{c_{\mathrm{hi}}}.$$

The four output claims are passed to the shift reduction (§4.5). Total soundness error is $O(\ell_{\mathrm{mul}})/|K|$, with hidden constants absorbing the depth-6 GKR trees and the constant-size oblong-multilinearization steps.

4.4.5 Prover Algorithm

The dominant cost is preparing and maintaining tables of values for the various exponentiated multilinears at the relevant cube points; this involves substantial work in K. We highlight a few implementation considerations.

Computing the layer-6 tables. For each of the four exponentiations, the prover initializes the leaf tables $\big(\widetilde{W}_i(x)\big)_{x \in \mathcal{B}_{\ell_{\mathrm{mul}}}}$ for $i = 0, \ldots, 63$ from the values $\widetilde{V}(x)$ and the bits $z_i(x) = \widehat{z}(\widehat{i}, x)$. Since each $z_i(x) \in \{0, 1\}$, the entry $\widetilde{W}_i(x)$ is either 1 or $\widetilde{V}(x)^{2^i} = \varphi^i(\widetilde{V}(x))$. For the fixed-base exponentiations (with $\widetilde{V} \equiv g$ or $g^{2^{64}}$), the powers g^{2^i} can be precomputed once into a table indexed by i, and the leaves are obtained by a bit-controlled select. For the variable-base exponentiation in the second step (base $\widetilde{A}$, exponent $\widehat{b}$), the prover applies φ^i to each entry of the $\widetilde{A}$-table, which costs i squarings per entry.

Sumcheck round computation. Each layer of the GKR loop runs a batched sumcheck with composing polynomials of individual degree 2 (plus the equality indicator), over ℓ_{mul} rounds. Standard sumcheck-prover algorithms (e.g. Thaler [Tha22, §4.1]) apply directly. The Frobenius step's sumcheck has composing polynomials of degree ≤ 2 in $\widetilde{V}, z_i$ (plus the equality indicator), so the per-round prover work is also linear in the table size.

Frobenius application. Computing $\varphi^{-i}(r_x)$ for $i = 0, \ldots, 63$, needed by the verifier in the Frobenius step, costs at most $\lambda \cdot 64$ field squarings in total. The prover precomputes the same quantities once and reuses them across the batched sumcheck.

4.5 The Shift Reduction

The shift reduction consumes the evaluation claims produced by the BitAnd reduction (§4.3) and the IntMul reduction (§4.4) and transforms all of them into a single multilinear-evaluation claim on the witness $\widetilde{w}$. It is the largest single component of the LIOP, with a sumcheck whose underlying domain contains the witness; the prover algorithm therefore has to exploit the sparsity of the constraint system to run in time close to the witness size.

For exposition we describe the reduction for a single oblong-multilinearized constraint array $\widehat{z}(\widehat{I}, X)$ over $D \times \mathcal{B}_{\ell_{\mathrm{cons}}}$, with a single input evaluation claim $\alpha_z \stackrel{?}{=} \widehat{z}(r_{\widehat{i}}, r'_x)$. The protocol extends to multiple

constraint arrays at the same point by random batching, with the verifier sampling fresh batching coefficients before phase 1. In the full Binius64 LIOP, the seven claims from the BitAnd and IntMul reductions (three from BitAnd, four from IntMul) are coordinated to share a common $(r_{\widehat{i}}, r'_x)$ via a preliminary batching step (with constraint arrays padded to a common $\ell_{\text{cons}} := \max(\ell_{\text{and}}, \ell_{\text{mul}})$), and then a single shift reduction processes the batched claim.

4.5.1 Mathematizing Constraint Arrays

We first re-express the oblong-multilinearization $\widehat{z}(\widehat{I}, X)$ in terms of $\widetilde{w}$ and the shift indicators.

Constraint matrices. For each op $\in$ $\mathsf{ShiftOps}$, define the three-dimensional $\mathbb{F}_2$-valued matrix $Z_{\text{cons,op}}$ of shape $n_{\text{cons}} \times n_{\text{words}} \times 64$ by

$$Z_{\text{cons,op}}[x, y, s] := \mathbb{1}\big[(y, \text{op}, s) \in L^z_x\big],$$

where L^z_x is the list of shifted value indices generating $z[x]$. The constraint array can be re-expressed as

$$z[x] = \sum_{\text{op}} \sum_{s \in \mathcal{B}_6} \sum_{y \in \mathcal{B}_{\ell_{\text{words}}}} Z_{\text{cons,op}}[x, y, s] \cdot \mathsf{shift}_{\text{op}}(w)(y, s),$$

where the outer sum is over the eight elements of $\mathsf{ShiftOps}$. Writing $\widetilde{Z}_{\text{cons,op}}$ for the multilinear extension of $Z_{\text{cons,op}}$, taking multilinear extensions and unrolling the shift-indicator identity (1) gives

$$\widetilde{z}(I, X) = \sum_{\text{op}} \sum_{s \in \mathcal{B}_6} \sum_{y \in \mathcal{B}_{\ell_{\text{words}}}} \sum_{j \in \mathcal{B}_6} \widetilde{Z}_{\text{cons,op}}(X, y, s) \cdot \widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(I, j, s) \cdot \widetilde{w}(j, y).$$

Applying the oblong-multilinearization construction of §2.2.1 (introducing $\delta_D(\widehat{I}, \widehat{i})$ and summing $i \in \mathcal{B}_6$) yields the analogous expression for $\widehat{z}(\widehat{I}, X)$. Specializing at $(r_{\widehat{i}}, r'_x)$ produces the explicit form

$$\alpha_z \overset{?}{=} \sum_{j \in \mathcal{B}_6} \sum_{s \in \mathcal{B}_6} \sum_{\text{op}} \sum_{y \in \mathcal{B}_{\ell_{\text{words}}}} \sum_{i \in \mathcal{B}_6} \delta_D(r_{\widehat{i}}, \widehat{i}) \cdot \widetilde{Z}_{\text{cons,op}}(r'_x, y, s) \cdot \widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(i, j, s) \cdot \widetilde{w}(j, y). \tag{3}$$

This nested sum is over $\ell_{\text{words}} + 18$ variables; running sumcheck on it naively is prohibitive. The next subsubsection shows how to factor it into two manageable phases.

4.5.2 The Two-Phase Sumcheck

Factorization. In (3), no single inner factor depends on both i and y: $\delta_D(r_{\widehat{i}}, \widehat{i})$ and $\widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(i, j, s)$ contain i but not y, whereas $\widetilde{Z}_{\text{cons,op}}(r'_x, y, s)$ and $\widetilde{w}(j, y)$ contain y but not i. We can therefore factor the two innermost sums and obtain

$$\alpha_z \overset{?}{=} \sum_{j \in \mathcal{B}_6} \sum_{s \in \mathcal{B}_6} \sum_{\text{op}} h_{\text{op}}(j, s) \cdot g_{\text{op}}(j, s),$$

where

$$h_{\text{op}}(J, S) := \sum_{i \in \mathcal{B}_6} \delta_D(r_{\widehat{i}}, \widehat{i}) \cdot \widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(i, J, S),$$
$$g_{\text{op}}(J, S) := \sum_{y \in \mathcal{B}_{\ell_{\text{words}}}} \widetilde{Z}_{\text{cons,op}}(r'_x, y, S) \cdot \widetilde{w}(J, y).$$

Both h_{op} and g_{op} are multilinear in the 12 variables (J, S): for h_{op}, the variable factors $\widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}(i, J, S)$ depend on J, S multilinearly and $\delta_D(r_{\widehat{i}}, \widehat{i})$ is constant in (J, S); for g_{op}, $\widetilde{Z}_{\text{cons,op}}(r'_x, y, S)$ depends on S but not J while $\widetilde{w}(J, y)$ depends on J but not S, so each summand is multilinear in (J, S).

Phase 1: 12 rounds. The parties run a 12-round sumcheck on

$$\alpha_z \stackrel{?}{=} \sum_{(j,s)\in\mathcal{B}_{12}} \sum_{\text{op}} h_{\text{op}}(j,s)\cdot g_{\text{op}}(j,s).$$

The composing polynomials are degree-2 products of multilinears (one $h_{\text{op}}\cdot g_{\text{op}}$ product per op, summed over $|\mathsf{ShiftOps}| = 8$); the prover sends a degree-2 round polynomial in each round. After 12 rounds, the sumcheck reduces α_z to a claim

$$\beta \stackrel{?}{=} \sum_{\text{op}} h_{\text{op}}(r_j, r_s)\cdot g_{\text{op}}(r_j, r_s),$$

where $r_j, r_s \in K^6$ are the sumcheck challenges. The verifier can evaluate each $h_{\text{op}}(r_j, r_s)$ on its own using the succinct shift-indicator arithmetizations of §4.2, after a single $O(2^6)$ precomputation of $\big(\delta_D(r_{\widehat{i}}, \widehat{i})\big)_{i\in\mathcal{B}_6}$.

Phase 2: ℓ_{words} rounds. Unrolling the definition of $g_{\text{op}}(r_j, r_s)$ and exchanging sums,

$$\beta \stackrel{?}{=} \sum_{y\in\mathcal{B}_{\ell_{\text{words}}}} \Big[\sum_{\text{op}} h_{\text{op}}(r_j, r_s)\cdot \widetilde{Z}_{\text{cons,op}}(r'_x, y, r_s)\Big]\cdot \widetilde{w}(r_j, y).$$

The bracketed quantity is multilinear in y, as is $\widetilde{w}(r_j, y)$, so the parties run an ℓ_{words}-round sumcheck on this claim. The composing polynomials are degree-2 products of multilinears in y. After ℓ_{words} rounds, the sumcheck produces a challenge $r_y \in K^{\ell_{\text{words}}}$ and a final claim

$$\gamma \stackrel{?}{=} \Big[\sum_{\text{op}} h_{\text{op}}(r_j, r_s)\cdot \widetilde{Z}_{\text{cons,op}}(r'_x, r_y, r_s)\Big]\cdot \widetilde{w}(r_j, r_y).$$

The verifier evaluates the bracketed factor on its own (see below) and outputs the single witness-evaluation claim $\widetilde{w}(r_j, r_y)$. This bit-level claim on the $\mathbb{F}_2$-valued witness is reduced to a linear query on the packed K-valued oracle π in the ring-switching step of §4.6.

Verifier evaluation of constraint matrices. At the very end, the verifier needs $\widetilde{Z}_{\text{cons,op}}(r'_x, r_y, r_s)$ for each op $\in \mathsf{ShiftOps}$. The straightforward sparse-evaluation algorithm runs in $O(n_{\text{cons}})+O(\text{circuit size})$ time by iterating over the constraint lists; succinct verification is left for future work, following e.g. the Spark compiler of Setty [Set20].

4.5.3 Zero-Shift Optimization

A large fraction of shifted value indices in typical circuits use shift amount $s = 0$, which acts as the identity for every op $\in \mathsf{ShiftOps}$: $\mathsf{sll}(v, 0) = \mathsf{srl}(v, 0) = \cdots = v$. In this case, $\mathsf{shift\text{-}ind}_{\text{op}}(i, j, 0) = \mathbb{1}[i = j]$ for every op, and the shift-indicator factor in (3) collapses to $\widetilde{\mathsf{eq}}_6(I, J)$.

We split the phase-1 sum into $s = 0$ and $s \neq 0$ parts. For the $s = 0$ part, summing i against $\widetilde{\mathsf{eq}}_6(I, J)$ collapses the inner sum:

$$\sum_{i\in\mathcal{B}_6} \delta_D(r_{\widehat{i}}, \widehat{i})\cdot \widetilde{\mathsf{eq}}_6(i, j) = \delta_D(r_{\widehat{i}}, \widehat{j}),$$

so the entire $s = 0$ contribution simplifies to

$$\sum_{y\in\mathcal{B}_{\ell_{\text{words}}}} \widehat{w}(r_{\widehat{i}}, y)\cdot \Big(\sum_{\text{op}} \widetilde{Z}_{\text{cons,op}}(r'_x, y, 0)\Big),$$

where $\widehat{w}(\widehat{I}, Y)$ is the oblong-multilinearization of $\widetilde{w}$ (in the J-variable). The double summation over $i, j \in \mathcal{B}_6$ has collapsed to a single dimension, a 64-fold reduction. The modified protocol splits phase 1 into:

- an $s = 0$ branch: directly produce a claim on $\widehat{w}(r_{\widehat{i}}, r_y)$ via an ℓ_{words}-round sumcheck (no J, S binding needed),

- an $s \neq 0$ branch: run the original phase-1 sumcheck but only over the non-zero shift amounts.

The verifier then closes out with both a claim on $\widehat{w}(r_{\widehat{i}}, r_y)$ from the $s = 0$ branch and a claim on $\widetilde{w}(r_j, r_y)$ from the $s \neq 0$ branch. The two claims can be batched into a single query by sharing the same r_y across both sumchecks; the verifier evaluates the connection via the Lagrange-extrapolation identity $\widehat{w}(r_{\widehat{i}}, r_y) = \sum_{j \in \mathcal{B}_6} \delta_D(r_{\widehat{i}}, \widehat{j}) \cdot \widetilde{w}(j, r_y)$.

4.5.4 Prover Algorithm

We sketch the prover-side data structures and the main loops; the goal is to make the prover's work close to linear in the witness size despite the formal $\ell_{\text{words}} + 18$-dimensional sum.

