Sum-check Is All You Need: An Opinionated Survey on Fast Provers in SNARK Design

Justin Thaler
Georgetown University and a16z crypto research

Abstract

SNARKs work by having a prover commit to a witness and then prove that the committed witness is valid. The prover's work is dominated by two tasks: (i) committing to data and (ii) proving that the committed data is well-formed. The central thesis of this survey is that fast SNARKs minimize both costs by using the sum-check protocol.

But not all uses of sum-check are equally effective. The fastest SNARKs invoke sum-check in highly sophisticated ways, exploiting repeated structure in computation to aggressively minimize commitment costs and prover work. I survey the key ideas that enable this: batch evaluation arguments, read/write memory checking, virtual polynomials, sparse sum-checks, and small-value preservation. These techniques unlock the full potential of the sum-check protocol as a foundation for fast SNARK proving.

1 Introduction

A SNARK (succinct non-interactive argument of knowledge) allows an untrusted prover to convince a verifier that it correctly executed some computation on a witness, or equivalently that it knows data satisfying a certain property. The naive way to do this is for the prover to send the full witness, and for the verifier to directly run the checker. This trivial proof system is secure, but not succinct.

Succinctness means the proof should be shorter than the trivial proof and faster to verify than re-running the checker on the full witness.

Across most constructions, SNARK verifiers are already very efficient. Proofs range from under a kilobyte to a few hundred kilobytes (no matter how big the statement being proven), and verification typically takes on the order of milliseconds. In particular, even the largest SNARK proofs are much smaller than the witness for large enough computations. As a result, the key bottleneck for applying SNARKs in practice is prover time, not proof size or verifier cost.

SNARK proving in a nutshell. Modern SNARKs operate in two stages:

- The prover *commits* to a large dataset (e.g., a witness).
- Then it proves that this committed data is well-formed, i.e., that the witness checker would accept the committed witness.

In practice, the prover often commits to more than just the bare witness—extra data called untrusted advice is included to make the checking procedure more efficient. All of this committed data is *claimed* to represent a valid execution of the witness-checking procedure, and the SNARK must not only commit to the data but also prove that claim is correct.

This "commit-then-prove-well-formed" structure introduces two major costs for the prover:

1. **Cryptography is expensive.** Committing involves operations such as hashing, encoding with an error-correcting code, or cryptographic group operations. These take a lot of time when applied to large vectors.

2. **More committed data means more checking.** Every object the prover commits to must later be proven well-formed, which further increases the proving burden.

Hence the high-level lesson: *commit to as little data as possible.* Not zero—there's a sweet spot. This is for two reasons. First, without any cryptography (i.e., using information-theoretically secure interactive proofs), it's impossible (under standard complexity-theoretic conjectures) to achieve proofs shorter than the bare witness itself [GH98, GVW02]. Second, as mentioned earlier, without committing to a fair amount of untrusted advice, checking procedures are (in general) expensive when expressed in a format that SNARK machinery can handle efficiently. For example, for an operation like division (i.e., "compute a/b"), it's extremely expensive to prove one ran a division algorithm correctly step-by-step (even though real CPUs perform such operations very quickly). But it's easy to prove the claimed result q is correct if the prover also commits to a remainder R and shows that $a = q \cdot b + R$.

Enter sum-check. What is the best way to minimize commitment costs? One of the key design principles in this survey is to treat *interaction as a resource*: use it to minimize commitments and prover work, and remove it just once, via Fiat-Shamir, at the very end.[1] Section 1.1 elaborates on this viewpoint and why popular SNARKs for years failed to fully embrace this principle.

With this principle in mind, the sum-check protocol is a remarkably effective tool for offloading work onto an untrusted prover while keeping the proof succinct. The sum-check protocol as a standalone object is information-theoretically secure, i.e., it uses no cryptography at all. In place of cryptography, it uses interaction and randomness. Fiat-Shamir can then be used to remove the interaction with minimal overhead.

The performance of SNARK provers follows a clear hierarchy:

- **Slowest:** SNARKs that do not use the sum-check protocol at all.

- **Faster:** SNARKs that invoke sum-check, but fail to fully exploit repeated structure in the computation.

- **Fastest:** SNARKs that combine sum-check with techniques that aggressively exploit repeated structure to minimize commitment costs and overall prover overhead. As we will see, these techniques include batch evaluation arguments, polynomial virtualization, small-value preservation, and more.

Happily, the fastest SNARKs also turn out to be the simplest.

[1]The Fiat-Shamir transformation [FS86] is a technique for taking an interactive protocol and rendering it non-interactive. The rough idea is to have the prover itself derive each verifier challenge by applying a cryptographic hash function to all messages sent by the prover up to that point in the protocol.

1.1 SNARK history: from IPs to PCPs and back again

Interactive proofs were introduced by Goldwasser, Micali, and Rackoff (GMR) in the 1980s. At the time, it was widely believed that interactive proofs with polynomial-time verifiers could prove only slightly more than **NP**. This changed with the work of Lund, Fortnow, Karloff, and Nisan (LFKN) in 1990 [LFKN90], which introduced the sum-check protocol and used it to show that #SAT—the problem of counting satisfying assignments to a Boolean formula—has an interactive proof with an efficient verifier. This soon led to the celebrated result **IP** = **PSPACE** [Sha92].

Shortly thereafter, the community shifted toward non-interactive proof models, especially probabilistically checkable proofs (PCPs). These allowed the verifier to spot-check a proof by reading only a few bits. While it is possible to construct a polynomial-size PCP from the sum-check protocol, obtaining *quasilinear* size PCPs requires different techniques based on univariate polynomials and quotienting—topics we return to in Section 3.1.1.

These univariate techniques became the foundation for the first SNARKs. Kilian showed how to turn a PCP into a succinct interactive argument by Merkle-committing to the PCP string, and Micali proposed using Fiat–Shamir to remove interaction. This led to a prevailing belief that PCPs—or more precisely, univariate polynomials and quotienting—were the right starting point for SNARK design.

But this compilation path is indirect and inefficient. The interaction in sum-check is first removed to build a PCP, then reintroduced via Kilian-style transformations, only to be removed again via Fiat–Shamir. If you're going to use Fiat–Shamir, it's better to apply it just once—directly to a well-structured interactive proof.

The first systems to show that SNARKs can be built this way—by applying a polynomial commitment scheme to a sum-check-based interactive proof—were vSQL [ZGK+17] and Hyrax [WTS+18] in 2017. Both systems compiled the GKR protocol and its refinements [GKR08, CMT12, Tha13] into SNARKs by committing to polynomials and applying Fiat–Shamir. However, they were somewhat limited in the kinds of circuits they supported, relying on low-depth layered circuits with structured wiring predicates. Other systems like Spartan and Clover [Set20, BTVW14] addressed these limitations and demonstrated that sum-check-based SNARKs could support arbitrary circuits.

Still, the dominant approach in both theory and practice remained based on univariate encodings and quotienting. This was partly due to the appeal of very short proofs (e.g., Groth16 [Gro16]), and partly due to limited awareness of sum-check-based systems and their performance. That began to change in earnest in 2023, with systems like Lasso [STW24] and Jolt [AST24]. These systems clarified the extent to which combining sum-check with notions such as batch evaluation arguments, memory checking, and small-value preservation can yield general-purpose SNARKs that are both simpler and faster than those built on univariate polynomials.

The modern perspective, developed in this survey, is not that sum-check solves all performance issues when applied naively. Rather, there is a spectrum of sophistication in how the protocol is used. By exploiting repeated structure in real computations, sum-check enables SNARKs that can scale to the needs of real-world infrastructure.

2 Preliminaries

Let $\mathbb{F}$ be a finite field. In most SNARKs, the size of $\mathbb{F}$, denoted $|\mathbb{F}|$, is between 2^{128} and 2^{256}.[2] For $n \in \mathbb{N}$, write $\{0,1\}^n$ for the Boolean hypercube of n-bit strings. Sum-check-based SNARKs consider functions $f : \{0,1\}^n \to \mathbb{F}$, and reason about their *low-degree extensions*. These are low-degree polynomials that effectively extend the domain of f from $\{0,1\}^n$ to $\mathbb{F}^n$. Moreover, the extension procedure is *distance-amplifying*: if two functions $f, g \colon \{0,1\}^n \to \mathbb{F}$ differ at even a single input, their low-degree extensions will differ almost everywhere. Below we present the details needed for this survey.

Multilinear polynomials. An n-variate polynomial $p(x_1, \ldots, x_n)$ over $\mathbb{F}$ is multilinear if p has degree at most one in each variable. For example, $2x_1x_2 + x_3$ is multilinear, but $2x_1^2 + x_2$ is not.

Fact 1. *A multilinear polynomial is uniquely specified by its evaluations over domain $\{0,1\}^n$. Hence, if p and q are two multilinear polynomials satisfying $p(x) = q(x)$ for all $x \in \{0,1\}^n$, then $p = q$.*

Multilinear extension. Any function $f : \{0,1\}^n \to \mathbb{F}$ has a unique *multilinear extension* $\tilde{f} : \mathbb{F}^n \to \mathbb{F}$, i.e., a polynomial of degree at most one in each variable that agrees with f on $\{0,1\}^n$. We denote this extension as $\tilde{f}$ or $\mathsf{MLE}(f)$.

Equality testing. We write $\mathsf{eq}(x, y)$ for the Boolean equality predicate on $\{0,1\}^m$, which returns 1 if $x = y$ and 0 otherwise. Its multilinear extension $\widetilde{\mathsf{eq}}(x, y) : \mathbb{F}^m \times \mathbb{F}^m \to \mathbb{F}$ is defined as

$$\widetilde{\mathsf{eq}}(x, y) := \prod_{i=1}^{m} \left((1 - x_i)(1 - y_i) + x_i y_i\right)$$

This is a multilinear polynomial that equals 1 if and only if $x = y$ in $\{0,1\}^m$, and 0 otherwise.