Computing h_{op}. The prover precomputes $\big(\delta_D(r_{\widehat{i}}, \widehat{i})\big)_{i \in \mathcal{B}_6}$ once (§2.2.1) and then, for each op, populates the 64×64 table $h_{\text{op}}[j][s]$ exploiting the structure of $\widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}$. For srl, the indicator $\mathsf{shift\text{-}ind}_{\mathsf{srl}}(i, j, s)$ is $\mathbb{1}[j = i + s]$ on the cube, so $h_{\mathsf{srl}}[j][s] = \delta_D(r_{\widehat{i}}, \widehat{i})$ for $i = j - s$ when valid (and 0 otherwise); similar shape patterns apply to sll, sra, ror, and their 32-bit variants. For sra, the case $j = 63$ admits multiple contributing i values; these are accumulated in a running prefix sum over s. Total table size across all eight ops is $8 \cdot 2^{12}$ field elements.

Constraint-column index. Most shifted value indices in practice are sparse: each constraint list L_x^z has $O(1)$ entries on average. The prover builds, in a single linear pass over the constraint system, a sparse index data structure $\mathsf{index}\colon \mathcal{B}_{\ell_{\text{words}}} \to (\mathsf{ShiftOps} \times \mathcal{B}_6 \to K)$ such that

$$\mathsf{index}[y][(\text{op}, s)] = \widetilde{Z}_{\text{cons,op}}(r'_x, y, s)$$

exactly for those (y, op, s) triples for which the value is nonzero (otherwise the key is absent). The build is

> precompute $\mathsf{TensorExpand}_{\ell_{\text{cons}}}(r'_x)$; then for each $x \in \mathcal{B}_{\ell_{\text{cons}}}$ and each $(y, \text{op}, s) \in L_x^z$, add $\mathsf{TensorExpand}(r'_x)[x]$ to $\mathsf{index}[y][(\text{op}, s)]$.

Total work and space are $O(\text{circuit size}) + O(n_{\text{words}})$ field elements.

Computing g_{op}. Using index, the table $g_{\text{op}}[j][s] = \sum_y \widetilde{Z}_{\text{cons,op}}(r'_x, y, s) \cdot \widetilde{w}(j, y)$ is computed by iterating over y and, for each populated key (op, s) in $\mathsf{index}[y]$, adding $\mathsf{index}[y][(\text{op}, s)]$ into $g_{\text{op}}[j][s]$ for each bit j where $w[y]_j = 1$. Total work is proportional to the (sparse) circuit size.

Phase-2 multilinear tables. For phase 2 the prover needs the value tables of the multilinears $\widetilde{w}(r_j, Y)$ (a standard partial specialization of $\widetilde{w}$, computed via $\mathsf{TensorExpand}_6(r_j)$) and $\sum_{\text{op}} h_{\text{op}}(r_j, r_s) \cdot \widetilde{Z}_{\text{cons,op}}(r'_x, Y, r_s)$. The second again uses index: precompute $\mathsf{TensorExpand}_6(r_s)$, then accumulate, for each y and each populated $(\text{op}, s) \in \mathsf{index}[y]$, $h_{\text{op}}(r_j, r_s) \cdot \mathsf{index}[y][(\text{op}, s)] \cdot \widetilde{\mathsf{eq}}_6(r_s, s)$ into a length-n_{words} vector.

4.6 Ring-Switching to the Packed Witness

The shift reduction (§4.5) closes by establishing a single evaluation claim

$$\widetilde{w}(r_j, r_y) = t, \qquad r_j \in K^6, \quad r_y \in K^{\ell_{\text{words}}}, \quad t \in K,$$

on the multilinear extension of the $\mathbb{F}_2$-valued witness w. Our LIOP framework (§2.3) requires the prover oracle to be K-valued and the final witness-side check to be a single linear query against that oracle. Ring-switching, due to Diamond and Posen [DP24], bridges the two by packing the $\mathbb{F}_2$-valued witness into a K-valued oracle and reducing the bit-level evaluation claim to a linear query on the packed oracle. We summarize the construction at a level sufficient for the LIOP specification and defer protocol details and proofs to [DP24].

The packed witness. We view the witness as a length-$64 \cdot n_{\text{words}}$ bit-string indexed by $\mathcal{B}_{6+\ell_{\text{words}}}$. Set the packing parameter

$$\kappa := \log_2 \lambda,$$

the $\mathbb{F}_2$-dimension of K. Our padding assumption $\lambda \mid 64 \cdot n_{\text{words}}$ (§2) ensures $6 + \ell_{\text{words}} \geq \kappa$, so we can group the $6 + \ell_{\text{words}}$ Boolean dimensions of w into the first κ ("packed" bit dimensions, indexed by $i \in \mathcal{B}_\kappa$) and the remaining

$$\ell_{\text{pack}} := 6 + \ell_{\text{words}} - \kappa$$

("unpacked" dimensions, indexed by $y \in \mathcal{B}_{\ell_{\text{pack}}}$). Each "cell" indexed by y contains λ witness bits; with $\lambda = 128$ this packs two consecutive 64-bit witness words into a single K-element.

Fix once and for all an $\mathbb{F}_2$-basis $(\beta_i)_{i \in \mathcal{B}_\kappa}$ of K, indexed by $\mathcal{B}_\kappa$ for notational convenience. The *packing map* pack_λ takes the witness $w \colon \mathcal{B}_{6+\ell_{\text{words}}} \to \mathbb{F}_2$ to the function

$$\mathsf{pack}_\lambda(w) \colon \mathcal{B}_{\ell_{\text{pack}}} \to K, \qquad \mathsf{pack}_\lambda(w)(y) := \sum_{i \in \mathcal{B}_\kappa} w(i, y) \cdot \beta_i,$$

i.e., each entry of $\mathsf{pack}_\lambda(w)$ is the inner product of λ witness bits with the chosen basis. The LIOP commits to the oracle $\pi := \mathsf{pack}_\lambda(w)$, of length $2^{\ell_{\text{pack}}}$ over K.

Ring-switching reduction. The reduction follows Diamond and Posen [DP24, Construction 3.1]. We split the shift reduction's evaluation point $(r_j, r_y) \in K^{6+\ell_{\text{words}}}$ into a packed part $r_{\text{pack}} \in K^\kappa$ and an unpacked part $r_{\text{rest}} \in K^{\ell_{\text{pack}}}$, reinterpreting (r_j, r_y) under the packing layout above.

1. The prover sends the vector of partial evaluations

   $$\hat{s} := \big(\hat{s}_i\big)_{i \in \mathcal{B}_\kappa}, \qquad \hat{s}_i := \widetilde{w}(i, r_{\text{rest}}),$$

   one K-element per index $i \in \mathcal{B}_\kappa$.

2. The verifier checks consistency with the input claim t by verifying the inner product

   $$t \stackrel{?}{=} \sum_{i \in \mathcal{B}_\kappa} \widetilde{\mathsf{eq}}_\kappa(i, r_{\text{pack}}) \cdot \hat{s}_i.$$

3. The verifier samples $r' \leftarrow K^{\ell_{\text{pack}}}$.

4. The verifier specializes the *ring-switching indicator multilinear* of [DP24, Construction 3.1] at (r_{rest}, r') to obtain a multilinear $\mathsf{ring\text{-}switch\text{-}ind}_{r_{\text{rest}}, r'}(Y) \in K[Y_0, \ldots, Y_{\ell_{\text{pack}}-1}]^{\preceq 1}$, and derives a target value

   $$s' \in K$$

   as a fixed K-linear combination of the entries of $\hat{s}$ whose coefficients depend on r_{rest}, r', and the basis (β_i). The exact functional form of both $\mathsf{ring\text{-}switch\text{-}ind}_{r_{\text{rest}}, r'}$ and s' is given by [DP24, Construction 3.1]; the verifier evaluates both in $O(\ell_{\text{pack}} \cdot \lambda)$ operations.

The output of ring-switching is the pair $(\mathsf{ring\text{-}switch\text{-}ind}_{r_{\text{rest}}, r'}, s')$, which defines the linear oracle query on π:

$$\sum_{y \in \mathcal{B}_{\ell_{\text{pack}}}} \mathsf{ring\text{-}switch\text{-}ind}_{r_{\text{rest}}, r'}(y) \cdot \pi(y) \stackrel{?}{=} s'.$$

Soundness of this reduction is established in [DP24, §3]; the error is $O(\ell_{\text{pack}})/|K|$ over r'.

For notational uniformity with the rest of the document, we abbreviate this output query as $(t_{\text{rs}}, s_{\text{rs}}) := (\mathsf{ring\text{-}switch\text{-}ind}_{r_{\text{rest}}, r'}, s')$. It is one of the two linear queries on π produced by the LIOP; it is random-combined with the public-input-check query of §4.7 to yield the single final query of §2.3.

4.7 Public Input Check

The public input check verifies that the first $n_{\text{public}} \coloneqq n_{\text{const}} + n_{\text{inout}}$ words of w equal the verifier-held public data: $w[y] = C(y)$ for $y \in \{0, \dots, n_{\text{const}} - 1\}$ and $w[n_{\text{const}} + y] = u(y)$ for $y \in \{0, \dots, n_{\text{inout}} - 1\}$. We concatenate the constants and the statement into a single vector

$$\mathsf{public}\colon \mathcal{B}_6 \times \mathcal{B}_{\ell_{\text{public}}} \to \mathbb{F}_2, \qquad \mathsf{public} \coloneqq C \parallel u,$$

under the identification of $\mathcal{B}_{\ell_{\text{public}}}$ with $\{0, \dots, n_{\text{public}} - 1\}$. The padding requirement $\lambda \mid 64 \cdot n_{\text{public}}$ (§2) lets us apply the packing map pack_λ (§4.6) to public in the same way as to w, producing

$$\mathsf{pack}_\lambda(\mathsf{public})\colon \mathcal{B}_{\ell_{\text{public-pack}}} \to K, \qquad \ell_{\text{public-pack}} \coloneqq \ell_{\text{public}} - (\log_2 \lambda - 6).$$

The check the verifier wants to enforce is that π agrees with $\mathsf{pack}_\lambda(\mathsf{public})$ on the first $2^{\ell_{\text{public-pack}}}$ entries: for all $y \in \mathcal{B}_{\ell_{\text{public-pack}}}$,

$$\pi(y, 0, \dots, 0) \stackrel{?}{=} \mathsf{pack}_\lambda(\mathsf{public})(y),$$

where the index $(y, 0, \dots, 0) \in \mathcal{B}_{\ell_{\text{pack}}}$ pads y with $\ell_{\text{pack}} - \ell_{\text{public-pack}}$ zeros on the high coordinates. Equivalently, the polynomial

$$\widetilde{\pi}(Y_0, \dots, Y_{\ell_{\text{public-pack}}-1}, 0, \dots, 0) - \widetilde{\mathsf{pack}_\lambda(\mathsf{public})}(Y_0, \dots, Y_{\ell_{\text{public-pack}}-1})$$

vanishes on $\mathcal{B}_{\ell_{\text{public-pack}}}$.

Reduction to a linear query. The verifier samples $r_p \leftarrow K^{\ell_{\text{public-pack}}}$ and checks

$$\widetilde{\pi}(r_p, 0, \dots, 0) \stackrel{?}{=} \widetilde{\mathsf{pack}_\lambda(\mathsf{public})}(r_p).$$

The right-hand side is computable by the verifier on its own in $O(n_{\text{public}}/\lambda)$ time via the standard $\mathsf{TensorExpand}$-based multilinear-evaluation algorithm. The left-hand side is an evaluation of the oracle's multilinear extension, which is exactly the multilinear-evaluation linear query on π at the point $(r_p, 0, \dots, 0) \in K^{\ell_{\text{pack}}}$:

$$t_{\text{pub}}(Y) \coloneqq \widetilde{\mathsf{eq}}_{\ell_{\text{pack}}}\big((r_p, 0, \dots, 0), Y\big) = \widetilde{\mathsf{eq}}_{\ell_{\text{public-pack}}}(r_p, Y_{<\ell_{\text{public-pack}}}) \cdot \prod_{i=\ell_{\text{public-pack}}}^{\ell_{\text{pack}}-1} (1 - Y_i),$$

with target $s_{\text{pub}} \coloneqq \widetilde{\mathsf{pack}_\lambda(\mathsf{public})}(r_p)$. The check passes iff $\sum_{y \in \mathcal{B}_{\ell_{\text{pack}}}} t_{\text{pub}}(y) \cdot \pi(y) = s_{\text{pub}}$.

By the standard zerocheck soundness, the reduction has error $\ell_{\text{public-pack}}/|K|$ over the choice of r_p.

Batching with the ring-switching query. The LIOP could output the two queries $(t_{\text{rs}}, s_{\text{rs}})$ from ring-switching and $(t_{\text{pub}}, s_{\text{pub}})$ from the public input check independently. We instead random-combine them into a single linear query on π. The verifier samples a batching coefficient $\xi \leftarrow K$ and outputs

$$t(Y) \coloneqq t_{\text{rs}}(Y) + \xi \cdot t_{\text{pub}}(Y), \qquad s \coloneqq s_{\text{rs}} + \xi \cdot s_{\text{pub}},$$

so that the LIOP's final linear query is $\sum_{y \in \mathcal{B}_{\ell_{\text{pack}}}} t(y) \cdot \pi(y) = s$. Both t_{rs} and t_{pub} are succinct (evaluable in $O(\ell_{\text{pack}})$ field operations), so t is too. The random batching contributes an additional $1/|K|$ to the soundness error.

5 Basic SNARK Compiler

This section presents a generic compiler from any LIOP (§2.3) to a SNARK. The compiler is a composition of three standard transformations,

$$\text{LIOP} \xrightarrow{\text{BaseFold}} \text{IOP} \xrightarrow{\text{BCS}} \text{IP} \xrightarrow{\text{Fiat–Shamir}} \text{SNARK},$$

each of which is independent of the LIOP and applies uniformly to LIOPs with any number of oracles and any number of linear queries. Applying the compiler to the Binius64 LIOP of §4 yields the Binius64 SNARK (§5.5).

5.1 Compiling LIOPs to SNARKs

Stage 1: LIOP → IOP via BaseFold. An LIOP's verifier closes by issuing linear queries, each specified by a multilinear polynomial t_j and target s_j against an oracle π_{i_j} (§2.3). To turn this into a standard IOP, in which the verifier issues point queries, we encode each LIOP oracle $\pi_i \in K^{2^{n_i}}$ as a codeword of a Reed–Solomon code over K. Each linear query then becomes an additional linear constraint on the codeword: the codeword is required not only to be a valid Reed–Solomon codeword but also to satisfy $\sum_{w \in \mathcal{B}_{n_i}} t_j(w) \cdot \pi_{i_j}(w) = s_j$. The pair (Reed–Solomon code, linear constraint) is a *constrained Reed–Solomon code* in the sense of Arnon, Chiesa, Fenzi, and Yogev [ACFY25, §4.1]. BaseFold (§5.2) is an *interactive oracle proof of proximity* (IOPP) for constrained Reed–Solomon codes: given oracle access to a candidate codeword, BaseFold accepts iff the codeword is close to a true codeword of the constrained code, using a polylogarithmic number of point queries. Composing one BaseFold instance per LIOP linear query yields a public-coin IOP whose oracles are the original π_i together with the folded codewords introduced by BaseFold.

Stage 2: IOP → IP via BCS. The BCS transform of Ben-Sasson, Chiesa, and Spooner [BCS16] replaces each oracle message of an IOP with the Merkle-tree root of its content; subsequent point queries are answered by the prover with the corresponding leaf and Merkle authentication path. In the random oracle model, this transformation preserves soundness up to a tight multiplicative loss in the adversary's query budget, and the result is a public-coin interactive proof (no oracles) of the same statement.

Stage 3: IP → SNARK via Fiat–Shamir. The Fiat–Shamir transform makes the public-coin IP non-interactive by deriving each verifier challenge as a hash of the running transcript. The resulting non-interactive argument is a SNARK in the random oracle model.

Succinctness. The compiled SNARK is succinct (proof size and verification time polylogarithmic in the witness size) provided the source LIOP is succinct in the sense of §2.3: when each linear query polynomial t_j is evaluable in $\mathrm{poly}(n_{i_j})$ time, BaseFold's verifier uses polylogarithmically many point queries, the Merkle openings are polylog-sized, and the LIOP's own non-oracle messages are short.

5.2 BaseFold

BaseFold is an IOPP for constrained Reed–Solomon codes (Arnon, Chiesa, Fenzi, and Yogev [ACFY25, §4.1]). The original construction is due to Zeilberger, Chen, and Fisch [ZCF24]; we use the binary-field adaptation of Diamond and Posen [DP24], with encoding via the additive number-theoretic transform (NTT) of Lin, Chung, and Han [LCH14]. Tightened soundness analyses are given by Haböck [Hab24].

Setup. Fix a power-of-two rate parameter $\mathcal{R} \geq 2$ and a nested chain of additive $\mathbb{F}_2$-linear subspaces $\{0\} = E_0 \subset E_1 \subset \cdots \subset E_n \subset K$, where each E_k has dimension $k + \log_2 \mathcal{R}$.

The encoding map $\mathsf{Enc}_k \colon K^{2^k} \to K^{|E_k|}$ takes an oracle $\pi \colon \mathcal{B}_k \to K$ to its evaluation on E_k of the polynomial

$$f_\pi(X) \coloneqq \sum_{i=0}^{2^k-1} \pi_{\mathsf{rev}(i)} \cdot X_i(X),$$

where $X_0, \ldots, X_{2^k-1}$ is the novel polynomial basis underlying the Lin–Chung–Han additive NTT [LCH14] (the basis of monic polynomials with $\deg X_i = i$ that diagonalizes the additive-NTT butterfly tree), and $\mathsf{rev}(i)$ is the k-bit reversal of $i \in \{0, \ldots, 2^k - 1\}$. The bit-reversed indexing is a deliberate choice with two consequences. First, it aligns hypercube zero-padding on the message side with codeword duplication on the encoding side, which we use in §5.3 to handle oracles of unequal sizes. Second, it causes BaseFold's sumcheck to specialize high-indexed variables of π's multilinear extension first, which yields a more SIMD-friendly prover algorithm than specializing low-indexed variables first.

The constrained Reed–Solomon code at level n is parameterized by a multilinear *multiplicand* $t \colon \mathcal{B}_n \to K$ and a target $s \in K$; it is the set of codewords $c = \mathsf{Enc}_n(\pi)$ for $\pi \colon \mathcal{B}_n \to K$ satisfying

$$\pi \cdot t = s \qquad \text{i.e.,} \qquad \sum_{w \in \mathcal{B}_n} \pi(w) \cdot t(w) = s.$$

Protocol. On input an oracle $c_n \in K^{|E_n|}$ and a constraint (t, s), BaseFold proceeds in two phases:

1. *Folding phase.* For each round $k = n-1, \ldots, 0$, the parties interleave one round of a sumcheck on the constraint $\sum_{w \in \mathcal{B}_n} \pi(w) \cdot t(w) = s$ with a single fold of the codeword. The prover sends a sumcheck round polynomial; the verifier samples $\rho_k \leftarrow K$ and updates the running sumcheck claim; both parties fold the codeword from $c_{k+1} \in K^{|E_{k+1}|}$ to $c_k \in K^{|E_k|}$ using ρ_k. The prover provides c_k as a new oracle each round.

2. *Query phase.* After n rounds, the codeword $c_0 \in K^{|E_0|}$ is short (of length $\mathcal{R}$) and the sumcheck claim reduces to a single multilinear-extension evaluation that the verifier can check against the prover's claimed reduced codeword. The verifier additionally samples μ random query positions in E_n, and for each position requests the corresponding entries of $c_n, c_{n-1}, \ldots, c_0$ from the prover oracles. The verifier checks two things at each query position: that the entries are consistent with each successive fold given the challenges ρ_k, and that the chain bottoms out in the sent c_0.

The verifier accepts iff all sumcheck round-polynomial checks pass, the final reduced claim matches, and all μ fold-consistency checks pass.

Parameters. For target soundness $2^{-\lambda}$, the query count μ is chosen logarithmic in λ and in the inverse proximity gap of the underlying code; concrete settings are given in [Hab24]. The prover work is $O(2^n \cdot \mathcal{R})$ in K, dominated by the initial encoding; the verifier work is $\mathrm{poly}(n) + O(\mu \cdot n)$.

5.3 Lifted BaseFold

The Batched BaseFold protocol of §5.4 requires the oracles to share a common variable count $\mathbf{n}$. When the LIOP commits oracles of unequal sizes, we lift each smaller oracle's codeword to the level-$\mathbf{n}$ encoding via the *Lifted BaseFold* (a.k.a. Lifted FRI, after the Fast Reed–Solomon IOPP) construction [HMH].

We use three operators on K-vectors. For $v \in K^N$ with $N = 2^\mu$, and $\eta \geq 0$, define

$$\mathsf{ZeroPadLSB}_\eta(v) \in K^{N \cdot 2^\eta}, \qquad \mathsf{ZeroPadLSB}_\eta(v)_i = \begin{cases} v_{i/2^\eta} & \text{if } 2^\eta \mid i, \\ 0 & \text{otherwise,} \end{cases}$$

$$\mathsf{ZeroPadMSB}_\eta(v) \in K^{N \cdot 2^\eta}, \qquad \mathsf{ZeroPadMSB}_\eta(v)_i = \begin{cases} v_i & \text{if } i < N, \\ 0 & \text{otherwise,} \end{cases}$$

$$\mathsf{Duplicate}_\eta(v) \in K^{N \cdot 2^\eta}, \qquad \mathsf{Duplicate}_\eta(v)_i = v_{\lfloor i/2^\eta \rfloor}.$$

The two zero-padding operators differ in which index bits they zero: $\mathsf{ZeroPadLSB}_\eta$ keeps entries only at positions whose low η bits vanish, while $\mathsf{ZeroPadMSB}_\eta$ keeps entries only at the bottom N indices (positions whose high η bits vanish).

The Lin–Chung–Han additive NTT exhibits a clean zero-pad/duplicate correspondence on the LSB side:

Theorem 5.1 (LCH NTT zero-pad correspondence)**.** *For all $\mu, \eta \geq 0$ and any coefficient vector $a \in K^{2^\mu}$ with respect to the novel polynomial basis of §5.2,*

$$\mathit{NTT}_{\mu+\eta}\big(\mathit{ZeroPadLSB}_\eta(a)\big) = \mathit{Duplicate}_\eta\big(\mathit{NTT}_\mu(a)\big).$$

Now fix an oracle $\pi \colon \mathcal{B}_n \to K$ with $n \leq \mathbf{n}$, identified with its value vector in K^{2^n} when applying the operators. Its lift $\mathsf{ZeroPadMSB}_{\mathbf{n}-n}(\pi) \colon \mathcal{B}_{\mathbf{n}} \to K$ has multilinear extension

$$\widetilde{\mathsf{ZeroPadMSB}}_{\mathbf{n}-n}(\pi)(X_0, \ldots, X_{\mathbf{n}-1}) = \widetilde{\pi}(X_0, \ldots, X_{n-1}) \cdot \widetilde{\mathsf{eq}}_{\mathbf{n}-n}\big(0^{\mathbf{n}-n}, X_n, \ldots, X_{\mathbf{n}-1}\big), \tag{4}$$

where $\widetilde{\mathsf{eq}}_{\mathbf{n}-n}(0^{\mathbf{n}-n}, Y_0, \ldots, Y_{\mathbf{n}-n-1}) = \prod_j (1 - Y_j)$ vanishes off the all-zeros high block.

The bit-reversed indexing of §5.2 turns end-padding on the hypercube into LSB zero-padding on the coefficient side: the coefficient vector of $\mathsf{ZeroPadMSB}_{\mathbf{n}-n}(\pi)$ in the novel basis is exactly $\mathsf{ZeroPadLSB}_{\mathbf{n}-n}$ applied to the coefficient vector of π. Combining with Theorem 5.1, we obtain the central identity of Lifted BaseFold:

$$\mathsf{Enc}_{\mathbf{n}}\big(\mathsf{ZeroPadMSB}_{\mathbf{n}-n}(\pi)\big) = \mathsf{Duplicate}_{\mathbf{n}-n}\big(\mathsf{Enc}_n(\pi)\big). \tag{5}$$

The lifted codeword is therefore the original codeword with each entry duplicated $2^{\mathbf{n}-n}$ times. Given Merkle access to the original codeword $c = \mathsf{Enc}_n(\pi)$, the verifier emulates oracle access to the lifted codeword by translating a query at position $e \in E_{\mathbf{n}}$ into a query at $\lfloor e \cdot |E_n|/|E_{\mathbf{n}}| \rfloor \in E_n$.

Using this lift, every LIOP oracle is virtually placed at a common level $\mathbf{n} = \max_i n_i$ (where n_i is the variable count of the i-th oracle) before the parties run Batched BaseFold (§5.4). Soundness of the lift requires a refined proximity-gap analysis (the lifted codeword's proximity to the level-$\mathbf{n}$ code is constrained by the original's proximity to the level-n code, with a scaling factor of $2^{\mathbf{n}-n}$); see [HMH] for details.

5.4 Batched BaseFold

The compiler of §5.1 runs one BaseFold instance per LIOP linear query. When the LIOP commits multiple oracles with exactly one opening claim per oracle, the openings can be discharged with a single BaseFold instance via two preliminary reductions. We present this batched variant, which is the setting needed for the IronSpartan LIOP of §6. (The Binius64 LIOP itself uses only one oracle and one opening, so it does not need this machinery.)

Suppose the LIOP holds opening claims

$$\langle \pi_i, t_i \rangle = s_i, \qquad i = 0, \ldots, k-1,$$

where $\pi_i \colon \mathcal{B}_{n_i} \to K$ is the i-th oracle, $t_i \colon \mathcal{B}_{n_i} \to K$ is the *transparent* operand—a multilinear the verifier can evaluate on its own, with no help from the prover—and $s_i \in K$ is the target.