Multilinear extension of a vector. Let $a \in \mathbb{F}^N$ for $N = 2^n$. We view a as a function $a : \{0,1\}^n \to \mathbb{F}$, where $a(x)$ denotes the entry of a indexed by the bit-string $x \in \{0,1\}^n$ in the natural way (see Figure 1 for an illustration). The *multilinear extension* $\widetilde{a} : \mathbb{F}^n \to \mathbb{F}$ of a is the unique multilinear polynomial that agrees with a on $\{0,1\}^n$. That is, $\widetilde{a}$ is the unique multilinear polynomial in each variable such that $\widetilde{a}(x) = a(x)$ for all $x \in \{0,1\}^n$. An explicit expression for $\widetilde{a}$ is below; this expression is sometimes called *multilinear Lagrange interpolation*:

$$\widetilde{a}(r) = \sum_{x \in \{0,1\}^n} a(x) \cdot \widetilde{\mathsf{eq}}(r, x), \tag{1}$$

where $\widetilde{\mathsf{eq}}(r, x)$ is the multilinear extension of the equality function, as defined above.

How we denote multilinear polynomials in this survey. For clarity, throughout the remainder of the survey, a polynomial that is guaranteed to be multilinear will always have a tilde over it. Elsewhere in the survey, the letter g denotes a multivariate polynomial that is *not* multilinear, but is often the product of two or more multilinear polynomials (e.g., $g(x) = \widetilde{p}(x) \cdot \widetilde{q}(x)$ where $\widetilde{p}$ and $\widetilde{q}$ are both multilinear).

[2] Some succinct arguments make partial use of fields that are smaller than 128 bits in size. However, even in these arguments, random verifier challenges are typically drawn from an extension field of size at least 2^{128} to ensure adequate security.

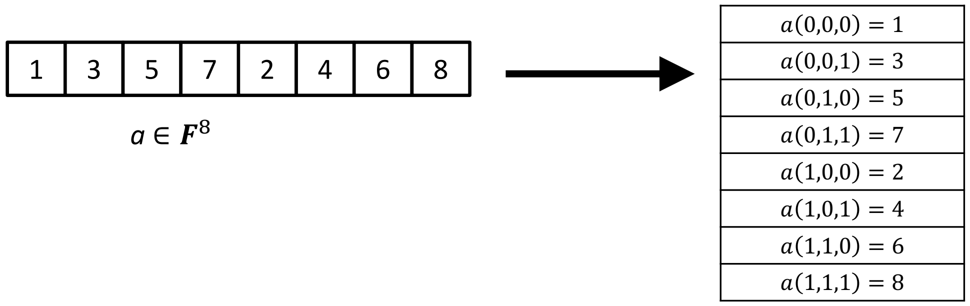


Figure 1: A vector $a \in \mathbb{F}^8$, interpreted as a function from $\{0,1\}^3$ to $\mathbb{F}$.

Hadamard product. Let $a, b \in \mathbb{F}^N$ be two vectors. Their *Hadamard product* (also called the entrywise product) is the vector $a \circ b \in \mathbb{F}^N$ defined by:

$$(a \circ b)_i := a_i \cdot b_i \quad \text{for all } i = 1, \ldots, N.$$

Univariate low-degree extension. If $a \in \mathbb{F}^N$ is a vector and $H = \{\alpha_1, \ldots, \alpha_N\} \subseteq \mathbb{F}$ is an evaluation domain, we define the unique univariate polynomial $\hat{a} \in \mathbb{F}[X]$ of degree at most $N-1$ satisfying $\hat{a}(\alpha_i) = a_i$ for all $i = 1, \ldots, N$. This is the *univariate low-degree extension* of a over H.

A basic result from algebra is the Factor Theorem.

Theorem 2 (Factor Theorem)**.** *Let $f \in \mathbb{F}[X]$ be a univariate polynomial. Then $f(a) = 0$ if and only if $(X - a)$ divides $f(X)$. In particular, if f has degree at most d and is not identically zero, then it has at most d distinct roots in $\mathbb{F}$.*

An immediate corollary of the factor theorem is that two distinct degree d polynomials can agree at most d points:

Theorem 3 (Bounded agreement for univariate polynomials)**.** *Let $f, g \in \mathbb{F}[X]$ be distinct univariate polynomials of degree at most d. Then $f(x) = g(x)$ for at most d values $x \in \mathbb{F}$.*

The famous Schwartz-Zippel lemma extends this corollary to the multivariate setting. To state it, we must introduce the concept of the *total degree* of a multivariate polynomial. The *total degree* of a multivariate polynomial $f \in \mathbb{F}[X_1, \ldots, X_n]$ is the maximum, over all monomials with nonzero coefficient in f, of the sum of the degrees in each variable. For example, the total degree of $X_1^2 X_2 + X_3^5$ is 5.

Theorem 4 (Schwartz–Zippel Lemma)**.** *The Schwartz–Zippel lemma generalizes the bounded agreement property to the multivariate setting. Let $f \in \mathbb{F}[X_1, \ldots, X_n]$ be a nonzero n-variate polynomial of total degree d. Then for any finite subset $S \subseteq \mathbb{F}$, the probability that f evaluates to 0 at a random point from S^n is at most $d/|S|$.*

In this survey, we generally regard a multiplication in a finite field $\mathbb{F}$ as taking one unit of time. Of course, in practice how long it takes depends on how big the field is. We discuss this issue further later in the survey (Section 7).

3 Polynomial IOPs and Polynomial Commitments

Modern SNARK constructions combine two components:

- A **polynomial IOP** (PIOP): an information-theoretically secure interactive proof whose first prover message is a large polynomial encoding the witness (the polynomial's size is comparable to the runtime of the witness-checking procedure). The polynomial is too big for the verifier to read a complete description of it; instead the verifier is only permitted to query it at a single random point chosen during the interaction. After sending this polynomial, the prover and verifier run a standard interactive proof to show that the encoded witness is valid — this is the two-stage paradigm from the introduction.

- A **polynomial commitment scheme** (PCS): a cryptographic primitive that can be used to render the PIOP into a succinct argument. A PCS lets the prover succinctly commit to the large polynomial and later open it at any requested point, providing a short proof that the revealed evaluation is consistent with the commitment (and enabling Fiat–Shamir to remove interaction).

Many PCS constructions are themselves best viewed as PIOPs wrapped in cryptography. For example, KZG corresponds to the quotienting PIOP, carried out "in the exponent" using a structured reference string. Similarly, Bulletproofs/IPA and Dory correspond to the sum-check protocol applied in the exponent as well. (See Section 6 for details.)

PIOPs typically fall into two classes: those based on univariate polynomials and quotienting, and those based on multilinear polynomials and the sum-check protocol. Similarly, practical PCSes can be grouped into four broad families:

- KZG and its variants, which use pairings and require a trusted setup;
- Group-based transparent schemes, such as Hyrax, Bulletproofs/IPA, and Dory;
- Hashing-based schemes like FRI [BBHR18], Ligero-PCS [AHIV17], Brakedown [GLS+23], and WHIR [ACFY25];
- Lattice-based schemes, which are still evolving but generally follow the structure of the group-based transparent schemes (while being post-quantum and offering more flexibility in choice of finite field used).

Generally speaking, any PIOP can be paired with any PCS of the same type: univariate PIOPs require univariate PCSes, and multilinear PIOPs require multilinear PCSes.[3]

3.1 The Two Classes of PIOPs

An illustrative task: proving product constraints. To illustrate the two classes of PIOPs, we'll consider the following task. Suppose we want to prove $a \circ b = c$ for three vectors $a, b, c \in \mathbb{F}^N$. Here, $\circ$ denotes entrywise product, i.e., we want to check that $a_i \cdot b_i = c_i$ for all $i = 1, \ldots, N$.

[3]This survey does not address zero-knowledge (ZK) in depth. However, sum-check-based SNARKs can generally be made ZK with minimal overhead when instantiated with curve-based PCSes [WTS+18, Lee21, KS24a]. The same applies to some hashing-based PCSes [AHIV17], though for others the extent to which ZK can be achieved without incurring additional prover cost remains an open question [Blo25, Dia25].

This task may seem simple, but it captures "half the logic" of many SNARKs for circuit satisfiability. More precisely, suppose we are given an arithmetic circuit consisting solely of multiplication gates. Let c_i denote the value assigned to gate i, and let a_i and b_i denote the values assigned to its left and right in-neighbors, respectively. Then checking that $a_i \cdot b_i = c_i$ is equivalent to verifying that gate i computes the correct output given the values of its inputs.

In SNARKs for circuit satisfiability, checking that $a \circ b = c$ is only half the battle: we must also verify that the values a_i and b_i are in fact the values assigned (via other entries in c) to the actual in-neighbors of gate i. In other words, we must check "wiring constraints" that ensure a and b are derived from c according to the structure of the circuit. This second step is more complicated, as it inherently involves *sparse polynomials*. We return to it later in this survey (see Sections 4.2.1 and 5.4).

3.1.1 The Quotienting PIOP

Let $H = \{\alpha_1, \dots, \alpha_N\} \subseteq \mathbb{F}$ be any set of size N. Recall that there are unique univariate polynomials $\hat{a}$, $\hat{b}$, $\hat{c}$ of degree at most $N-1$ such that $\hat{a}(\alpha_i) = a_i$ for all $i = 1, \dots, N$, and similarly $\hat{b}(\alpha_i) = b_i$ and $\hat{c}(\alpha_i) = c_i$ for $i = 1, \dots, N$. These are called the *univariate low-degree extensions* of the vectors a, b, and c. Suppose that $\hat{a}$, $\hat{b}$, and $\hat{c}$ have each been committed with our favorite univariate PCS.

Let $Z_H(X) = \prod_{i=1}^{N}(X - \alpha_i)$ be the *vanishing polynomial* of H, i.e., the unique polynomial of degree at most N that maps all elements of H to 0 (and no other elements of $\mathbb{F}$ to zero). An easy consequence of the Factor Theorem (Theorem 2) is that $\hat{a}(\alpha_i) \cdot \hat{b}(\alpha_i) = \hat{c}(\alpha_i)$ for all $\alpha_i \in H$ if and only if the polynomial $\hat{a}(X) \cdot \hat{b}(X)$ is divisible by $Z_H(X)$.

The prover sends (or commits to) a quotient polynomial $q(X)$ satisfying:

$$\hat{a}(X) \cdot \hat{b}(X) - \hat{c}(X) = q(X) \cdot Z_H(X). \tag{2}$$

The verifier samples $r \leftarrow \mathbb{F}$ and checks the identity at r, i.e., the verifier checks that

$$\hat{a}(r) \cdot \hat{b}(r) - \hat{c}(r) = q(r) \cdot Z_H(r).$$

A popular choice for H in practice is the set of N'th roots of unity in $\mathbb{F}$, in which case $Z_H(X) = X^N - 1$, and hence the verifier can compute $Z_H(r)$ on its own in $O(\log N)$ time. The verifier can get $\hat{a}(r)$, $\hat{b}(r)$, $\hat{c}(r)$, and $q(r)$ from the commitments to these four polynomials.