Step 1: Batched sumcheck. The verifier samples a single batching challenge $\alpha \leftarrow K$. Let $\pi_i^{(\uparrow)}, t_i^{(\uparrow)} \colon \mathcal{B}_{\mathbf{n}} \to K$ denote the ZeroPadMSB lifts (§5.3) of π_i, t_i to the common variable count $\mathbf{n} \coloneqq \max_i n_i$; the lifts vanish off the original domains, so $\sum_{x \in \mathcal{B}_{\mathbf{n}}} \widetilde{\pi}_i^{(\uparrow)}(x)\, \widetilde{t}_i^{(\uparrow)}(x) = s_i$. The parties run a sumcheck on

$$\sum_{i=0}^{k-1} \alpha^i s_i \overset{?}{=} \sum_{x \in \mathcal{B}_{\mathbf{n}}} \sum_{i=0}^{k-1} \alpha^i \cdot \widetilde{\pi}_i^{(\uparrow)}(x) \cdot \widetilde{t}_i^{(\uparrow)}(x).$$

After $\mathbf{n}$ rounds with sumcheck challenges $r \in K^{\mathbf{n}}$, the batched claim reduces to $\sum_{i=0}^{k-1} \alpha^i \cdot \widetilde{\pi}_i^{(\uparrow)}(r) \cdot \widetilde{t}_i^{(\uparrow)}(r)$. The verifier evaluates each $\widetilde{t}_i^{(\uparrow)}(r)$ on its own (via the factorization (4)). The remaining unknowns are the prover-claimed evaluations

$$\alpha_i \coloneqq \widetilde{\pi}_i^{(\uparrow)}(r), \qquad i = 0, \ldots, k-1.$$

Step 2: Random reduction to a piecewise-concatenated oracle. To verify the k evaluation claims α_i via a single BaseFold instance, we treat the $\pi_i^{(\uparrow)}$ as restrictions of a single *piecewise concatenated* multilinear

$$\boldsymbol{\pi}(X, Y) \coloneqq \sum_{i=0}^{k-1} \widetilde{\pi}_i^{(\uparrow)}(X) \cdot \widetilde{\mathsf{eq}}_{\log_2 k}(i, Y),$$

in $X \in K^{\mathbf{n}}$ and $\log_2 k$ fresh oracle-index variables $Y \in K^{\log_2 k}$, so that $\boldsymbol{\pi}(X, i) = \widetilde{\pi}_i^{(\uparrow)}(X)$ for each $i \in \mathcal{B}_{\log_2 k}$.

The verifier samples $r' \leftarrow K^{\log_2 k}$ and computes the target

$$s' \coloneqq \sum_{i=0}^{k-1} \alpha_i \cdot \widetilde{\mathsf{eq}}_{\log_2 k}(i, r') = \boldsymbol{\pi}(r, r').$$

The parties then run a single BaseFold instance on $\boldsymbol{\pi}$ at the point (r, r') with target s'. The k separately-committed codewords of $\pi_0, \ldots, \pi_{k-1}$ from the LIOP together form the virtual oracle for $\boldsymbol{\pi}$: a BaseFold query at position $(e, y) \in E_{\mathbf{n}} \times \mathcal{B}_{\log_2 k}$ is answered by querying the codeword of π_y at position e, with the Merkle authentication path drawn from the original π_y commitment.

5.5 The Binius64 SNARK

The Binius64 SNARK is the result of applying the compiler of §5.1 to the Binius64 LIOP of §4. The LIOP has a single oracle $\pi = \mathsf{pack}_\lambda(w)\colon \mathcal{B}_{\ell_{\text{pack}}} \to K$ and ends with a single succinct linear query (t, s), namely the random combination of the ring-switching and public-input-check queries (§4.7). The compiler instantiates this LIOP query with a single BaseFold IOPP instance on the constrained Reed–Solomon code (RS code over K with the constraint $\sum_y t(y) \cdot \pi(y) = s$), and applies BCS and Fiat–Shamir on top.

Concretely, the non-interactive proof consists of:

- the Merkle root of $\mathsf{Enc}_{\ell_{\text{pack}}}(\pi)$, sent at the start (commitment to the oracle);
- the LIOP's non-oracle messages (the univariate polynomial $g(\widehat{I})$ of the BitAnd reduction, GKR-step claimed evaluations and Frobenius-step values of the IntMul reduction, the round polynomials of the various sumchecks, the vector $\hat{s}$ of the ring-switching step), with verifier challenges Fiat–Shamir-derived from the running transcript;
- the BaseFold transcript verifying the LIOP's final linear query: Merkle roots of the folded codewords, the sumcheck round polynomials of BaseFold's folding phase, and the Merkle openings answering the μ query positions.

The verifier reconstructs each LIOP and BaseFold challenge by hashing the transcript prefix in the canonical Fiat–Shamir order, runs each LIOP verifier-side check, and runs the BaseFold verifier on the IOPP transcript.

Soundness. Combining the soundness errors of the LIOP, BaseFold, and BCS gives an overall soundness error of $O(\ell_{\text{words}} + \ell_{\text{cons}})/|K| + 2^{-\lambda}$ in the random oracle model, where the first term collects the various sumcheck and Schwartz–Zippel errors of §4.3–§4.7 and the second term is the BaseFold soundness floor. Knowledge soundness follows from the same compilation chain with knowledge extraction at each stage.

6 The IronSpartan LIOP

The Binius64 LIOP of §4 is specialized to the BitAnd and IntMul constraint system. We now specify a second LIOP, *IronSpartan*, for the simple language of rank-one constraint systems (R1CS). IronSpartan adapts the Spartan PIOP of Setty [Set20] to the LIOP framework of §2.3, and presents it as a *commit-and-prove* argument: the assignment is split into a public, a precommit, and a witness segment, and the prover commits the precommit segment as an oracle *before* the public instance is fixed. Where Spartan answers its single witness-evaluation query with a polynomial commitment scheme, IronSpartan phrases the analogous claim as linear queries on the committed segment oracles, to be discharged by the batched compiler of §5. The construction is moreover honest-verifier zero-knowledge (HVZK), so composed with the Batched ZK BaseFold compiler of §7.2 it yields a zero-knowledge commit-and-prove SNARK for R1CS.

6.1 R1CS and the Spartan PIOP

IronSpartan proves satisfaction of a rank-one constraint system in *commit-and-prove* form, over an assignment partitioned into three *data segments*

$$z = (z_{\mathsf{pub}}, z_{\mathsf{pre}}, z_{\mathsf{wit}}),$$

indexed by $\sigma \in \{\mathsf{pub}, \mathsf{pre}, \mathsf{wit}\}$ – the *public*, *precommit*, and *witness* segments, of sizes $n_{\mathsf{pub}}, n_{\mathsf{pre}}, n_{\mathsf{wit}}$. The public segment z_{pub} carries the constant 1 together with the public inputs and is held by the verifier; the precommit and witness segments are private and are committed as oracles. The defining feature of the commit-and-prove setting is that the prover sends the precommit oracle z_{pre} *before* the public instance z_{pub} is fixed: the precommitted data cannot depend on the statement, and one precommitment may be reused across many statements.

The constraint system is specified by *nine wiring matrices* $W_{\mathsf{op},\sigma} \in K^{n_M \times n_\sigma}$, one per operator $\mathsf{op} \in \{A, B, C\}$ and segment σ. Concatenating the segments columnwise recovers the usual R1CS

matrices, $A = [\, W_{A,\mathsf{pub}} \mid W_{A,\mathsf{pre}} \mid W_{A,\mathsf{wit}} \,]$ and likewise for B and C, so that each matrix–vector product splits across segments as

$$\mathsf{op} \cdot z = \sum_{\sigma} W_{\mathsf{op},\sigma}\, z_\sigma, \qquad \mathsf{op} \in \{A, B, C\}.$$

The assignment *satisfies* the instance if

$$(Az) \circ (Bz) = Cz,$$

where $\circ$ denotes the entry-wise product. We pad n_M and each segment size n_σ to powers of two, $n_M = 2^{\ell_M}$ and $n_\sigma = 2^{\ell_\sigma}$, indexing constraints (matrix rows) by $\mathcal{B}_{\ell_M}$ and the entries of segment σ (matrix columns) by $\mathcal{B}_{\ell_\sigma}$.

Multilinear encoding. Each wiring matrix $W_{\mathsf{op},\sigma} \colon \mathcal{B}_{\ell_M} \times \mathcal{B}_{\ell_\sigma} \to K$ and each segment $z_\sigma \colon \mathcal{B}_{\ell_\sigma} \to K$ has a multilinear extension, written $\widetilde{W}_{\mathsf{op},\sigma}(X, Y)$ and $\widetilde{z}_\sigma(Y)$. For $M \in \{A, B, C\}$, write

$$\widetilde{(Mz)}(X) \coloneqq \sum_{\sigma} \sum_{y \in \mathcal{B}_{\ell_\sigma}} \widetilde{W}_{M,\sigma}(X, y)\, \widetilde{z}_\sigma(y)$$

for the multilinear extension of the product vector Mz. The relation holds iff $\widetilde{(Az)}(x) \cdot \widetilde{(Bz)}(x) = \widetilde{(Cz)}(x)$ for every constraint $x \in \mathcal{B}_{\ell_M}$.

The Spartan PIOP. After committing the assignment, Spartan [Set20] verifies this relation in two sumcheck stages.

1. **Constraint check.** The verifier samples $\tau \leftarrow K^{\ell_M}$, and the parties run an MLE-check (§2) on

$$\sum_{x \in \mathcal{B}_{\ell_M}} \widetilde{\mathsf{eq}}_{\ell_M}(\tau, x) \cdot \left[\widetilde{(Az)}(x)\, \widetilde{(Bz)}(x) - \widetilde{(Cz)}(x) \right] = 0.$$

   This reduces to a random point $r_x \in K^{\ell_M}$ at which the prover sends the three claimed products $a = \widetilde{(Az)}(r_x)$, $b = \widetilde{(Bz)}(r_x)$, $c = \widetilde{(Cz)}(r_x)$, for which the verifier checks $a \cdot b = c$.

2. **Wiring check.** The verifier samples $\gamma \leftarrow K$ and a second MLE-check batches the three products into a single evaluation query on the committed assignment.

Spartan answers the resulting evaluation query with a polynomial commitment scheme; IronSpartan replaces this stage, as we now describe.

6.2 IronSpartan Overview

IronSpartan departs from Spartan in how the wiring check is discharged, exploiting the segment structure to keep the public data out of the committed oracles. For the batching challenge γ, define for each segment the *batched wiring operand*

$$t_\sigma(Y) \coloneqq \widetilde{W}_{A,\sigma}(r_x, Y) + \gamma\, \widetilde{W}_{B,\sigma}(r_x, Y) + \gamma^2\, \widetilde{W}_{C,\sigma}(r_x, Y), \qquad \sigma \in \{\mathsf{pub}, \mathsf{pre}, \mathsf{wit}\},$$

a multilinear on $\mathcal{B}_{\ell_\sigma}$ that the verifier evaluates from the public wiring matrices and the challenges r_x, γ. Batching the products a, b, c with γ and splitting across segments,

$$a + \gamma b + \gamma^2 c = \sum_{\sigma} \langle z_\sigma, t_\sigma \rangle = \langle z_{\mathsf{pub}}, t_{\mathsf{pub}} \rangle + \langle z_{\mathsf{pre}}, t_{\mathsf{pre}} \rangle + \langle z_{\mathsf{wit}}, t_{\mathsf{wit}} \rangle.$$

The public summand the verifier computes on its own; the precommit and witness summands are *linear queries* (§2.3) on the committed oracles z_{pre} and z_{wit}, with operands t_{pre} and t_{wit}. As in the Binius64 case these operands are *transparent* – the verifier evaluates each t_σ from the public wiring matrices – but, unlike there, they are not succinct: evaluating t_σ reads the wiring matrices. IronSpartan does not require succinctness. It therefore omits Spartan's second sumcheck entirely, emitting the wiring claim as linear queries that the compiler (§5) verifies against the Reed–Solomon encodings of the segment oracles. The constraint check is retained as in Spartan.

Splitting the target. The constraint check fixes only the total $s := a + \gamma b + \gamma^2 c$, whereas the compiler needs a separate target for each oracle query. Once γ is sampled, the prover sends a single field element, the precommit contribution

$$s_{\mathsf{pre}} := \langle z_{\mathsf{pre}}, t_{\mathsf{pre}} \rangle.$$

The verifier computes the public contribution $s_{\mathsf{pub}} := \langle z_{\mathsf{pub}}, t_{\mathsf{pub}} \rangle$ on its own and recovers the witness target as the residual

$$s_{\mathsf{wit}} := s - s_{\mathsf{pub}} - s_{\mathsf{pre}}.$$

IronSpartan then emits two linear queries, $(t_{\mathsf{pre}}, s_{\mathsf{pre}})$ on z_{pre} and $(t_{\mathsf{wit}}, s_{\mathsf{wit}})$ on z_{wit}. Together with the verifier's public computation these pin the full inner product: a prover satisfying both queries satisfies $\sum_\sigma \langle z_\sigma, t_\sigma \rangle = s$. The extra element s_{pre} costs no soundness, as it is itself certified by the precommit query.

Toward zero knowledge. Three leakage sources must be closed to make IronSpartan honest-verifier zero-knowledge:

- the *constraint-check sumcheck*, whose round polynomials depend on the committed segments; we mask it with a Libra-style masking polynomial committed as an auxiliary oracle π_g (§6.3);
- the *oracle openings* of z_{pre}, z_{wit}, and π_g performed by the compiler's BaseFold instance; these are masked by the randomizable support and mask oracles of the Batched ZK BaseFold compiler (§7.2);
- the revealed products (a, b, c), witness-dependent linear images of the assignment; we mask these with dummy constraints (§6.4).

The first and third are intrinsic to the LIOP and are treated below; the second is supplied by the compiler. Instantiated over binary fields via the binary BaseFold of §5.2 (FRI-Binius [DP25]), the result runs over the same fields as the rest of this document.

6.3 Zero-Knowledge Sumcheck Machinery

IronSpartan's constraint check (§6.5.1) is a sumcheck, and the standard sumcheck protocol is not zero-knowledge: each round polynomial is a partial sum of the witness-dependent summand and leaks information about it. We recall the masking technique of Libra [Xie+19], which restores zero knowledge in odd characteristic but *fails* in characteristic 2 – the case of interest, since K has characteristic 2 – and then show that the *multilinear-extension-check* form of IronSpartan's constraint check repairs the masking in every characteristic. The honest-verifier zero-knowledge statement is Theorem 6.3, proved in Appendix C.

Throughout, $f \in K[X_0, \ldots, X_{n-1}]$ is an n-variate polynomial of individual degree at most d, and we treat the sumcheck claim $\sum_{x \in \mathcal{B}_n} f(x) = s_f$. In round $i = 0, \ldots, n-1$ the prover binds X_{n-i-1} and sends the univariate *round polynomial*

$$R_f^{(i)}(X) := \sum_{v \in \mathcal{B}_{n-i-1}} f(v, X, r_{n-i}, \ldots, r_{n-1}), \qquad \deg R_f^{(i)} \le d,$$

where $r_{n-i}, \ldots, r_{n-1}$ are the prior challenges; the verifier checks $R_f^{(i)}(0) + R_f^{(i)}(1) = s^{(i)}$ against the running claim (with $s^{(0)} = s_f$), samples $r_{n-i-1} \leftarrow K$, and sets $s^{(i+1)} := R_f^{(i)}(r_{n-i-1})$. Transmitting d of the $d+1$ coefficients of each $R_f^{(i)}$ suffices, the last being fixed by the consistency check; these coefficients, with s_f, are the messages a zero-knowledge variant must hide.

6.3.1 Libra and Its Failure in Characteristic 2

Libra masks the sumcheck by additively blending in a random polynomial. The prover samples a *masking polynomial*

$$g(X_0, \ldots, X_{n-1}) = g_c + \sum_{j=0}^{n-1} g_j(X_j) \cdot X_j, \qquad g_j(X) = \sum_{k=0}^{d-1} g_{j,k} X^k,$$

with constant $g_c \in K$ and random univariates g_j of degree $\le d-1$, so g has individual degree $\le d$ and $1+nd$ coefficients. It announces $s_g := \sum_{v\in\mathcal{B}_n} g(v)$; the verifier samples a blending challenge $\alpha \leftarrow K$; and the parties sumcheck the combined claim

$$s_f + \alpha\, s_g = \sum_{x\in\mathcal{B}_n} (f + \alpha\, g)(x),$$

with round polynomials $R^{(i)} = R_f^{(i)} + \alpha\, R_g^{(i)}$. After the final challenge the prover reveals $g^* := g(r)$ at $r = (r_0, \dots, r_{n-1})$, and the verifier recovers the unmasked value $f^* := s^{(n)} - \alpha\, g^*$.

Committing the mask. Since g has individual degree $d > 1$ it is not multilinear and cannot be an LIOP oracle directly; the prover commits a multilinear proxy for its coefficients. Put

$$\nu := \lceil \log_2 n \rceil, \qquad \delta := \lceil \log_2 (d+1) \rceil,$$

and let $g' \colon \mathcal{B}_\nu \times \mathcal{B}_\delta \to K$ be a uniformly random multilinear, committed as the oracle $\pi_g := g'$. Identifying $\mathcal{B}_\nu \cong \{0, \dots, 2^\nu - 1\}$ and $\mathcal{B}_\delta \cong \{0, \dots, 2^\delta - 1\}$ (§2), the prover reads the mask coefficients off g',

$$g_c := \sum_{j\in\mathcal{B}_\nu} g'(j, 0), \qquad g_{j,k-1} := g'(j, k) \quad (j < n,\ 1 \le k \le d),$$

the other entries of g' remaining slack. To evaluate g at the sumcheck point r, define the multilinear $\mathsf{libra\text{-}eval}_r \colon \mathcal{B}_\nu \times \mathcal{B}_\delta \to K$ by

$$\mathsf{libra\text{-}eval}_r(j, k) := \begin{cases} r_j^{\,k} & \text{if } j < n \text{ and } k \le d, \\ 0 & \text{otherwise,} \end{cases}$$

with r_j the j-th coordinate of r. Expanding,

$$\langle g', \mathsf{libra\text{-}eval}_r \rangle = \sum_{(j,k)\in\mathcal{B}_{\nu+\delta}} g'(j, k)\, \mathsf{libra\text{-}eval}_r(j, k) = g_c + \sum_{j=0}^{n-1} g_j(r_j)\, r_j = g(r),$$

so the claim $g^* = g(r)$ is a linear query on π_g with operand $\mathsf{libra\text{-}eval}_r$ and target g^*, which the verifier evaluates in $O(nd)$ operations.

PROTOCOL 6.1 (Zero-knowledge sumcheck via Libra masking)**.** *Claim:* $\sum_{x\in\mathcal{B}_n} f(x) = s_f$, for f of individual degree $\le d$.

1. *Masking.* The prover samples $g' \leftarrow K^{\mathcal{B}_{\nu+\delta}}$, commits $\pi_g := g'$, derives g as above, and sends $s_g = \sum_{v\in\mathcal{B}_n} g(v)$. The verifier samples $\alpha \leftarrow K$ and sets $s^{(0)} := s_f + \alpha\, s_g$.
2. *Rounds.* For $i = 0, \dots, n-1$: the prover sends all but one coefficient of $R^{(i)} = R_f^{(i)} + \alpha\, R_g^{(i)}$; the verifier recovers the last from $R^{(i)}(0) + R^{(i)}(1) = s^{(i)}$, samples $r_{n-i-1} \leftarrow K$, and sets $s^{(i+1)} := R^{(i)}(r_{n-i-1})$.
3. *Reduction.* The prover reveals $g^* = g(r)$. The verifier forms $f^* := s^{(n)} - \alpha\, g^*$ and outputs the evaluation claim $f^* = f(r)$ on f and the linear query $g^* = \langle g', \mathsf{libra\text{-}eval}_r \rangle$ on π_g.

Failure in characteristic 2. The masking messages – s_g and the transmitted coefficients of the $R_g^{(i)}$ – form a fixed K-linear image L_r of the $1+nd$ coefficients of g. Expanding over the hypercube,

$$s_g = 2^n g_c + 2^{n-1} \sum_{j=0}^{n-1} g_j(1), \qquad R_g^{(i)}(X) = 2^{n-i-1}\, g_{n-i-1}(X)\, X + (\text{terms in } g_c \text{ and the } g_j,\ j \ne n-i-1),$$

so the message block carrying the fresh univariate g_{n-i-1} is scaled by 2^{n-i-1}, and L_r is block lower-triangular with these powers of two on its diagonal. In odd characteristic the diagonal entries are units, L_r is invertible, and a uniform mask yields uniform messages (Appendix C). In characteristic 2, however, $2^k = 0$ for $k > 0$: the diagonal collapses for every round $i < n-1$, the rank of L_r falls to $1+d$, and the non-constant coefficients of the first $n-1$ round polynomials are left unmasked. Libra masking thus gives no zero knowledge over K.

6.3.2 Zero-Knowledge Multilinear-Extension Check

IronSpartan's constraint check is not a bare sumcheck but a *multilinear-extension check* (§2): the summand carries an explicit equality-indicator factor, the claim being $\sum_{x\in\mathcal{B}_n} \widetilde{\mathsf{eq}}_n(\tau,x)\, f(x) = s_f$ for a verifier challenge $\tau \in K^n$. Masking it with the same g, g', α – now summing the $\widetilde{\mathsf{eq}}_n(\tau,\cdot)$-weighted combination – repairs the masking in every characteristic, following Gruen [Gru24]. The equality-indicator factor is pulled out of the round polynomials, so each $R^{(i)}$ stays of degree $\le d$ and the round relation becomes the $\widetilde{\mathsf{eq}}$-*weighted* consistency

$$s^{(i)} = (1-\tau_{n-i-1})\, R^{(i)}(0) + \tau_{n-i-1}\, R^{(i)}(1), \qquad s^{(i+1)} := R^{(i)}(r_{n-i-1}),$$

in place of the unweighted $R^{(i)}(0) + R^{(i)}(1) = s^{(i)}$ of the bare sumcheck.

PROTOCOL 6.2 (Zero-knowledge multilinear-extension check)**.** *Claim:* $\sum_{x\in\mathcal{B}_n} \widetilde{\mathsf{eq}}_n(\tau,x)\, f(x) = s_f$, for a challenge $\tau \in K^n$ and f of individual degree $\le d$.

The protocol is Protocol 6.1 with every hypercube sum weighted by $\widetilde{\mathsf{eq}}_n(\tau,\cdot)$: the prover sends $s_g = \sum_{v\in\mathcal{B}_n} \widetilde{\mathsf{eq}}_n(\tau,v)\, g(v)$, and $R_f^{(i)}, R_g^{(i)}$ are the equality-indicator-factored round polynomials of the weighted summand, each of degree $\le d$, computed by the MLE-check prover of [Gru24]. In round i the prover sends the degree-≥ 1 coefficients of $R^{(i)} = R_f^{(i)} + \alpha\, R_g^{(i)}$; the verifier recovers the constant coefficient from $s^{(i)} = (1-\tau_{n-i-1})\, R^{(i)}(0) + \tau_{n-i-1}\, R^{(i)}(1)$, samples $r_{n-i-1} \leftarrow K$, and sets $s^{(i+1)} := R^{(i)}(r_{n-i-1})$. The reduction is unchanged: it outputs the evaluation claim $f^* = f(r)$ on f and the linear query $g^* = \langle g', \mathsf{libra\text{-}eval}_r\rangle$ on π_g.

Why it works. With the equality-indicator factor pulled out, the as-yet-unbound low variables are summed against $\widetilde{\mathsf{eq}}_{n-i-1}(\tau_{<n-i-1},\cdot)$, contributing the factor $\sum_{x_{<n-i-1}} \widetilde{\mathsf{eq}}_{n-i-1}(\tau_{<n-i-1}, x_{<n-i-1}) = 1$ – in place of the $\sum_{x_{<n-i-1}} 1 = 2^{n-i-1}$ of the bare sumcheck. The masking contributions are therefore

$$s_g = g_c + \sum_{j=0}^{n-1} \tau_j\, g_j(1), \qquad R_g^{(i)}(X) = g_c + \sum_{j>n-i-1} g_j(r_j)\, r_j + \sum_{j<n-i-1} \tau_j\, g_j(1) + g_{n-i-1}(X)\, X.$$

The fresh univariate appears as $g_{n-i-1}(X)\, X$ with *unit* coefficient, rather than scaled by the vanishing power of two 2^{n-i-1}. Its degree-≥ 1 coefficients are exactly those of g_{n-i-1}, so the round message hides the corresponding coefficients of f in every characteristic; the masking is in fact perfect once the blending challenge α is nonzero (Appendix C).

Theorem 6.3. *The zero-knowledge multilinear-extension check (Protocol 6.2) is honest-verifier zero-knowledge: there is an efficient simulator whose transcript, given the public claim (τ, s_f) and the reduced value f^*, is statistically $O(1)/|K|$-close to an honest execution. The same masking applied to the bare sumcheck (Protocol 6.1) is honest-verifier zero-knowledge only when* $\mathrm{char}(K) \neq 2$.

The proof is given in Appendix C. This masked multilinear-extension check is what IronSpartan uses for its constraint check (§6.5.1).

6.4 Dummy-Constraint Masking

Masking the sumcheck and the oracle openings still leaves one leak: the constraint check reveals the products (a, b, c), which are witness-dependent linear images of the assignment evaluated at the random point r_x. We neutralize this by enlarging the instance with *dummy constraints* carrying fresh random values, in the manner of Diamond [Dia25]. Concretely, we append two multiplication constraints whose operands are fresh witness-segment variables assigned uniformly random field elements consistent with the multiplicative relation, occupying two otherwise-unused matrix rows and six fresh columns of the witness segment (two operands and one product per dummy constraint). The wiring extensions $\widetilde{W}_{A,\mathsf{wit}}(r_x,\cdot), \widetilde{W}_{B,\mathsf{wit}}(r_x,\cdot), \widetilde{W}_{C,\mathsf{wit}}(r_x,\cdot)$ then carry random components supported on the dummy columns, rendering the revealed (a, b, c) uniform subject to $a \cdot b = c$ and independent of the real witness.

6.5 The IronSpartan LIOP

Setup. The setup pads the constraint count and each segment to power-of-two dimensions, reserving room for the masking machinery. Writing q for the compiler's query count (§7.2), it chooses ℓ_M and the ℓ_σ with

$$2^{\ell_M} \geq n_M + 2, \qquad 2^{\ell_{\mathsf{pub}}} \geq n_{\mathsf{pub}}, \qquad 2^{\ell_{\mathsf{pre}}} \geq n_{\mathsf{pre}} + (q+1), \qquad 2^{\ell_{\mathsf{wit}}} \geq n_{\mathsf{wit}} + 6 + (q+1).$$

The two extra constraint rows hold the dummy constraints of §6.4; the six extra witness-segment columns hold their fresh variables; and each committed segment, z_{pre} and z_{wit}, reserves a size-$(q+1)$ randomizable support (§2.3) for the ZK compiler. The randomizable-support columns are all-zero across the wiring matrices, so every operand t_σ vanishes on them, meeting the well-formedness condition of §2.3; the dummy columns, by contrast, are used by the dummy rows. Remaining padding entries are zero.