Soundness holds because if Equation (2) is false, then by Theorem 3, the probability that it holds at a random point is at most $d/|\mathbb{F}|$, where $d = O(N)$ is an upper bound on the degrees of $\hat{a}(X) \cdot \hat{b}(X) - \hat{c}(X)$ and $q(X) \cdot Z_H(X)$.

3.2 The Sum-Check Protocol

Let $g : \mathbb{F}^n \to \mathbb{F}$ be an n-variate polynomial with degree at most d in each variable (for simplicity let us think of $d = 2$ henceforth). Often, g will be a product of d multilinear polynomials. The prover's goal is to prove that

$$\sum_{x \in \{0,1\}^n} g(x) \tag{3}$$

is equal to some claimed value C_1.

From the verifier's perspective, this is a hard task: summing 2^n evaluations of g is expensive. For the verifier, the sum-check protocol reduces this to the much simpler task of evaluating g at a single random point $r \in \mathbb{F}^n$.

Verifier view. In each round i, the prover sends a degree-d univariate polynomial $s_i(X)$ that purports to equal the partial sum

$$g_i(X) := \sum_{x_{i+1},\ldots,x_n \in \{0,1\}} g(r_1, \ldots, r_{i-1}, X, x_{i+1}, \ldots, x_n). \tag{4}$$

This is a degree-d univariate polynomial, which can be specified by sending, e.g., $d+1$ coefficients in the standard monomial basis, or $d+1$ evaluations (e.g., sending $s_i(0)$, $s_i(1), \ldots, s_i(d)$).

In round 1, the verifier checks that $C_1 = s_1(0) + s_1(1)$, and in each round $i > 1$ the verifier checks that

$$s_{i-1}(r_{i-1}) = s_i(0) + s_i(1),$$

and sends a new random challenge $r_i \in \mathbb{F}$. At the end of n rounds, the verifier checks that the final value $s_n(r_n)$ matches $g(r_1, \ldots, r_n)$.

Description of Sum-Check Protocol.

- Let g be an n-variate polynomial over $\mathbb{F}$ of degree at most d in each variable.
- At the start of the protocol, the prover sends a value C_1 claimed to equal the value defined in Equation (3).
- In the first round, $\mathcal{P}$ sends the univariate polynomial $g_1(X_1)$ claimed to equal

  $$\sum_{(x_2,\ldots,x_n) \in \{0,1\}^{n-1}} g(X_1, x_2, \ldots, x_n).$$

  $\mathcal{V}$ checks that

  $$C_1 = s_1(0) + s_1(1),$$

  and that g_1 is a univariate polynomial of degree at most d, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_1 \in \mathbb{F}$, and sends r_1 to $\mathcal{P}$.
- In the jth round, for $1 < j < n$, $\mathcal{P}$ sends to $\mathcal{V}$ a univariate polynomial $s_j(X_j)$ claimed to equal

  $$\sum_{(x_{j+1},\ldots,x_n) \in \{0,1\}^{n-j}} g(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_n).$$

  $\mathcal{V}$ checks that s_j is a univariate polynomial of degree at most d, and that $s_{j-1}(r_{j-1}) = s_j(0) + s_j(1)$, rejecting if not.
- $\mathcal{V}$ chooses a random element $r_j \in \mathbb{F}$, and sends r_j to $\mathcal{P}$.
- In round n, $\mathcal{P}$ sends to $\mathcal{V}$ a univariate polynomial $s_n(X_n)$ claimed to equal

  $$g(r_1, \ldots, r_{n-1}, X_n).$$

  $\mathcal{V}$ checks that s_n is a univariate polynomial of degree at most d, rejecting if not, and also checks that $s_{n-1}(r_{n-1}) = s_n(0) + s_n(1)$.
- $\mathcal{V}$ chooses a random element $r_n \in \mathbb{F}$ and evaluates $g(r_1, \ldots, r_n)$ with a single oracle query to g. $\mathcal{V}$ checks that $s_n(r_n) = g(r_1, \ldots, r_n)$, rejecting if not.
- If $\mathcal{V}$ has not yet rejected, $\mathcal{V}$ halts and accepts.

Soundness sketch. Recall that in each round i of the sum-check protocol, $s_i(X)$ denotes the polynomial *actually* sent by the prover, while g_i (defined in Equation (4)) denotes the polynomial that the *honest* prover sends.

If the prover begins with a false claim, i.e., $C_1 \neq \sum_{x \in \{0,1\}^n} g(x)$, then it must be the case that the prover's *actual* message $s_1(X)$ in round one does not equal the univariate polynomial it is claimed to equal, namely $g_1(X) = \sum_{x' \in \{0,1\}^{n-1}} g(X, x')$ (as otherwise the prover would fail the check that $C_1 = s_1(0) + s_1(1)$).

In this case, the probability (over the random choice of $r_1 \in \mathbb{F}$) that $s_1(r_1) = g_1(r_1)$ is at most $d/|\mathbb{F}|$. If $s_1(r_1) \neq g_1(r_1)$, then the prover is left to prove a false claim in the next round. This analysis can then be repeated round-over-round, yielding that with probability at least $1 - dn/|\mathbb{F}|$, the prover is still left with a false claim after all n rounds are done. In this event, the prover fails the verifier's final-round check that $s_n(r_n) = g(r_1, \ldots, r_n)$.

Prover cost. In round i, the prover must evaluate $\tilde{g}$ at all points of the form

$$(r_1, \ldots, r_{i-1}, t, x_{i+1}, \ldots, x_n), \quad \text{for } t \in \{0, 1, \ldots, d\},\ (x_{i+1}, \ldots, x_n) \in \{0,1\}^{n-i}. \tag{5}$$

So the prover performs 2^{n-i} evaluations of $\tilde{g}$ per round. This leads to a clean recursive structure where prover work halves each round. As we'll see, this is the basis for fast prover algorithms.

Proof size and verifier time. A degree-d univariate polynomial is sent in each of the n rounds. Since any degree d polynomial can be specified by $d+1$ coefficients (or evaluations), the total communication is $(d+1)n$ field elements.[4,5] Verifier time is $O(d)$ field operations per round, and hence $O(dn)$ field operations in total. Note that if $N = 2^n$ is the number of terms in the sum being proved, then the verifier time and proof size is just logarithmic in N. Concretely, for $N = 2^{30}$, $d = 2$, and working over a 256-bit field, the proof is a few KB.

3.2.1 Application: Proving $a \circ b = c$ via sum-check

As in Section 3.1.1, suppose the prover has committed to three vectors $a, b, c \in \mathbb{F}^N$, and we want to check that $a_i \cdot b_i = c_i$ for all $i = 1, \ldots, N$, where $N = 2^n$. Let $\widetilde{a}, \widetilde{b}, \widetilde{c} : \mathbb{F}^n \to \mathbb{F}$ denote the multilinear extensions of these vectors, and define

$$g(x) := \widetilde{a}(x) \cdot \widetilde{b}(x) - \widetilde{c}(x).$$

Our goal is to verify that $g(x) = 0$ for all $x \in \{0,1\}^n$. We can accomplish this by applying the sum-check protocol to compute

$$\sum_{x \in \{0,1\}^n} \widetilde{\mathsf{eq}}(r, x) \cdot g(x) \tag{6}$$

where $r \in \mathbb{F}^n$ is chosen at random by the verifier, and having the verifier accept if Expression (6) equals 0 and reject otherwise.

Clearly, Expression (6) equals 0 if $g(x) = 0$ for all $x \in \{0,1\}^n$, so this protocol is complete. We now explain that if $g(x) \neq 0$ for even one $x \in \{0,1\}^n$, then Expression (6) is very unlikely to equal 0, and hence the protocol is sound.

[4] There are simple techniques that slightly reduce this proof size, see Section 6.3 for details.

[5] The sum-check verifier sends one field element to the prover per round. When the Fiat-Shamir transformation is applied to render the protocol non-interactive, there is no verifier-to-prover to communication.

Intuitively, expression (6) behaves like a *random linear combination (RLC)* of the values $g(x)$ as x ranges over $\{0,1\}^n$. When $r \in \mathbb{F}^n$ is random, each of the N individual values $\widetilde{\mathsf{eq}}(r,x)$ as x ranges over $\{0,1\}^n$ is uniformly distributed in $\mathbb{F}$. The N values are *not* independent, but we show next that they are nonetheless "independent enough" for purposes of proving that $a \circ b = c$.

Theorem 5. *If $g(x) \neq 0$ for even a single $x \in \{0,1\}^n$, then the probability of the random choice of $r \in \mathbb{F}^n$ that Expression (6) equals 0 is at most $n/|\mathbb{F}|$.*

Proof. Let us refer to Expression (6) as $\widetilde{q}(r)$. As the tilde notation suggests, $\widetilde{q}$ is easily seen to be a multilinear polynomial in r, and it agrees with $g(r)$ whenever $r \in \{0,1\}^n$, since $\widetilde{\mathsf{eq}}(r,x)$ is the multilinear extension of the equality predicate eq that satisfies $\widetilde{\mathsf{eq}}(r,x) = 1$ if and only if $x = r$ (when $r \in \{0,1\}^n$). Hence, $\widetilde{q}$ is the zero polynomial if $g(x) = 0$ for all $x \in \{0,1\}^n$, and otherwise $\widetilde{q}$ is a non-zero multilinear polynomial. By the Schwartz-Zippel lemma, the probability over a random input $r \in \mathbb{F}^n$ that a non-zero n-variate multilinear polynomial evaluates to 0 is at most $n/|\mathbb{F}|$. □

4 Fast Sum-Check Proving Algorithms

4.1 Dense Sums in Linear Time

Suppose we are applying the sum-check protocol to a polynomial

$$g(x_1, \ldots, x_n) = \widetilde{p}(x_1, \ldots, x_n) \cdot \widetilde{q}(x_1, \ldots, x_n) \tag{7}$$

where both $\widetilde{p}$ and $\widetilde{q}$ are multilinear. Observe that g is *not* multilinear, as it has degree up to two in each variable. (The algorithm applies to the product of any number of multilinear polynomials, but we focus on the case of two for clarity.)

In round i, the prover must evaluate g at all points in the form of Expression (5). Let $N = 2^n$ denote the number of terms in the original sum (Expression (3)) and note that $N/2^i$ evaluations of g are needed by the prover in round i. Across all n rounds, this is $O(N + N/2 + N/4 + \cdots + 1) = O(N)$ evaluations in total.

Clearly to obtain the evaluations of g in round i at all points of the form of Expression (5) in $O(N/2^i)$ time, it suffices for the prover to evaluate $\widetilde{p}$ and $\widetilde{q}$ individually at points of that form. This is the task to which we now turn.

Fact 6. *Let $\widetilde{p}(x_1, \ldots, x_n)$ be a multilinear polynomial. Then for any $r_1 \in \mathbb{F}$,*

$$\widetilde{p}(r_1, x_2, \ldots, x_n) = (1 - r_1) \cdot \widetilde{p}(0, x_2, \ldots, x_n) + r_1 \cdot \widetilde{p}(1, x_2, \ldots, x_n). \tag{8}$$

Proof. By Fact 1, $\{0,1\}^n$ is an interpolating set for multilinear polynomials. So it suffices to confirm that both the left hand side and right hand side of Equation (8) are multilinear polynomials in $(r_1, x_2, \ldots, x_n)$ and that they agree whenever $(r_1, x_2, \ldots, x_n) \in \{0,1\}^n$. This is easily confirmed. Indeed, if $r_1 = 0$ then the right hand side evaluates to $\widetilde{p}(0, x_2, \ldots, x_n)$ and if $r_1 = 1$, then the right hand side evaluates to $\widetilde{p}(1, x_2, \ldots, x_n)$. □

Linear-time prover algorithm. At the start of the protocol, the prover stores all evaluations of $\widetilde{p}$ and $\widetilde{q}$ over inputs in $\{0,1\}^n$, in arrays A and B of size $N = 2^n$ (we assume that initializing A and B to store these evaluations can be done in $O(N)$ time, and this is the case in all applications of the sum-check protocol with which we are concerned in this survey).

After the verifier chooses the random challenge $r_1 \in \mathbb{F}$ in round 1, the prover updates A and B to be size $N/2$ arrays respectively storing all evaluations of $\widetilde{p}$ and $\widetilde{q}$ at points of the form (r_1, x') with $x' \in \{0,1\}^{n-1}$. This is done by simply applying Fact 6 for each x', i.e., updating

$$A[x'] \leftarrow (1 - r_1) \cdot A[0, x'] + r_1 \cdot A[1, x'] = A[0, x'] + r_1 \cdot (A[1, x'] - A[0, x']),$$

$$B[x'] \leftarrow (1 - r_1) \cdot B[0, x'] + r_1 \cdot (B[1, x'] - B[0, x']).$$

The prover simply repeats this update procedure in each round. So at the end of round i, after the prover learns $r_i \in \mathbb{F}$, for each $x' \in \{0,1\}^{n-i}$ the prover updates:

$$A[x'] \leftarrow A[0, x'] + r_i \cdot (A[1, x'] - A[0, x']),$$

$$B[x'] \leftarrow B[0, x'] + r_i \cdot (B[1, x'] - B[0, x']).$$

In this way, in each round i, in $O(N/2^i)$ time, the prover is able to compute all necessary evaluations of $\widetilde{p}$ and $\widetilde{q}$.

4.2 Sparse Sums (and Streaming Proving)

Many sum-check applications involve sparse sums. This means that the number of terms $N = 2^n$ in the original sum $\sum_{x \in \{0,1\}^n} g(x)$ is huge, but most of the terms are 0. Say only T out of N terms are non-zero and call T the *sparsity* of the sum. Is it possible for the sum-check prover to perform $O(T)$ field operations instead of the $O(N)$ achieved in Section 4.1? Here, we show the answer is *yes*, so long as g satisfies some natural structural properties.

In fact something stronger is achievable. Fix any desired constant $c > 0$, and suppose we want to minimize the memory usage of the prover, e.g., by having it run in space $O(N^{1/c})$ (even if T is larger than $N^{1/c}$). Of course, this is only possible if all non-zero terms of the original sum can be iteratively generated by the prover in small space: if it takes linear-in-N space just to compute the original sum, then of course we can't hope for a prover that runs in space sublinear in N.

A clean setting to illustrate the ideas. Let us continue to assume per Expression (7) that

$$g(x) = \widetilde{p}(x) \cdot \widetilde{q}(x)$$

for multilinear polynomials $\widetilde{p}$ and $\widetilde{q}$, and that $\widetilde{p}(x) \neq 0$ for T values of $x \in \{0,1\}^n$. Fix $c = 2$, so that we are targeting prover space usage of $O(N^{1/2})$ and prover time of $O(T + N^{1/2})$. We pick $c = 2$ as a concrete example, but the algorithm below generalizes to larger values of c.

To cleanly illustrate the main ideas of the algorithm, suppose that on input $(i, j) \in \{0,1\}^{n/2} \times \{0,1\}^{n/2}$, we can write

$$\widetilde{q}(i, j) = \widetilde{f}(i) \cdot \widetilde{h}(j) \tag{9}$$

for multilinear polynomials $\widetilde{f}$ and $\widetilde{h}$. Intuitively, we are assuming $\widetilde{q}$ displays independence, or product structure: the inputs to $\widetilde{q}$ can be decomposed into two "chunks" i and j such that i and j contribute independently to the output of $\widetilde{q}$. This is *not* a toy scenario: this setting already captures important applications (see for example Section 4.2.1 below), and elicits the key ideas that generalize to much more complicated scenarios (see, for example, Section 5.1).

For any input $(i, j) \in \{0,1\}^{n/2} \times \{0,1\}^{n/2}$, we refer to i as the *prefix* and j as the *suffix*, and therefore call the efficient sum-check prover algorithm in this setting the *prefix-suffix sum-check proving algorithm* [NTZ25].

Prefix-Suffix Proving Algorithm Overview. It turns out this structure is enough to have the prover "focus on one chunk of variables at a time". That is, our prover algorithm will consist of two stages, one per chunk, where the prover pays time T to "initialize" each stage, but once initialized, the stage runs in time and space proportional to the size of the chunk, i.e., $2^{n/2} = O(\sqrt{N})$, rather than the size of the entire hypercube 2^n.

To initialize each stage, the prover has to make one streaming pass over the non-zero terms of the original sum, i.e., over all values $(\widetilde{p}(i,j), \widetilde{f}(i), \widetilde{h}(j))$ for $(i,j) \in \{0,1\}^n$ such that $\widetilde{p}(i,j) \neq 0$. Outside of these stage-initialization procedures, the prover does not have to refer back to the terms of the original sum at all. This is what we mean by a streaming prover[6] that uses $O(N^{1/2})$ space.

Stage 1. At the start of Stage 1, the prover initializes two arrays P and Q, both of size $\sqrt{N} = 2^{n/2}$. Each array can be initialized in $O(T + N^{1/2})$ time, where recall that T is the number of non-zero evaluations of $\widetilde{p}(x)$ as x ranges over $\{0,1\}^n$. For $i \in \{0,1\}^{n/2}$, $P[i]$ and $Q[i]$ are defined as follows:

$$P[i] = \sum_{j \in \{0,1\}^{n/2}} \widetilde{p}(i,j) \cdot \widetilde{h}(j). \tag{10}$$

$$Q[i] = \widetilde{f}(i). \tag{11}$$

We claim the prover's message in each of the first $n/2$ rounds of the sum-check protocol applied to compute

$$\sum_{x \in \{0,1\}^{\log N}} \widetilde{p}(x) \cdot \widetilde{q}(x) = \sum_{(i,j) \in \{0,1\}^{n/2} \times \{0,1\}^{n/2}} \widetilde{p}(i,j) \cdot \widetilde{f}(i) \cdot \widetilde{h}(j)$$

is identical to the corresponding prover message in the sum-check protocol applied to compute

$$\sum_{i \in \{0,1\}^{n/2}} \widetilde{P}(i) \cdot \widetilde{Q}(i). \tag{12}$$

Here, $\widetilde{P}$ and $\widetilde{Q}$ denote the multilinear extensions of the functions that maps $i \in \{0,1\}^{n/2}$ to $P[i]$ and $Q[i]$ respectively.

Indeed, if $r_1, \ldots, r_{k-1}$ denotes the random verifier challenges selected in rounds $1, \ldots, k-1$, then the honest sum-check prover's message in round k specifies the polynomial

$$\begin{aligned} &\sum_{x \in \{0,1\}^{n-k}} \widetilde{p}(r_1, \ldots, r_{k-1}, X, x) \cdot \widetilde{q}(r_1, \ldots, r_{k-1}, X, x) \\ &= \sum_{i' \in \{0,1\}^{n/2-k}} \sum_{j \in \{0,1\}^{n/2}} \widetilde{p}(r_1, \ldots, r_{k-1}, X, i', j) \cdot \widetilde{f}(r_1, \ldots, r_{k-1}, X, i') \cdot \widetilde{h}(j) \\ &= \sum_{i' \in \{0,1\}^{n/2-k}} \widetilde{Q}(r_1, \ldots, r_{k-1}, X, i') \cdot \widetilde{P}(r_1, \ldots, r_{k-1}, X, i'). \end{aligned} \tag{13}$$

Expression (13) is precisely the prover's message in round k of the sum-check protocol applied to compute Expression (12).

[6]A "streaming prover" refers to controlling prover memory usage without invoking SNARK recursion. There are also recursive techniques that can reduce prover memory, and several of the fastest such approaches also rely heavily on the sum-check protocol [KS24a, KS24b, AS24, KS22].

Accordingly, the prover's messages in rounds $1, \dots, n/2$ can be computed by applying the standard linear-time sum-check proving algorithm (Section 4.1) to the polynomial $\widetilde{P}(i) \cdot \widetilde{Q}(i)$. Once the arrays P and Q are initialized, this algorithm takes only $O(N^{1/2})$ time with no additional passes over the evaluations of p and q needed.