Protocol. In keeping with the commit-and-prove setting, the prover first commits the precommit oracle z_{pre} (with its randomizable support filled uniformly). Once the public instance z_{pub} is fixed, the prover commits the witness oracle z_{wit} (likewise) and the Libra oracle π_g (the masking coefficients g' of §6.3.2). The parties then run the constraint-check reduction (§6.5.1) followed by the reduction to inner-product claims (§6.5.2), terminating in linear queries on z_{pre}, z_{wit}, and π_g – three oracles with one query each, exactly the setting of Batched BaseFold (§5.4).

6.5.1 Constraint-Check Reduction

The constraint-check reduction is the zero-knowledge MLE-check of §6.3.2 applied to the R1CS constraint polynomial.

1. The verifier samples $\tau \leftarrow K^{\ell_M}$.
2. The parties run a zero-knowledge MLE-check, masked by the Libra polynomial committed in π_g, on
$$\sum_{x \in \mathcal{B}_{\ell_M}} \widetilde{\mathsf{eq}}_{\ell_M}(\tau, x) \cdot \left[\widetilde{(Az)}(x)\,\widetilde{(Bz)}(x) - \widetilde{(Cz)}(x)\right] = 0.$$
3. The check reduces to a random point $r_x \in K^{\ell_M}$, at which the prover sends the claimed products $a = \widetilde{(Az)}(r_x)$, $b = \widetilde{(Bz)}(r_x)$, $c = \widetilde{(Cz)}(r_x)$ of §6.1; the verifier checks $a \cdot b = c$ and samples a batching challenge $\gamma \leftarrow K$.

The reduction outputs the claimed products (a, b, c), the point r_x, the challenge γ, and a Libra evaluation claim – a linear query on π_g (operand $\mathsf{libra\text{-}eval}_r$, target g^*).

6.5.2 Reduction to Inner-Product Claims on Matrix Extensions

It remains to certify the three products (a, b, c). As in §6.2, IronSpartan batches them with γ and splits the inner product across segments using the operands t_σ. The prover sends the precommit contribution $s_{\mathsf{pre}} = \langle z_{\mathsf{pre}}, t_{\mathsf{pre}} \rangle$; the verifier computes the public contribution $s_{\mathsf{pub}} = \langle z_{\mathsf{pub}}, t_{\mathsf{pub}} \rangle$ and forms the witness target $s_{\mathsf{wit}} = (a + \gamma b + \gamma^2 c) - s_{\mathsf{pub}} - s_{\mathsf{pre}}$. The LIOP thus terminates with three linear queries – $(t_{\mathsf{pre}}, s_{\mathsf{pre}})$ on z_{pre}, $(t_{\mathsf{wit}}, s_{\mathsf{wit}})$ on z_{wit}, and the Libra query on π_g – whose transparent operands the verifier evaluates from the public matrices. The compiler of §5 discharges all three with a single Batched BaseFold instance. Replacing it with the Batched ZK BaseFold compiler of §7.2, and using the randomizable supports and dummy-constraint masking established above, makes the compiled argument honest-verifier zero-knowledge; the full simulator is given in §7.

7 Zero-Knowledge Compiler

7.1 Proof Composition for ZK-SNARKs

Our zero-knowledge SNARK is obtained by *proof composition*: an arbitrary, not-necessarily-zero-knowledge LIOP for a relation R is wrapped inside the zero-knowledge commit-and-prove IronSpartan

LIOP of §6, yielding a zero-knowledge LIOP for the same relation R. The strategy instantiates the classical paradigm of Ben-Or et al. [Ben+90] – every interactive proof can be made zero-knowledge by having the prover commit (encrypt) its messages and then prove, in zero knowledge, that the verifier's acceptance predicate holds on the committed messages. We follow the concrete realizations of Frigo and shelat [Fs24] and of VEIL [DHRR26], which specialize this paradigm to the hash-based multilinear proof systems compiled in §5. IronSpartan is the outer, zero-knowledge proof: its R1CS commit-and-prove interface is exactly what is needed to certify the inner verifier's predicate without revealing the inner prover's messages.

We call the wrapped protocol the *inner* LIOP, and its prover and verifier the inner prover and inner verifier. The composition encrypts every value the inner prover would reveal under a one-time pad, so it requires the number of such values to be *predetermined* – a fixed function of the relation R rather than of the witness. There are two kinds, each needing its own OTP key: the non-oracle field elements the inner prover sends during the protocol, and the targets of the linear claims its verifier reduces to. Both counts are predetermined for the LIOPs of this document. The two protocols are then interleaved as follows.

1. **Precommit.** Before the public instance is fixed, the composed prover commits IronSpartan's precommit segment z_{pre}, which holds one independent uniform *one-time-pad (OTP) key* for each non-oracle field element the inner prover will send. (The OTP keys for the linear-claim targets are supplied separately, by the randomizable supports of the inner oracles; see below.) Because the count is predetermined and the precommit segment may be fixed ahead of the statement (§6), this commitment is well defined.

2. **Run the inner LIOP.** The composed prover runs the inner prover against the public-coin verifier. Whenever the inner prover would send a non-oracle field element m, the composed prover instead sends the ciphertext $c \coloneqq m + k$, where k is the OTP key precommitted for that element. Since each k is uniform and used once, every c is uniform and independent of m, so the inner prover's non-oracle messages leak nothing. The inner oracles, meanwhile, are committed as oracles of the composed LIOP, to be discharged by the compiler of §7.2.

3. **Run IronSpartan.** Finally, the composed prover invokes the IronSpartan LIOP to prove that the plaintexts $m = c - k$ – with the ciphertexts c public and the keys k read from the precommit segment – satisfy exactly the algebraic relations that the inner verifier would check on the inner prover's messages. These checks are low-degree in the transmitted elements and the (public) challenges, so they form an R1CS instance over $(z_{\mathsf{pre}}, z_{\mathsf{wit}})$ that IronSpartan discharges in zero knowledge.

The inner verifier's predicate ultimately reduces to a set of linear claims on the inner oracles – the inner LIOP's terminal linear queries (§2.3). The target $s = \langle \pi_i, t \rangle$ of each such claim is a witness-dependent linear image of an inner oracle, so it too is hidden under a one-time pad. Here the key is drawn from the oracle itself: the targeted oracle π_i carries one additional index in its randomizable support (§2.3), whose uniform value k is the claim's OTP key. The composed prover reveals only the ciphertext $c \coloneqq s + k$, which is uniform; IronSpartan then certifies, against c, that the underlying claim is the one the inner verifier would check. Thus each reduced linear claim consumes exactly one OTP key, just as each non-oracle field element does – the difference being that the claim keys live in the inner oracles' randomizable supports rather than in z_{pre}.

7.2 Batched ZK BaseFold

The compiler of §5.1 is not zero-knowledge: BaseFold's query phase reveals codeword symbols of the committed oracles, and its folding phase reveals sumcheck round polynomials and folded codewords that depend on those oracles. We now give an *alternate* compiler, *Batched ZK BaseFold*, with the stronger guarantee that whenever the source LIOP is honest-verifier zero-knowledge (HVZK), the compiled interactive proof is HVZK as well – and hence, after Fiat–Shamir, a non-interactive zero-knowledge argument in the random oracle model. It follows the construction of Diamond [Dia25], itself inspired by the low-degree-test masking technique of Aurora [Ben+19], adapted to the batched setting of §5.4. It replaces the Batched BaseFold instantiation of Stage 1 in the compiler of §5.1, leaving the BCS and Fiat–Shamir stages untouched.

Two leakage channels must be closed, and we close them with two separate mechanisms. The *query phase* opens each committed codeword at the verifier's chosen positions, directly exposing symbols of $\mathsf{Enc}(\pi_i)$; these we hide with the randomizable support of §2.3. The *folding phase* reveals round polynomials and intermediate codewords that depend on the oracles; these we hide by combining each oracle with an independent random mask codeword, in the manner of Aurora.

Randomizable-support requirement. Write q for the number of query positions sampled in BaseFold's query phase (the parameter μ of §5.2). We require each LIOP oracle $\pi_i \colon \mathcal{B}_{n_i} \to K$ to carry a randomizable support (§2.3) $I_i \subseteq \mathcal{B}_{n_i}$ of size $|I_i| = q+1$. Of these coordinates, q absorb the q codeword symbols of $\mathsf{Enc}_{n_i}(\pi_i)$ exposed in the query phase, and the remaining one supplies the randomness that makes the Merkle commitment hiding, whose role we defer to the detailed zero-knowledge analysis below. By the well-formedness condition of §2.3, every query operand t_i targeting π_i vanishes on I_i, so filling I_i with randomness leaves the opening targets s_i – and thus completeness and soundness – unchanged.

Commit phase. When the LIOP prover would send an oracle $\pi_i \colon \mathcal{B}_{n_i} \to K$, the IOP prover instead proceeds as follows. It first fills the randomizable support, drawing $\pi_i(w) \leftarrow K$ uniformly and independently for each $w \in I_i$ and leaving the protocol-determined coordinates intact. It then samples a uniformly random *mask* $\omega_i \colon \mathcal{B}_{n_i} \to K$ of the same length, and commits to the *interleaved* Reed–Solomon codeword whose two rows are $\mathsf{Enc}_{n_i}(\pi_i)$ and $\mathsf{Enc}_{n_i}(\omega_i)$: the Merkle commitment is taken over the columns of this $2 \times |E_{n_i}|$ array, so that opening a single codeword position e exposes both $\mathsf{Enc}_{n_i}(\pi_i)[e]$ and $\mathsf{Enc}_{n_i}(\omega_i)[e]$ under one authentication path.

Opening phase. The LIOP concludes with opening claims $\langle \pi_i, t_i \rangle = s_i$ for $i = 0, \ldots, k-1$ (§5.4), which the parties discharge in three steps.

1. For each i, the prover sends the masked inner product $\sigma_i := \langle \omega_i, t_i \rangle = \sum_{w \in \mathcal{B}_{n_i}} t_i(w) \cdot \omega_i(w)$.
2. The verifier samples a masking challenge $\gamma \leftarrow K$.
3. Both parties form, for each i, the virtual oracle $\pi_i^\gamma := \pi_i + \gamma \cdot \omega_i$ with target $s_i^\gamma := s_i + \gamma \cdot \sigma_i$, so that $\langle \pi_i^\gamma, t_i \rangle = s_i^\gamma$, and run Batched BaseFold (§5.4) on the claims $\langle \pi_i^\gamma, t_i \rangle = s_i^\gamma$. The verifier emulates oracle access to the codeword $\mathsf{Enc}_{n_i}(\pi_i^\gamma) = \mathsf{Enc}_{n_i}(\pi_i) + \gamma \cdot \mathsf{Enc}_{n_i}(\omega_i)$ by reading both rows of the interleaved commitment at the queried column and combining them with γ (lifting to the common level $\mathbf{n}$ as in §5.3 where needed).

Because γ scales a genuine codeword $\mathsf{Enc}_{n_i}(\omega_i)$, the combination $\mathsf{Enc}_{n_i}(\pi_i^\gamma)$ is again a codeword of the same code, and the masked sumcheck constraint $\langle \pi_i^\gamma, t_i \rangle = s_i^\gamma$ is exactly the constraint Batched BaseFold expects.

Soundness. Fix the committed rows π_i, ω_i. The masked identity $\langle \pi_i^\gamma, t_i \rangle = s_i^\gamma$ is an affine identity in γ that holds for a uniformly random γ only if both $\langle \pi_i, t_i \rangle = s_i$ and $\langle \omega_i, t_i \rangle = \sigma_i$, except with probability $1/|K|$ by Schwartz–Zippel; sending σ_i before γ is sampled prevents the prover from adapting it. The masking step therefore reduces verifying the original claims to verifying the masked claims, adding $O(1)/|K|$ to the soundness error of Batched BaseFold. Knowledge soundness is preserved: the extractor recovers each π_i from the first row of its interleaved commitment exactly as in §5.4.

Zero knowledge. The mask $\gamma \cdot \omega_i$ is a uniformly random codeword (for $\gamma \neq 0$), so the folding-phase transcript of Batched BaseFold run on π_i^γ – its sumcheck round polynomials, intermediate folded codewords, and final reduced codeword – is distributed independently of the witness, as is the revealed σ_i. Independently, the q exposed symbols of each $\mathsf{Enc}_{n_i}(\pi_i)$ are rendered jointly uniform by the random fill on I_i, so the query-phase openings leak nothing. Composed with the HVZK simulator for the source LIOP – which produces the opening targets s_i and the non-oracle transcript without the witness – these two mechanisms let a simulator reproduce the entire compiled transcript. We defer the full simulator, including the treatment of the Merkle commitments, to the detailed analysis below.

7.3 Compiling Binius64 to a ZK-SNARK

We assemble the zero-knowledge SNARK for the Binius64 constraint system from three components: the Binius64 LIOP of §4, the IronSpartan LIOP of §6, and the Batched ZK BaseFold compiler of §7.2. The Binius64 LIOP is complete and knowledge-sound but not zero-knowledge: its sumcheck round polynomials and its terminal evaluation claim are witness-dependent. We take it as the *inner* LIOP of the proof composition of §7.1 and discharge its verifier's predicate with IronSpartan as the outer zero-knowledge proof.