Stage 2. Let $\vec{r} = (r_1, \dots, r_{n/2})$ denote the random verifier challenges chosen over the course of the first $n/2$ sum-check rounds comprising Stage 1. At the start of Stage 2, the prover once again initializes two arrays P and Q of size $\sqrt{N}$, where now we define:

$$P[j] = \widetilde{p}(\vec{r}, j).$$
$$Q[j] = \widetilde{f}(\vec{r}) \cdot \widetilde{h}(j).$$

All entries of P and Q can be computed by the prover in $O(T + \sqrt{N})$ total time (for brevity, we omit the details of how to achieve this). It is easy to see that the prover's messages in the final $n/2$ rounds of the sum-check protocol applied to compute

$$\sum_{x \in \{0,1\}^{\log N}} \widetilde{p}(x) \cdot \widetilde{q}(x) = \sum_{(i,j) \in \{0,1\}^{n/2} \times \{0,1\}^{n/2}} \widetilde{p}(i,j) \cdot \widetilde{f}(i) \cdot \widetilde{h}(j)$$

are identical to the prover's messages in the sum-check protocol applied to compute

$$\sum_{j \in \{0,1\}^{n/2}} \widetilde{P}(j) \cdot \widetilde{Q}(j).$$

4.2.1 An application: Linear-time proving in the GKR protocol

To motivate the usefulness of "prefix-suffix structure" captured in Equation (9), consider a layered arithmetic circuit of depth d consisting exclusively of multiplication gates of fan-in two. Suppose for simplicity that all layers of the circuit have $S = 2^s$ gates, and number the layers from 1 to d, with layer d denoting the layer of output gates and layer 1 denoting the inputs to the circuit.

Let $V_k \colon \{0,1\}^s \to \mathbb{F}$ denote the function that takes as input the label of a gate at layer k and outputs its value. Let $\text{mult}_k(a,b,c) \colon \{0,1\}^{3s} \to \{0,1\}$ denote the function that takes as input three gate labels and outputs 1 if and only if gates b and c at layer $k-1$ are the in-neighbors of gate a at layer k. Then

$$\tilde{V}_k(r) = \sum_{i,j \in \{0,1\}^s} \widetilde{\text{mult}}_k(r,i,j) \cdot \widetilde{V}_{k-1}(i) \cdot \widetilde{V}_{k-1}(j). \tag{14}$$

As usual, to check that Equation (14) holds, we must check that the right hand side is multilinear in r and that it agrees with the left hand side whenever r is in $\{0,1\}^s$. This is indeed the case: the right hand side is clearly multilinear in r, and when r is in $\{0,1\}^s$, the right hand side is summing over all possible pairs of gates (i,j) at layer $k-1$, testing (via the term $\widetilde{\text{mult}}(r,i,j)$) whether i and j are indeed the in-neighbors of gate r, and if so, outputting the product of the values of those two gates, which is exactly the value of gate r at layer k.

The GKR protocol for circuit evaluation iterates over each layer k of the circuit, and applies the sum-check protocol to compute the right hand side of Equation (14). The prefix-suffix sum-check proving algorithm applies directly to this scenario, with $\widetilde{p}(i,j)$ equal to $\widetilde{\text{mult}}_k(r,i,j)$ and

$\widetilde{q}(i,j) = \widetilde{V}_{k-1}(i) \cdot \widetilde{V}_{k-1}(j)$. The sparsity of $\widetilde{p}$ in this setting is at most S, the number of gates at layer k of the circuit. That is, although there are S^2 inputs $(i,j) \in \{0,1\}^s \times \{0,1\}^s$, $\widetilde{p}(i,j) \neq 0$ for only at most S of these inputs.

Thus, the prefix-suffix sum-check proving algorithm (with $\widetilde{f}(i) = \widetilde{V}_{k-1}(i)$ and $\widetilde{h}(j) = \widetilde{V}_{k-1}(j)$) easily recovers the result, first shown in [XZZ+19], that the GKR protocol has a linear-time prover when applied to any layered arithmetic circuit. That is, the prover runs in time linear in the number of gates in the circuit.

Later, in Section 5.1, we sketch much more sophisticated applications of the prefix-suffix sum-check proving algorithm, in the context of the Shout batch evaluation argument and the Jolt zkVM.

Remark 7 (Small-value preservation and sum-check). While small-value preservation is most often discussed in the context of polynomial commitments (see Section 6.1), the sum-check protocol also benefits from it. Specifically, if the polynomials being summed—e.g., $g(x) = \widetilde{p}(x) \cdot \widetilde{q}(x)$— evaluate to small field elements over the relevant domain $\{0,1\}^n$, then the sum-check prover's field operations can be cheaper both in practice and asymptotically [Gru24, BDDT25].

5 Batch Evaluation, Virtual Polynomials, and Memory Checking

Let $f\colon \{0,1\}^\ell \to \mathbb{F}$ denote a function mapping ℓ-bit inputs to $\mathbb{F}$. In many SNARKs, the prover claims that a vector of committed values $(z_1,\ldots,z_T)$ is equal to $(f(y_1),\ldots,f(y_T))$ for some function f that is known to the verifier. Here, the T inputs $y_1,\ldots,y_T$ to f are also committed. This is called a *batch evaluation argument*. The verifier wants to check that all $z_i = f(y_i)$.

From the verifier's perspective, a batch-evaluation argument reduces the task of evaluating f at the T inputs $y_1,\ldots,y_T \in \{0,1\}^\ell$ to the task of evaluating the multilinear extension $\widetilde{f}$ of f at a *single* random input $r \in \mathbb{F}^\ell$.

We assume throughout this section that $\widetilde{f}(r)$ can be computed quickly by the verifier, say in $O(\ell)$ time. This is the case in key applications of batch-evaluation arguments, such as to proving correct executions of primitive CPU instructions as arises in zkVMs like Jolt (see Section 7).

Read-only memory checking: An alternative perspective on batch evaluation arguments Batch evaluation as formulated above is equivalent to a read-only memory check: f is the memory array, and the prover is claiming to read z_i from address y_i. The batch evaluation perspective views this as checking functional consistency; the memory-checking perspective treats it as proving that z_i is a valid access to a public memory array. These are two views of the same problem. SNARKs for read-only memory checking are also referred to as *lookup arguments*.

Both perspectives are useful. For example, the Jolt zkVM (Section 7) runs two separate instances of Shout. One is naturally viewed as a batch-evaluation argument that checks that each of the VM's primitive instructions are executed correctly, and the other is naturally viewed as a lookup argument (with the read-only memory storing the code of the computer program that the prover is proving it ran correctly).

5.1 The Shout batch evaluation argument

Recall that in a batch evaluation argument, the prover claims that for inputs $y_1, \ldots, y_T \in \{0,1\}^\ell$ and a known function $f : \{0,1\}^\ell \to \mathbb{F}$, the following equations hold:

$$z_1 = f(y_1), \ \ldots, \ z_T = f(y_T).$$

Let $\mathsf{rv} \in \mathbb{F}^T$ denote the vector of claimed output values $z_1, \ldots, z_T$, and let $\mathsf{ra} \in \{0,1\}^{2^\ell \times T}$ be the *access matrix*, where $\mathsf{ra}(x,j) = 1$ if and only if $x = y_j$. Note that for each fixed $j \in \{0,1\}^{\log T}$, $(\mathsf{ra}(x,j) \colon x \in \{0,1\}^\ell)$ is a *unit vector* in $\mathbb{F}^{2^\ell}$: exactly one entry of this vector equals 1 and the other entries are all 0. This vector is called the *one-hot representation* of y_j.

Then the batch evaluation claim is equivalent to the following constraint system:

$$\mathsf{rv}(j) = \sum_{x \in \{0,1\}^\ell} \mathsf{ra}(x,j) \cdot f(x) \text{ for all } j \in \{0,1\}^{\log T}. \tag{15}$$

Intuitively, the right hand side of Equation (15) is summing over all possible values x for y_j, testing (via the factor $\mathsf{ra}(x,j)$) if indeed x equals y_j, and if so, outputting $f(x)$.

As usual, let $\widetilde{f}$ be the multilinear extension of f, and define the multilinear extensions $\widetilde{\mathsf{ra}}$ and $\widetilde{\mathsf{rv}}$ of the access matrix and read-value vector, respectively. By the Schwartz-Zippel lemma (Theorem 4), to confirm that Equation (15) holds at all $j \in \{0,1\}^{\log T}$, up to soundness error $\log(T)/|\mathbb{F}|$, it suffices for the verifier to pick a random $r' \in \mathbb{F}^{\log T}$ and confirm that:

$$\widetilde{\mathsf{rv}}(r') = \sum_{x \in \{0,1\}^\ell} \widetilde{\mathsf{ra}}(x, r') \cdot \widetilde{f}(x). \tag{16}$$

The Shout protocol simply applies sum-check to prove this identity. Since $\widetilde{f}$ is known to the verifier and (we assume) fast to evaluate at a random point $r \in \mathbb{F}^\ell$, and $\widetilde{\mathsf{ra}}$ and $\widetilde{\mathsf{rv}}$ are committed by the prover, this gives a succinct argument that each of the claimed read-values $\mathsf{rv}(j)$ is indeed equal to $f(x_j)$.

Implementing Shout's sum-check prover efficiently. The sum in Equation (16) has 2^ℓ terms. Specifically, the sum is over *all possible inputs* to the function f, whose domain is $\{0,1\}^\ell$ In important applications of batch evaluation arguments, such as to zkVMs like Jolt that invoke them with f equal to a primitive CPU instruction, ℓ can be as large as 128, meaning this sum is over 2^{128} inputs. Hence, we cannot afford a sum-check prover that spends time linear in the number of terms being summed. Fortunately, in these settings $\widetilde{\mathsf{ra}}(x, r')$ is highly sparse: it evaluates to a non-zero value for only T inputs $x \in \{0,1\}^\ell$. Moreover, in these applications $\widetilde{f}$ is highly structured. Accordingly, the prefix-suffix sum-check proving algorithm of Section 4.2 applies [NTZ25, ST25, STW24] and hence the sum-check prover can be implemented in time T and in small space.

The Shout prover also has to prove that the access matrix ra is "well-formed". This means that each column $j \in \{0,1\}^{\log T}$ of the matrix must be one-hot, i.e.,

- $\mathsf{ra}(x,j) \in \{0,1\}$ for all $(x,j) \in \{0,1\}^{\ell + \log T}$.
- For each column $j \in \{0,1\}^{\log T}$, exactly one entry $\mathsf{ra}(x,j)$ of column j equals 1.

This can be proven efficiently via additional applications of the sum-check protocol that are all amenable to the prefix-suffix sum-check proving algorithm (Section 4.2). We omit further details for brevity.

When Shout is applied to the batch evaluation of a function $f : \{0,1\}^\ell \to \mathbb{F}$ (with input domain size $K = 2^\ell$), the prover performs $O(cK^{1/c} + cT)$ field operations, for any desired constant $c > 0$. The optimal setting of c is the one for which $K^{1/c} \approx T$. For example, if $K = 2^{128}$ and $T = 2^{30}$, then a reasonable choice is $c = 4$ or 8. This yields a total cost of $O(cT)$ field operations when $K^{1/c} \leq T$, and thus an amortized cost of just $O(c)$ per evaluation. This is typically far faster than the best-known circuit-based SNARKs applied to any circuit computing f.

Committing to $\widetilde{\mathsf{ra}}$ efficiently. The access matrix ra is Boolean and has exactly one 1 per column j. So it is very sparse—only T nonzero entries in a $2^\ell \times T$ matrix. If the polynomial commitment scheme used has commitment costs that depend primarily on the *number of nonzero entries* rather than total length (see Section 6), then committing to ra can be done efficiently without further optimization.

To the extent that this is not true—e.g., if the commitment key or evaluation proof computation time grows with the size of the committed vector and not just the number of non-zeros—it can be helpful to avoid explicitly committing to ra due to its size. In Section 5.2, introduces the notion of *virtual polynomials*, which allows ra to be expressed implicitly (e.g., as a tensor product of a small number of vastly smaller vectors). This mitigates any downsides of committing to huge-but-sparse vectors.

5.2 Reducing commitment costs via virtual polynomials

The idea of a *virtual polynomial* is to avoid committing to a polynomial or vector a directly, instead committing to a smaller or simpler object b such that a can be expressed as a low-degree function of b. This is especially useful when committing to a is slow, requires an enormous commitment key, or leads to expensive evaluation proofs. If each a_i is a low-degree function of b, then sum-check can still reason about a as if it were committed.

The terminology is recent (originating in the Binius paper [DP23]), but the idea dates back at least to the original GKR protocol [GKR08]. There, the (multilinear extension of) the vector of gate values at each layer of a circuit is treated as a virtual polynomial. The only polynomial that is actually explicitly committed by the prover is the input layer of the circuit.[7] We elaborate on these details shortly. The protocols in Jolt, Twist, and Shout also lean hard into this idea to minimize commitment costs for the prover.

Example: Virtualizing $\widetilde{\mathsf{rv}}$ in Shout. In the Shout batch-evaluation argument, there is no need for the prover to explicitly commit to the vector rv of claimed output values. Rather, the correct vector rv is *implied* by the committed inputs as captured by $\widetilde{\mathsf{ra}}$, along with the public function f. Equation (16) directly explains how to virtualize $\widetilde{\mathsf{rv}}$. When the verifier needs to evaluate $\widetilde{\mathsf{rv}}(r')$, the sum-check protocol is applied to the right hand side of Equation (16). From the verifier's perspective, this reduces the task of evaluating $\widetilde{\mathsf{rv}}(r')$ to the task of evaluating $\widetilde{\mathsf{ra}}(r, r')$ and $\widetilde{f}(r)$ for some fresh

[7]More precisely, in the GKR protocol itself, the input is public and the verifier explicitly reads it in full; in SNARKs derived from the GKR protocol, such as vSQL and Hyrax [ZGK+17, WTS+18], the input layer is committed by the prover.

random value $r \in \mathbb{F}^\ell$. $\widetilde{f}$ is public and assumed to be quickly evaluable, so the verifier can compute $\widetilde{f}(r)$ on its own. Meanwhile, $\widetilde{\mathsf{ra}}$ is either committed directly, allowing the prover to provide $\widetilde{\mathsf{ra}}(r, r')$, *or* $\widetilde{\mathsf{ra}}$ itself is virtualized in terms of other committed polynomials as described next.

Example: Virtualizing $\widetilde{\mathsf{ra}}$ in Shout. Let $K = 2^\ell$ be a memory size, and let $k \in \{0,1\}^{\log K}$ be an address. Recall that in Shout, the access matrix ra has K rows and T columns, and each column is *one-hot*: each column contains exactly one 1, with all other entries equal to 0. A straightforward approach to commit to $\widetilde{\mathsf{ra}}$ would be to materialize and commit to the full K-length vector for each j. As described later (Section 6), in some contexts this is acceptable. When it is not, we can *virtualize* $\widetilde{\mathsf{ra}}(\cdot, j)$ as follows.

Break $k \in \{0,1\}^{\log K}$ into d chunks of $\ell' = \log(K)/d$ bits each: $k = (k_1, \ldots, k_d)$ with $k_i \in \{0,1\}^{\ell'}$. Then define $\widetilde{\mathsf{ra}}_i(k_i, j)$ to be the one-hot vector for the i-th chunk. These $\widetilde{\mathsf{ra}}_i$ are committed to, and the full access matrix is defined implicitly by:

$$\mathsf{ra}(k, j) := \prod_{i=1}^{d} \mathsf{ra}_i(k_i, j). \tag{17}$$

Under this definition, the following equality of polynomials holds: For $r = (r_1, \ldots, r_d) \in \left(\mathbb{F}^{\log(K)/d}\right)^d$,

$$\widetilde{\mathsf{ra}}(r, r') = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r', j') \prod_{i=1}^{d} \widetilde{\mathsf{ra}}_i(r_i, j'). \tag{18}$$

To see that Equation (18) holds, observe that both the left-hand side and right-hand side are multilinear polynomials in the variables $(r_1, \ldots, r_d, r')$, and by Equation (17) and the definition of eq, it's easily seen that they agree whenever all variables in $(r_1, \ldots, r_d)$ and r' are from $\{0,1\}$. Since the multilinear extension of a function over $\{0,1\}^m$ is uniquely defined by its values on the hypercube, it follows that this identity holds over all of $\mathbb{F}^{\log K + \log T}$.

Although $\widetilde{\mathsf{ra}}$ is not committed directly, the sum-check protocol lets the verifier query it as if it were committed—by reducing to queries on the committed $\widetilde{\mathsf{ra}}_i$. Thus, the prover avoids materializing or committing to a massive $K \times T$ matrix. Instead, the prover commits to d matrices $\mathsf{ra}_1, \ldots, \mathsf{ra}_d$ that are each of size only $K^{1/d} \times T$.

Example: The GKR protocol for circuit evaluation. Recall from Section 4.2.1 that in the GKR protocol for arithmetic circuit evaluation, $\widetilde{V}_k$ denotes the multilinear extension of the values of all gates at layer k of the circuit. Equation (14) virtualizes $\widetilde{V}_k$ in terms of $\widetilde{V}_{k-1}$. From the verifier's perspective, when the verifier needs to evaluate $\widetilde{V}_k(r)$, the sum-check protocol can be applied to the right hand side of Equation (14) to reduce this task to evaluating the "circuit's wiring polynomial" $\widetilde{\mathsf{mult}}_k$ at a random point, and evaluating $\widetilde{V}_{k-1}$ at two random points.

For circuits with repeated structure, the verifier can evaluate $\widetilde{\mathsf{mult}}_k$ at the necessary point on its own in time logarithmic in the size of the circuit. The two evaluations of $\widetilde{V}_{k-1}$ are first interactively reduced to a claim about a single evaluation of $\widetilde{V}_{k-1}$ (we omit these details for brevity), and then this single claim is handled via a subsequent invocation of Equation (14) (with k now replaced by $k-1$). Eventually, the prover is forced to make a claim about $\tilde{V}_1(r')$ for a random point r', and as $\tilde{V}_1$ is the multilinear extension of the inputs to the circuit, this claim can be checked by the verifier directly (either because the verifier is able to directly process those inputs, or because they are committed by the prover, who can then provide $\tilde{V}_1(r')$ along with an evaluation proof. See Section 6 for more details on polynomial commitment schemes and their evaluation proofs).

5.3 Twist: Read/write memory checking

Twist solves the problem of proving correct access to a *mutable* memory array, where both reads and writes occur over time. This is the *read/write memory checking* problem. It generalizes Shout, which only handles *read-only* memory.

The setting. The prover executes a sequence of T memory operations over an array of size K. At each time step $j \in \{0,1\}^{\log T}$, there is both a read and a write (with the read logically occurring first). The prover defines:

- a *read address* $\mathsf{ra}(\cdot, j)$ and *write address* $\mathsf{wa}(\cdot, j)$, encoded as one-hot vectors in $\{0,1\}^K$,
- a *write value* $\mathsf{wv}(j)$ to be written to the cell selected by $\mathsf{wa}(\cdot, j)$,
- and a *read value* $\mathsf{rv}(j)$ returned from the cell selected by $\mathsf{ra}(\cdot, j)$.

Let $\widetilde{\mathsf{ra}}$, $\widetilde{\mathsf{wa}}$ denote the multilinear extensions of ra and wa, respectively. Let $\widetilde{\mathsf{rv}}$ and $\widetilde{\mathsf{wv}}$ denote the multilinear extensions of the read and write value vectors.

The goal. Prove that each read returns the value most recently written to the same memory cell.

From Shout to Twist. In Shout, memory is static: the value stored at address k is some fixed function $f(k)$, and we check:

$$\widetilde{\mathsf{rv}}(r) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{ra}}(k, r) \cdot \widetilde{f}(k).$$

In the mutable setting, the value at address k can change over time. Instead of a static $f(k)$, we now consider a dynamic function $f(k, j)$ denoting the value stored in cell k at time j.

By analogy to Shout, we wish to prove:

$$\widetilde{\mathsf{rv}}(r) = \sum_{k \in \{0,1\}^{\log K}, j \in \{0,1\}^{\log T}} \widetilde{\mathsf{eq}}(r, j) \cdot \widetilde{\mathsf{ra}}(k, j) \cdot \widetilde{f}(k, j), \tag{19}$$

where $\widetilde{\mathsf{eq}}(r, j)$ selects the time step j and $\widetilde{\mathsf{ra}}(k, j)$ selects the memory cell read at that time.