Recall (§4.1) that the Binius64 LIOP commits a single oracle – the packed witness $\pi = \mathsf{pack}_\lambda(w)$ – runs the BitAnd, IntMul, shift, ring-switching, and public-input reductions, and terminates in a single linear query on π. Apart from this committed oracle, the prover transmits a predetermined number N of non-oracle field elements: the sumcheck round-polynomial coefficients, the claimed evaluations passed between reductions, and the GKR layer claims, all fixed in count by the padded constraint-system parameters. This is the predetermined-N condition the composition of §7.1 requires.

Following that composition, the composed prover first precommits, in IronSpartan's precommit segment z_{pre}, one uniform OTP key per non-oracle element. It then runs the Binius64 prover, sending each of the N elements masked by its key, so the transcript reveals only uniform ciphertexts. The Binius64 verifier's acceptance is a conjunction of low-degree checks – the per-round sumcheck consistency relations $R^{(i)}(0) + R^{(i)}(1) = s^{(i)}$, the products checked between the reductions, and the derivation of the final query target – each a low-degree function of the transmitted elements and the public Fiat–Shamir challenges. The composed prover encodes this conjunction as an R1CS instance whose public segment z_{pub} holds the ciphertexts, the challenges, and the Binius64 public statement u; whose precommit segment holds the OTP keys; and whose witness segment z_{wit} holds the plaintext elements and the intermediate values of the checks. Since the Binius64 verifier is succinct, this R1CS is small – its size is that of the verifier's computation, polylogarithmic in the witness – so IronSpartan proves it cheaply, in zero knowledge.

The one place the inner oracle survives into the compiled argument is the Binius64 LIOP's terminal linear query $\langle \pi, t \rangle = s$ – the combined ring-switching and public-input query of §4.1. As s is a witness-dependent linear image of π, the packed-witness oracle carries one index in its randomizable support beyond the $q + 1$ required by Batched ZK BaseFold (§7.2); its uniform value is the OTP key k for this single claim. The composed prover reveals only the ciphertext $c = s + k$, IronSpartan certifies the underlying claim against c, and the query itself is discharged against the committed π by the compiler.

The composed LIOP is thus honest-verifier zero-knowledge and commits four oracles, each with a single opening claim: the packed witness π and IronSpartan's z_{pre}, z_{wit}, and Libra oracle π_g. This is precisely the input shape of Batched ZK BaseFold; compiling Stage 1 with it and leaving the BCS and Fiat–Shamir stages of §5.1 untouched yields a non-interactive zero-knowledge argument for the Binius64 relation in the random oracle model.

Security. The argument is honest-verifier zero-knowledge: a simulator with no access to the witness produces the transcript in three layers. The N non-oracle messages are reproduced as uniform ciphertexts, matching the honest distribution because each OTP key is uniform and used once. IronSpartan's portion is produced by its HVZK simulator (§6.3, Appendix C), which simulates the masked constraint check, the dummy-masked products (§6.4), and the segment-query targets. Finally, the Batched ZK BaseFold simulator (§7.2) reproduces the commit, folding, and query phases of all four oracles from the opening targets alone, using the randomizable supports – including the extra index on π – to render the exposed codeword symbols and the revealed claim ciphertext c uniform. After Fiat–Shamir this is a non-interactive zero-knowledge argument in the random oracle model. Knowledge soundness and the soundness error are inherited from the inner LIOP, IronSpartan, and the compiler, increased only by the $O(1)/|K|$ terms of the masking and composition steps.

A Expressing Common Operations

The constraint system is far more expressive than its two primitive forms suggest; bitwise XOR, NOT, OR, integer addition, and integer subtraction are all expressible using BitAnd constraints with appropriate shifted value indices. We sketch the encodings below, motivated by the RV64I instruction set.

Bitwise AND, XOR, NOT, OR. Bitwise AND is native. To constrain that an unshifted value equals $w[y]$, one uses the shifted index $(y, \mathsf{sll}, 0)$, which acts as the identity. Bitwise XOR is expressible by a BitAnd constraint whose second list contains a shifted index naming the constant `0xFFFFFFFFFFFFFFFF` (available because that constant is placed in the constant stretch): the constraint

$$(x \oplus y) \wedge \texttt{0xFFFFFFFFFFFFFFFF} = z$$

forces $z = x \oplus y$. Bitwise NOT is the same trick: constrain $x \wedge \texttt{0xFFFFFFFFFFFFFFFF} = y \oplus \texttt{0xFFFFFFFFFFFFFFFF}$, which forces $y = \overline{x}$. Bitwise OR follows from the identity $x \vee y = (x \wedge y) \oplus x \oplus y$: the constraint $x \wedge y = x \oplus y \oplus z$ forces $z = x \vee y$.

Integer addition. To constrain $z = (x + y) \bmod 2^{64}$, the prover supplies an auxiliary witness word c_{out} whose bits are the per-position carries. Writing $c_{\text{in}} \coloneqq \mathsf{sll}(c_{\text{out}}, 1)$, the BitAnd constraint

$$(x \oplus c_{\text{in}}) \wedge (y \oplus c_{\text{in}}) = c_{\text{in}} \oplus c_{\text{out}}$$

enforces correct carry propagation, and an XOR-only constraint forces $z = x \oplus y \oplus c_{\text{in}}$.

Integer subtraction. To constrain $z = (x - y) \bmod 2^{64}$, the prover supplies an auxiliary witness word b_{out} whose bits are the per-position borrows. Writing $b_{\text{in}} \coloneqq \mathsf{sll}(b_{\text{out}}, 1)$, the BitAnd constraint

$$(\overline{x} \oplus b_{\text{in}}) \wedge (y \oplus b_{\text{in}}) = b_{\text{in}} \oplus b_{\text{out}}$$

enforces correct borrow propagation, and an XOR-only constraint forces $z = x \oplus y \oplus b_{\text{in}}$.

B Arithmetization of Shift Indicators

This appendix gives the explicit arithmetizations of the eight shift indicators $\widetilde{\mathsf{shift\text{-}ind}}_{\text{op}}$ defined in §4.2, establishing the succinctness claim stated there. The constructions generalize the shifted virtual polynomial construction of Diamond and Posen [DG25, §4.3].

B.1 Logical Shifts and Rotation

We arithmetize $\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{srl}}$ first. The case sll follows by transposing the first two arguments, since on the cube

$$\mathsf{shift\text{-}ind}_{\mathsf{sll}}(i, j, s) = \mathsf{shift\text{-}ind}_{\mathsf{srl}}(j, i, s),$$

and the same identity extends to the multilinear extensions.

The shift indicators of srl, sll, and ror all reduce to two auxiliary functions $\mathsf{s}_k, \mathsf{s}'_k$ on $\mathcal{B}_k^3$, defined for $k \in \{0, \ldots, 6\}$ by

$$\mathsf{s}_k(i, j, s) \coloneqq \begin{cases} 1 & \text{if } j = i + s \\ 0 & \text{otherwise} \end{cases}, \qquad \mathsf{s}'_k(i, j, s) \coloneqq \begin{cases} 1 & \text{if } j = i + s - 2^k \\ 0 & \text{otherwise} \end{cases},$$

where $i, j, s \in \mathcal{B}_k$ are read as integers (§2) and the equalities $j = i + s$ and $j = i + s - 2^k$ are taken over the integers. So s_k indicates that $i + s$ does not overflow modulo 2^k and equals j, while s'_k indicates that $i + s$ overflows by exactly 2^k and equals $j + 2^k$.

By definition,

$$\mathsf{shift\text{-}ind}_{\mathsf{srl}} = \mathsf{s}_6.$$

The function $\mathsf{shift\text{-}ind}_{\mathsf{ror}}(i, j, s)$ holds iff $j \equiv i + s \pmod{64}$; since $0 \le i, s < 64$, this means $j = i + s$ or $j = i + s - 64$, and so

$$\mathsf{shift\text{-}ind}_{\mathsf{ror}} = \mathsf{s}_6 + \mathsf{s}'_6.$$

Recursive arithmetization. The functions $\mathsf{s}_k, \mathsf{s}'_k$ satisfy a recursive identity. The base case is $\mathsf{s}_0 = 1$ and $\mathsf{s}'_0 = 0$. For the inductive case, write $i = (i_{<k-1}, i_{k-1})$ for the splitting of $i \in \mathcal{B}_k$ into its low $k-1$ bits and its top bit, similarly for j and s; then on $\mathcal{B}_k^3$,

$$\mathsf{s}_k(i,j,s) = \begin{cases} \mathsf{s}_{k-1}(i_{<k-1}, j_{<k-1}, s_{<k-1}) & \text{if } i_{k-1} + s_{k-1} = j_{k-1}, \\ \mathsf{s}'_{k-1}(i_{<k-1}, j_{<k-1}, s_{<k-1}) & \text{if } i_{k-1} + s_{k-1} + 1 = j_{k-1}, \\ 0 & \text{otherwise,} \end{cases}$$

$$\mathsf{s}'_k(i,j,s) = \begin{cases} \mathsf{s}_{k-1}(i_{<k-1}, j_{<k-1}, s_{<k-1}) & \text{if } i_{k-1} + s_{k-1} = j_{k-1} + 2, \\ \mathsf{s}'_{k-1}(i_{<k-1}, j_{<k-1}, s_{<k-1}) & \text{if } i_{k-1} + s_{k-1} + 1 = j_{k-1} + 2, \\ 0 & \text{otherwise,} \end{cases}$$

where all bit arithmetic is taken over the integers.

Each of the four bit-pattern conditions on the top bits $(i_{k-1}, j_{k-1}, s_{k-1}) \in \mathcal{B}_1^3$ admits a 2-multiplication arithmetization over $\mathbb{F}_2$:

$$\begin{aligned} i + s = j &\iff 1 + j + s + i \cdot \big(1 + s \cdot (1 + j)\big) = 1, \\ i + s + 1 = j &\iff (1 - i) \cdot j \cdot (1 - s) = 1, \\ i + s = j + 2 &\iff i \cdot (1 - j) \cdot s = 1, \\ i + s + 1 = j + 2 &\iff i + s + j \cdot \big(i + s \cdot (1 + i)\big) = 1. \end{aligned}$$

Substituting these into the recursion yields multilinear formulas for $\widetilde{\mathsf{s}}_k$ and $\widetilde{\mathsf{s}}'_k$ in terms of $\widetilde{\mathsf{s}}_{k-1}$ and $\widetilde{\mathsf{s}}'_{k-1}$. Together, the two formulas advance a pair of registers using at most 12 multiplications per level; over $k = 1, \ldots, 6$ this gives at most 72 multiplications in K to evaluate the pair $(\widetilde{\mathsf{s}}_6, \widetilde{\mathsf{s}}'_6)$ at any point of K^{18}. In particular, $\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{srl}}$, $\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{sll}}$, and $\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{ror}}$ are all succinct.

B.2 Arithmetic Right Shift

The arithmetic right shift sra behaves like srl except that the sign bit of v (bit 63) is replicated into the vacated high positions. Concretely, $\mathsf{sra}(v,s)_i = v_{i+s}$ if $i + s < 64$, and $\mathsf{sra}(v,s)_i = v_{63}$ otherwise. Hence

$$\mathsf{shift\text{-}ind}_{\mathsf{sra}}(i,j,s) = \mathsf{shift\text{-}ind}_{\mathsf{srl}}(i,j,s) + \Big(\prod_{k=0}^{5} j_k\Big) \cdot \mathsf{a}(i,s),$$

where the indicator $\prod_{k=0}^{5} j_k$ selects $j = 63$ and the auxiliary function $\mathsf{a} \colon \mathcal{B}_6 \times \mathcal{B}_6 \to \mathbb{F}_2$ is defined by $\mathsf{a}(i,s) = 1$ iff $i \geq 64 - s$ (equivalently, iff $i + s$ overflows modulo 64).

We arithmetize a via the sequence of functions $\mathsf{a}_k \colon \mathcal{B}_k \times \mathcal{B}_k \to \mathbb{F}_2$, $k \in \{0, \ldots, 6\}$, defined by $\mathsf{a}_k(i,s) = 1$ iff $i \geq 2^k - s$, so that $\mathsf{a} = \mathsf{a}_6$. The base case is $\mathsf{a}_0 = 0$, and for $k \geq 1$,

$$\mathsf{a}_k(i,s) = \begin{cases} 1 & \text{if } i_{k-1} = s_{k-1} = 1, \\ \mathsf{a}_{k-1}(i_{<k-1}, s_{<k-1}) & \text{if } i_{k-1} \neq s_{k-1}, \\ 0 & \text{otherwise.} \end{cases}$$

The arithmetization of this recursion is the single-line update

$$\widetilde{\mathsf{a}}_k(I,S) = I_{k-1} \cdot S_{k-1} + (I_{k-1} + S_{k-1}) \cdot \widetilde{\mathsf{a}}_{k-1}(I,S),$$

costing at most two multiplications per level, hence at most 12 multiplications in K for $\widetilde{\mathsf{a}}_6$. Together with the 72 multiplications for $\widetilde{\mathsf{s}}_6$, evaluating $\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{sra}}$ at any point of K^{18} costs at most 84 multiplications plus a 6-way product for $\prod_{k=0}^{5} J_k$.

B.3 Parallel 32-bit Shifts

The four parallel 32-bit operations $\mathsf{sll32}, \mathsf{srl32}, \mathsf{sra32}, \mathsf{ror32}$ treat each 64-bit word as two independent 32-bit halves and apply the corresponding 32-bit shift to both halves. The shift amount s is taken modulo 32, i.e., only the low 5 bits $S_0, \ldots, S_4$ are used; the top bit S_5 is ignored.