Why not commit to $\widetilde{f}(k, j)$ directly? We could try having the prover commit to the full $K \times T$ table of values $\widetilde{f}(k, j)$, then prove Equation (19) using sum-check. But this would require committing to up to KT non-zero entries. This is prohibitively expensive, especially when K is large.

Virtualizing $\widetilde{f}(k,j)$. Twist avoids this cost by virtualizing f. Rather than committing to $\widetilde{f}(k,j)$ directly, we express it as a low-degree function of smaller committed data.

Let $\mathsf{Inc}(j)$ be the difference between the new value to be written at time j and the current value already stored at the target cell:

$$\mathsf{Inc}(j) := \mathsf{wv}(j) - \sum_k \mathsf{wa}(k,j) \cdot f(k,j).$$

That is, $\mathsf{Inc}(j)$ represents how much the value at the write-address for time j changes thanks to the write operation at time j.

To express $f(k,j)$, we need to aggregate the effects of prior writes to cell k. Define $\widetilde{\mathsf{LT}}(j',j)$ as the multilinear extension of the strict less-than predicate on $\{0,1\}^{\log T}$: it evaluates to 1 if $j' < j$, and 0 otherwise. For any $(r,r') \in \mathbb{F}^{\log T} \times \mathbb{F}^{\log T}$, $\widetilde{\mathsf{LT}}(r,r')$ can be evaluated by the verifier in just $O(\log T)$ field operations (we omit details for brevity).

Then, the value stored in cell k at time j is:

$$\widetilde{f}(k,j) = \sum_{j' \in \{0,1\}^{\log T}} \widetilde{\mathsf{wa}}(k,j') \cdot \widetilde{\mathsf{Inc}}(j') \cdot \widetilde{\mathsf{LT}}(j',j). \tag{20}$$

Virtualizing $\widetilde{\mathsf{wv}}$. Once $\widetilde{f}(k,j)$ is expressed in terms of $\widetilde{\mathsf{Inc}}(j')$, the write values $\widetilde{\mathsf{wv}}(j)$ can also be expressed in terms of $\widetilde{\mathsf{Inc}}$:

$$\widetilde{\mathsf{wv}}(j) = \sum_{k \in \{0,1\}^{\log K}} \widetilde{\mathsf{wa}}(k,j) \cdot \left(\widetilde{f}(k,j) + \widetilde{\mathsf{Inc}}(j) \right).$$

This simply says that the new value being written is the old value at the cell plus the increment.

Conclusion. By committing only to $\widetilde{\mathsf{wa}}$ and $\widetilde{\mathsf{Inc}}$, the prover avoids committing to the full $K \times T$ memory table. The read values $\widetilde{\mathsf{rv}}$ are then proven correct via the Shout-style sum-check identity:

$$\widetilde{\mathsf{rv}}(r) \stackrel{?}{=} \sum_{j,k} \widetilde{\mathsf{eq}}(r,j) \cdot \widetilde{\mathsf{ra}}(k,j) \cdot \widetilde{f}(k,j),$$

with $\widetilde{f}(k,j)$ defined as a virtual polynomial via Equation (20).

Cost and efficiency. Proving memory consistency via Twist takes $O(T \log K)$ field operations in the worst case. This is already very good for small memories (e.g., many CPU architectures involve only $K = 32$ registers). Moreover, if memory accesses exhibit *locality*—i.e., most memory accesses target a cell that was already accessed not long before—then prover work drops significantly, with prover work growing only with a measure of the locality of the access sequence rather than with the logarithm of the memory size. For this and other reasons, proving reads and writes to large memories is a small fraction of total prover time in key applications of Twist (such as to the Jolt zkVM described in Section 7).

5.4 The "old" way of doing things: permutation-checking via grand products

A common task in SNARKs is to check that a list of read and write operations is consistent with memory semantics: each read must return the value most recently written to the same memory cell. As discussed in the previous section, this is the *read/write memory checking problem*. The dominant approach for many years was to reduce this task to a *permutation check*, which can then be verified using grand products.

Reducing to permutation checking via re-sorting. The idea is to introduce *untrusted advice*—data that the prover commits to up front, but which is not part of the public input or original witness. For memory checking, the prover commits to a reordered version of the read and write operations, sorted by memory address. Once sorted, consistency becomes easy to check: we can verify that the value read from each memory cell equals the value written in the most recent write to that address. But to trust this reasoning, we must verify that the sorted copy is indeed a *permutation* of the original list.

Permutation via grand products. Suppose we have two vectors $a, b \in \mathbb{F}^n$ and want to check that b is a permutation of a. A standard approach [Lip89] is to view a and b as the lists of roots of two univariate polynomials P, Q of degree n, and reduce the task to checking $P = Q$. Since a degree-n univariate polynomial is determined by its values at $n + 1$ distinct points, it suffices to check equality at a random point:

$$\prod_{i=1}^{n}(a_i - r) = \prod_{i=1}^{n}(b_i - r). \tag{21}$$

To prove Equation (21), one invokes a *grand product argument*, i.e., a protocol for verifying that a large product of committed values evaluates to a claimed result.

Why this is costly. This approach introduces several sources of overhead:

- The prover must commit to extra data (the sorted copy), increasing total commitment cost.
- Sorting may be incompatible with streaming or low-memory proving—especially if the trace is too large to fit in memory.
- Even the fastest-prover grand product arguments (e.g., those based on GKR [GKR08, Tha13]) require nontrivial prover work, namely several field operations per product term.
- Some grand product arguments require the prover to commit to *partial products*, which consist of random field elements (due to the randomness in Equation (21)) and thus are expensive to commit to. That is, these grand product arguments do not achieve small-value preservation (see Section 6.1 and Remark 7 for details).

The Twist and Shout perspective. Twist and Shout take a different approach to memory checking arguments: rather than reducing sparse problems to dense ones, they *embrace sparsity*, committing directly to sparse representations and verifying them using *sparse sum-checks*, where prover cost scales with the number of nonzero terms rather than the ambient dimension.

This design minimizes prover work at both the PCS and PIOP layers. Combined with techniques like virtual polynomials and prefix-suffix decompositions, it yields SNARKs that are not only faster and simpler, but also compatible with streaming proving—that is, provers can run with low memory without relying on recursion.

In practice, the payoff is significant. For many functions f and sufficiently large batch sizes, Shout achieves faster proving than even the best-performing SNARKs for circuits (even ones based on sum-check, like the GKR protocol [GKR08, CMT12, Tha13] or Spartan [Set20]). Similarly, Twist outperforms all known permutation-based approaches to memory checking. Shout also enhances simplicity by eliminating the need to specify a circuit for the function being batch-evaluated.

6 Viewing PCSes as "Cryptographically Wrapped" PIOPs

6.1 Background on Commitment Schemes

Many polynomial commitment schemes involve computing expressions of the form

$$\prod_{i=1}^{N} g_i^{c_i}, \tag{22}$$

where $c_i \in \mathbb{F}$ are coefficients and $g_1, \ldots, g_N$ are fixed elements of a (multiplicative) group $\mathbb{G}$. This is known as a *Pedersen commitment* to the vector $c = (c_1, \ldots, c_N)$.

This operation is also called a *multi-exponentiation*, or a *multi-scalar multiplication* (MSM) when using additive group notation. In practice, MSMs are a central performance bottleneck in many cryptographic SNARK components.

MSMs enjoy two key efficiency properties:

- **Zero coefficients are free.** If $c_i = 0$, then the corresponding term $g_i^{c_i} = 1$ has no effect on the product. These terms can be omitted entirely, saving work.
- **Small coefficients are faster.** If the c_i's are drawn from a small range (e.g., $\{0, 1, \ldots, 2^{20}\}$) rather than the full field, the exponentiations are much cheaper to compute. For example, computing g^4 is vastly faster than computing $g^{2^{128}}$.

Other commitment families—such as lattice-based or hashing-based schemes—also benefit from small or sparse vectors, often in similar ways [DP23, DP24, BCF+24, NS25]. We omit further details of these schemes for brevity.

We say a SNARK design exhibits *small-value preservation* if the vectors or polynomials involved in commitments and openings predominantly consist of small (e.g., low-bit-width) field elements. This property accelerates both commitment and opening operations, often by an order of magnitude in practice, and is a key goal in modern prover design. (As discussed earlier—see Remark 7—small-value preservation is useful not only for commitment schemes but also for optimizing the prover cost of sum-check itself).

6.2 KZG is quotienting "in the exponent"

Throughout this section, we use *multiplicative group notation*: group elements are written as powers of a generator g, and the structured reference string (SRS) is a list of the form $\{g^{\tau^i}\}_{i=0}^{N-1}$. (Later, when discussing Bulletproofs/IPA, we will switch to additive notation.)

This SRS is commonly referred to as the "powers of τ" SRS. Techniques have been developed to generate this SRS via a "ceremony" that many parties can participate in, each contributing their own randomness to determine τ, and such that no entity can reconstruct τ unless all parties participating in the ceremony collude [BGM17, GKM+18].

KZG commitments are simply the quotienting PIOP from Section 3.1.1, with the verifier's challenge τ hidden *in the exponent* of the SRS. Let us explain.

Let $p(X)$ be a univariate polynomial of degree at most $N-1$. The prover commits to p by computing:

$$\mathsf{cm}(p) := g^{p(\tau)}.$$

Even though τ is unknown, the prover can compute $g^{p(\tau)}$ using the SRS. If $p(X) = \sum_{i=0}^{N-1} c_i X^i$, then:

$$g^{p(\tau)} = \prod_{i=0}^{N-1} \left(g^{\tau^i}\right)^{c_i}.$$

To open the commitment at a point $r \in \mathbb{F}$, the prover sends the claimed evaluation $y := p(r)$ and constructs the quotient polynomial:

$$q(X) := \frac{p(X) - y}{X - r}.$$

If indeed $p(r) = y$, then $p(X) - y$ is divisible by $X - r$ and hence q has degree at most $N-2$. The prover then computes $g^{q(\tau)}$ from the SRS. The verifier wants to check that:

$$p(\tau) - y = q(\tau) \cdot (\tau - r).$$

This is done *in the exponent* using pairings.