Each 32-bit indicator is a product of a 5-bit version of the corresponding 64-bit indicator (acting on $I_0, \dots, I_4$ and $J_0, \dots, J_4$) with the equality indicator $\widetilde{\mathsf{eq}}(I_5, J_5)$, which restricts movement to within a single half. Explicitly, writing $I_{<5} \coloneqq (I_0, \dots, I_4)$, $J_{<5} \coloneqq (J_0, \dots, J_4)$, and $S_{<5} \coloneqq (S_0, \dots, S_4)$:

$$\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{srl32}}(I, J, S) = \widetilde{\mathsf{s}}_5(I_{<5}, J_{<5}, S_{<5}) \cdot \widetilde{\mathsf{eq}}(I_5, J_5),$$
$$\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{sll32}}(I, J, S) = \widetilde{\mathsf{s}}_5(J_{<5}, I_{<5}, S_{<5}) \cdot \widetilde{\mathsf{eq}}(I_5, J_5),$$
$$\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{ror32}}(I, J, S) = \big(\widetilde{\mathsf{s}}_5(I_{<5}, J_{<5}, S_{<5}) + \widetilde{\mathsf{s}}'_5(I_{<5}, J_{<5}, S_{<5})\big) \cdot \widetilde{\mathsf{eq}}(I_5, J_5),$$
$$\widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{sra32}}(I, J, S) = \widetilde{\mathsf{shift\text{-}ind}}_{\mathsf{srl32}}(I, J, S) + \widetilde{\mathsf{eq}}(I_5, J_5) \cdot \Big(\prod_{k=0}^{4} J_k\Big) \cdot \widetilde{\mathsf{a}}_5(I_{<5}, S_{<5}).$$

The sign-bit selector $\prod_{k=0}^{4} J_k$ in the $\mathsf{sra32}$ formula identifies $j_{<5} = (1, 1, 1, 1, 1)$, which is bit 31 when $J_5 = 0$ and bit 63 when $J_5 = 1$, i.e., the sign bit of whichever half j lies in; the factor $\widetilde{\mathsf{eq}}(I_5, J_5)$ then forces i to lie in the same half.

Each 5-bit auxiliary function is succinct by the same recursion as in §B.1 and §B.2, truncated to $k = 5$: $\widetilde{\mathsf{s}}_5$ and $\widetilde{\mathsf{s}}'_5$ cost at most 60 multiplications, and $\widetilde{\mathsf{a}}_5$ costs at most 10 multiplications. Hence each of the four 32-bit shift indicators is succinct.

C Zero-Knowledge of the Sumcheck Machinery

We prove Theorem 6.3. Both the success of the masked multilinear-extension check (§6.3.2) and the failure of Libra masking in characteristic 2 (§6.3.1) follow from the distribution of the prover's masking messages.

The masking messages. Fix the challenges τ, r. The masking messages – s_g together with, for each round i, the degree-≥ 1 coefficients of $R_g^{(i)}$ that enter the sent $R^{(i)} = R_f^{(i)} + \alpha\, R_g^{(i)}$ – number $1 + nd$ and form a fixed K-linear image of the coefficient vector $c_g = (g_c, (g_{j,k})_{j<n,\, k<d})$ of the mask.

Lemma C.1. *For the multilinear-extension check this map is a bijection of K^{1+nd}, in every characteristic; for the bare sumcheck it is a bijection if and only if $\mathrm{char}(K) \neq 2$. Whenever it is a bijection and g' is uniform, the masking messages are uniform on K^{1+nd}.*

Proof. By the closed form of §6.3.2, $R_g^{(i)}(X) = \big(g_c + \sum_{j>n-i-1} g_j(r_j) r_j + \sum_{j<n-i-1} \tau_j g_j(1)\big) + g_{n-i-1}(X)\, X$, so the degree-$\geq 1$ coefficients of $R_g^{(i)}$ are exactly the coefficients $g_{n-i-1,0}, \dots, g_{n-i-1,d-1}$ of the fresh univariate g_{n-i-1}. Reading the round messages in order $i = 0, \dots, n-1$ recovers $g_{n-1}, g_{n-2}, \dots, g_0$ in turn, after which $s_g = g_c + \sum_j \tau_j g_j(1)$ recovers g_c; the map is thus a bijection, with no condition on the challenges. For the bare sumcheck (§6.3.1) the corresponding coefficients are $2^{n-i-1} g_{n-i-1,k}$, so the round-i block is this same bijection scaled by 2^{n-i-1} – invertible exactly when $2^{n-i-1} \neq 0$ for every i, i.e. when $\mathrm{char}(K) \neq 2$. In characteristic 2 the blocks with $i < n-1$ vanish, the rank falls to $1 + d$, and the non-constant coefficients of the first $n-1$ round polynomials lie outside the image. A bijection carries the uniform distribution on g to the uniform distribution on the messages. □

Simulator.

Proof of Theorem 6.3. The simulator $\mathcal{S}$ receives the public claim (τ, s_f) and the reduced value f^* – in the IronSpartan composition f^* is itself furnished by the surrounding honest-verifier simulator (§7) – and has no access to f or the witness. It runs:

1. Sample the oracle $\pi_g = g' \leftarrow K^{\mathcal{B}_{\nu+\delta}}$ uniformly, as the honest prover does.
2. Sample $s_g \leftarrow K$; receive α; set $s^{(0)} \coloneqq s_f + \alpha\, s_g$.
3. For $i = 0, \dots, n-1$: sample the transmitted coefficients of $R^{(i)}$ uniformly; let the verifier recover the remaining coefficient and sample r_{n-i-1}; set $s^{(i+1)} \coloneqq R^{(i)}(r_{n-i-1})$.
4. Set $g^* \coloneqq (s^{(n)} - f^*)/\alpha$ and program the response to the single query $\langle g', \mathsf{libra\text{-}eval}_r \rangle$ to equal g^*.

In an honest run the messages are $s^{(0)} = s_f + \alpha\, s_g$ and $R^{(i)} = R_f^{(i)} + \alpha\, R_g^{(i)}$. Conditioned on the challenges, f fixes s_f and the $R_f^{(i)}$, while the mask supplies s_g and the $R_g^{(i)}$; by Lemma C.1 these are jointly uniform over the uniform choice of g', so for $\alpha \neq 0$ the sent $(s^{(0)}, (R^{(i)})_i)$ are uniform, matching $\mathcal{S}$ exactly. The lone opening of π_g lands at the point encoded by $\mathsf{libra\text{-}eval}_r$, where $\mathcal{S}$ programs the response to satisfy $f^* = s^{(n)} - \alpha\, g^*$; as g' is committed and opened nowhere else, this is undetectable. The two transcripts coincide except on the event $\alpha = 0$, of probability $1/|K|$, giving statistical distance $O(1)/|K|$. The same argument fails for the bare sumcheck in characteristic 2, where by Lemma C.1 the masking messages are confined to a proper subspace and so cannot match the uniform simulation. □

References

[ACFY25] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. "WHIR: Reed–Solomon Proximity Testing with Super-Fast Verification". In: *Advances in Cryptology – EUROCRYPT 2025*. Ed. by Serge Fehr and Pierre-Alain Fouque. Cham: Springer Nature Switzerland, 2025, pp. 214–243. ISBN: 978-3-031-91134-7.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. "Interactive Oracle Proofs". In: *International Conference on Theory of Cryptography*. Vol. 9986. Berlin, Heidelberg: Springer-Verlag, 2016, pp. 31–60. ISBN: 978-3-662-53644-5. DOI: 10.1007/978-3-662-53644-5_2.

[Ben+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. "Aurora: Transparent Succinct Arguments for R1CS". In: *Advances in Cryptology – EUROCRYPT 2019*. Berlin, Heidelberg: Springer-Verlag, 2019, pp. 103–128. ISBN: 978-3-030-17652-5. DOI: 10.1007/978-3-030-17653-2_4.

[Ben+90] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Håstad, J. Kilian, S. Micali, and P. Rogaway. "Everything provable is provable in zero-knowledge". In: *Proceedings on Advances in Cryptology*. CRYPTO '88. Santa Barbara, California, USA: Springer-Verlag, 1990, pp. 37–56. ISBN: 0387971963.

[CFS17] Alessandro Chiesa, Michael A. Forbes, and Nicholas Spooner. *A Zero Knowledge Sumcheck and its Applications*. Cryptology ePrint Archive, Paper 2017/305. 2017. URL: https://eprint.iacr.org/2017/305.

[CFW26] Alessandro Chiesa, Giacomo Fenzi, and Guy Weissenberg. *Zero-Knowledge IOPPs for Constrained Interleaved Codes*. Cryptology ePrint Archive, Paper 2026/391. 2026. URL: https://eprint.iacr.org/2026/391.

[DG25] Benjamin E. Diamond and Angus Gruen. "Proximity Gaps in Interleaved Codes". In: *IACR Communications in Cryptology* 1.4 (Jan. 13, 2025). ISSN: 3006-5496. DOI: 10.62056/a0ljbkrz.

[DHRR26] Rahul Dalal, Tamir Hemo, Eugene Rabinovich, and Ron Rothblum. *VEIL: Lightweight Zero-Knowledge for Hash-Based Multilinear Proof Systems*. Cryptology ePrint Archive, Paper 2026/683. 2026. URL: https://eprint.iacr.org/2026/683.

[Dia25] Benjamin E. Diamond. *Zero-Knowledge Polynomial Commitment in Binary Fields*. Cryptology ePrint Archive, Paper 2025/1015. 2025. URL: https://eprint.iacr.org/2025/1015.

[DP24] Benjamin E. Diamond and Jim Posen. *Polylogarithmic Proofs for Multilinears over Binary Towers*. Cryptology ePrint Archive, Paper 2024/504. 2024. URL: https://eprint.iacr.org/2024/504.

[DP25] Benjamin E. Diamond and Jim Posen. "Succinct Arguments over Towers of Binary Fields". In: *Advances in Cryptology – EUROCRYPT 2025*. Ed. by Serge Fehr and Pierre-Alain Fouque. Cham: Springer Nature Switzerland, 2025, pp. 93–122. ISBN: 978-3-031-91134-7.

[DT24] Quang Dao and Justin Thaler. *Constraint-Packing and the Sum-Check Protocol over Binary Tower Fields*. Cryptology ePrint Archive, Paper 2024/1038. 2024. URL: https://eprint.iacr.org/2024/1038.

[Fs24] Matteo Frigo and abhi shelat. *Anonymous credentials from ECDSA*. Cryptology ePrint Archive, Paper 2024/2010. 2024. URL: https://eprint.iacr.org/2024/2010.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. "Delegating Computation: Interactive Proofs for Muggles". In: *Journal of the ACM* 62.4 (Sept. 2015). ISSN: 0004-5411. DOI: 10.1145/2699436.

[Gru24] Angus Gruen. *Some Improvements for the PIOP for ZeroCheck*. Cryptology ePrint Archive, Paper 2024/108. 2024. URL: https://eprint.iacr.org/2024/108.

[Hab24] Ulrich Haböck. *Basefold in the List Decoding Regime*. Cryptology ePrint Archive, Paper 2024/1571. 2024. URL: https://eprint.iacr.org/2024/1571.

[HMH] Adrian Hamelink, Andrew Milson, and Ulrich Haböck. *ZK13: Lifted FRI: A uniform multi-domain polynomial commitment scheme*. YouTube. URL: https://www.youtube.com/watch?v=p6Z3WjRcZD0.

[Hu+25] Yuncong Hu, Chongrong Li, Zhi Qiu, Tiancheng Xie, Yue Ying, Jiaheng Zhang, and Zhenfei Zhang. *GKR for Boolean Circuits with Sub-linear RAM Operations*. Cryptology ePrint Archive, Paper 2025/717. 2025. URL: https://eprint.iacr.org/2025/717.

[LCH14] Sian-Jheng Lin, Wei-Ho Chung, and Yunghsiang S. Han. "Novel Polynomial Basis and Its Application to Reed–Solomon Erasure Codes". In: *IEEE 55th Annual Symposium on Foundations of Computer Science*. 2014, pp. 316–325. DOI: 10.1109/FOCS.2014.41.

[LFKN92] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. "Algebraic Methods for Interactive Proof Systems". In: *Journal of the ACM* 39.4 (Oct. 1992), pp. 859–868. DOI: 10.1145/146585.146605. URL: https://doi.org/10.1145/146585.146605.

[Set20] Srinath Setty. "Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup". In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Cham: Springer International Publishing, 2020, pp. 704–737. ISBN: 978-3-030-56877-1. DOI: 10.1007/978-3-030-56877-1_25.

[Tha22] Justin Thaler. *Proofs, Arguments and Zero-Knowledge*. Vol. 4. Foundations and Trends in Privacy and Security 2–4. now publishers, 2022.

[Xie+19] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. "Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation". In: *Advances in Cryptology – CRYPTO 2019*. Berlin, Heidelberg: Springer-Verlag, 2019, pp. 733–764. ISBN: 978-3-030-26953-1. DOI: 10.1007/978-3-030-26954-8_24.

[ZCF24] Hadas Zeilberger, Binyi Chen, and Ben Fisch. "BaseFold: Efficient Field-Agnostic Polynomial Commitment Schemes from Foldable Codes". In: *Advances in Cryptology – CRYPTO 2024*. Berlin, Heidelberg: Springer-Verlag, 2024, pp. 138–169. ISBN: 978-3-031-68402-9. DOI: 10.1007/978-3-031-68403-6_5.