What do pairings do? For simplicity of notation, let us focus on so-called *symmetric pairings* for simplicity. This refers to a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ with the property that:

$$e(g^a, g^b) = e(g, g)^{ab}.$$

This lets the verifier check multiplicative relations between exponents without learning them. For example, given g^a, g^b, and g^c, without knowing a, b, c, the verifier can test whether $ab = c$ by checking:

$$e(g^a, g^b) \stackrel{?}{=} e(g, g^c).$$

The KZG check. In our case, the verifier checks:

$$e\left(g^{p(\tau)} \cdot g^{-y},\ g\right) \stackrel{?}{=} e\left(g^{q(\tau)},\ g^{\tau - r}\right).$$

This confirms that $p(\tau) - y = q(\tau) \cdot (\tau - r)$, using only group elements and pairings.

Commitments are fast, openings are not. Committing to p is fast: if p is sparse, the many zero-coefficients do not affect the commitment time, and if all of p's coefficients are small then even the non-zero coefficients are fast to commit to. But the KZG evaluation proof is a commitment $g^{q(\tau)}$ to the quotient polynomial $q(X) := \frac{p(X)-y}{X-r}$. Unfortunately, even if p is sparse and/or has small coefficients, q will not. Here, we see a downside of quotienting in action: the key properties of sparsity and small-values are not preserved under quotienting. This makes quotient polynomials especially expensive to commit.

6.3 Bulletproofs/IPA is sum-check for inner product

In this section, we switch from multiplicative to additive group notation, so that a Pedersen vector commitment to a length-N vector $u \in \mathbb{F}^N$ is no longer written as $\prod_{i=1}^N g_i^{u_i}$ but rather as

$$\sum_{i=1}^{N} u_i \cdot g_i. \tag{23}$$

$u_i \cdot g_i$ is called a *scalar multiplication*, and Expression (23) is often abbreviated as $\langle u, g\rangle$.

The core of the Bulletproofs/IPA protocol [BCC+16, BBB+18] is proving knowledge of a preimage to a Pedersen vector commitment. That is, given a group element C, the prover wishes to convince the verifier that it knows a vector $u = (u_1, \ldots, u_N) \in \mathbb{F}^N$ such that

$$C = \langle u, g\rangle = \sum_{i=1}^{N} g_i^{u_i},$$

where $g = (g_1, \ldots, g_N)$ is a vector of random group elements.

The Bulletproofs commitment is just a Pedersen commitment to u. In polynomial commitment schemes based on Bulletproofs, the vector u represents the coefficient vector of a univariate polynomial $p(X)$ (or a multilinear one, but let us focus on the univariate case for simplicity), and the prover wants to prove that it knows u such that:

$$\langle u, g\rangle = C \quad \text{and} \quad \langle u, \vec{r}\rangle = y, \tag{24}$$

where $\vec{r} = (1, r, r^2, \ldots, r^{N-1})$ is the evaluation vector for a claimed evaluation point $r \in \mathbb{F}$, and $y = p(r)$ is the claimed evaluation. In this view, polynomial commitment schemes reduce to proving knowledge of u satisfying the two inner product constraints of Expression (24). Let us focus on the constraint that $\langle u, g\rangle = C$ since this captures the core of the protocol.

A homomorphic sum-check. Fix a group element h and let $z = (z_1, \ldots, z_N)$ be such that $g_i = z_i \cdot h$ for each $i = 1, \ldots, N$. Since the commitment key elements $g_1, \ldots, g_N$ were chosen at random and the discrete logarithm problem is assumed to be hard, the scalar coefficients z_i are unknown to everyone—including the prover. That is, the prover knows the group elements g_i but not the underlying scalars z_i (i.e., the discrete log of g_i with respect to h is hidden).

Now let $N = 2^n$ and consider the sum-check protocol applied to compute the inner product

$$\langle u, z\rangle = \sum_{x \in \{0,1\}^n} \widetilde{u}(x) \cdot \widetilde{z}(x).$$

Recall that in round i, the honest sum-check prover sends a degree-two univariate polynomial:

$$s_i(X) = \sum_{x' \in \{0,1\}^{n-i}} \widetilde{u}(r_1, \ldots, r_{i-1}, X, x') \cdot \widetilde{z}(r_1, \ldots, r_{i-1}, X, x').$$

If $s_i(X) = a + bX + cX^2$, the prover can specify the polynomial by sending its coefficients. In practice, it suffices to send just two coefficients (e.g., a and c), as the verifier can reconstruct the full polynomial using the sum-check consistency check $s_{i-1}(r_{i-1}) = s_i(0) + s_i(1)$.

The key point is that each coefficient (e.g., a or c) is a linear combination of the z_i, with coefficients depending only on u and the verifier's challenges $r_1, \ldots, r_{i-1}$. Even though the prover does not know the z_i explicitly, it does know the group elements $g_i = z_i \cdot h$, and so it can compute $a \cdot h$ and $c \cdot h$ as linear combinations of the g_i. For example, if $a = 10z_1 + 20z_2$, then:

$$a \cdot h = 10 \cdot g_1 + 20 \cdot g_2,$$

so the prover can compute $a \cdot h$ homomorphically, using only the public group elements g_1 and g_2, without knowing the hidden scalars z_1 or z_2.

At the end of the protocol, the verifier must check that:

$$s_n(r_n) = \widetilde{u}(r_1, \ldots, r_n) \cdot \widetilde{z}(r_1, \ldots, r_n).$$

In the context of Bulletproofs, this check is performed using group elements: the prover sends the scalar $\widetilde{u}(r_1, \ldots, r_n)$, and the verifier computes $\widetilde{z}(r_1, \ldots, r_n) \cdot h$ using the commitment key $g_1, \ldots, g_N$. The verifier then checks:

$$s_n(r_n) \cdot h \overset{?}{=} \widetilde{u}(r_1, \ldots, r_n) \cdot \left(\widetilde{z}(r_1, \ldots, r_n) \cdot h\right).$$

Unlike KZG, Bulletproofs requires no pairings and does not rely on a structured reference string: the commitment key consists of independently sampled group elements rather than a "powers of τ" SRS. Conversely, unlike KZG, Bulletproofs evaluation proofs consist of logarithmically many group elements (rather than a constant number of them) and involve an especially slow prover and verifier.

Security caveats. Although Bulletproofs *is* just the sum-check protocol, its *knowledge soundness*—i.e., establishing that a successful Bulletproofs prover actually must *know* the vector u—does not follow directly from the soundness of sum-check. We do not cover the actual security analysis of Bulletproofs in this survey, but merely highlight its mechanical equivalence to the sum-check protocol. For further details of this connection, see [BCS21, Appendix A].

Costs and downsides. The Bulletproofs evaluation proof consists of $\log N$ rounds with two group elements sent per round. For example, for $N = 2^{30}$ this translates to a roughly 2 KB proof.

However, the prover and verifier are very slow. Roughly speaking, all the *field multiplications* that the sum-check prover does translate into *scalar multiplications* that the Bulletproofs prover has to do. A scalar multiplication is several thousands of times slower than a 256-bit field multiplication.[8] The Bulletproofs verifier also has to compute $\widetilde{z}(r_1, \ldots, r_n) \cdot h$, which requires forming a linear combination of the group elements $g_1, \ldots, g_N$. This entails roughly N scalar multiplications.

[8]A scalar multiplication involves about 400 group additions via the standard double-and-add algorithm. Each group addition in turn involves at least 6-12 operations over the *base field* of the elliptic curve group (which is at least 256 bits in practice) [GW20].

6.4 Hyrax: Avoiding Sum-Check by Leveraging Product Structure

This survey is, of course, a proponent of the sum-check protocol, due in part to its extremely fast prover (see Sections 4.1 and 4.2). However, in the context of Bulletproofs/IPA, the sum-check protocol is actually very slow. This is because every field operation the sum-check prover does in the PIOP setting translates into a scalar multiplication in the Bulletproofs/IPA setting, and scalar multiplications are thousands of times slower than field operations. So in this section, we describe a commitment scheme that *avoids* sum-check by leveraging "multiplicative structure" in polynomial evaluation queries that Bulletproofs/IPA ignores.[9]

Like Bulletproofs, Hyrax [WTS+18] is a polynomial commitment scheme based on Pedersen commitments. Unlike Bulletproofs, Hyrax does not rely on sum-check; instead, it uses the identity:

$$p(r) = \langle c, \vec{r} \rangle = \vec{b}^\top M \vec{a}$$

to reduce evaluation proofs to verifying a *vector-matrix-vector* computation.

Here:

- $p(X) = \sum_{i=0}^{N-1} c_i X^i$ is the committed polynomial
- $c = (c_0, \ldots, c_{N-1})$ is its coefficient vector,
- $\vec{r} = (1, r, r^2, \ldots, r^{N-1})$ is the evaluation vector at point r,
- $N = m^2$ is assumed to be a perfect square for convenience,
- $\vec{a}, \vec{b} \in \mathbb{F}^m$ and $M \in \mathbb{F}^{m \times m}$ satisfy:

$$\vec{a} = (1, r, \ldots, r^{m-1}), \quad \vec{b} = (1, r^m, \ldots, r^{m(m-1)}),$$

  and the matrix M is defined by reshaping the coefficient vector c into row-major order:

$$M_{i,j} := c_{i \cdot m + j}.$$

Hence, evaluation of a degree-N polynomial at point r reduces to computing:

$$p(r) = \vec{b}^\top M \vec{a}.$$

Commit phase. The prover commits to each *column* of the matrix M separately, using a vector of m group generators $G = (g_1, \ldots, g_m)$. Let $M^{(j)}$ denote the j-th column of M. Then the prover computes:

$$C_j = \sum_{i=1}^{m} M_{i,j} \cdot g_i = \langle M^{(j)}, G \rangle$$

for $j = 1, \ldots, m$, and the full commitment is the vector $(C_1, \ldots, C_m)$.

[9] IPA stands for Inner Product Argument, which refers to the fact that, in full generality, Bulletproofs/IPA is able to prove the inner product of any two vectors (one or both of which is committed), even when *neither* of those vectors has multiplicative structure. Hyrax is not a general inner product argument, but its leveraging of multiplicative structure in polynomial evaluation queries leads to much faster evaluation proofs, shorter commitment keys, and faster verification time than Bulletproofs/IPA.

Evaluation phase. To open $p(r)$, the prover computes $\vec{v} = M\vec{a} \in \mathbb{F}^m$, so that:

$$p(r) = \langle \vec{b}, \vec{v} \rangle.$$

The prover sends $\vec{v}$ to the verifier.

The verifier computes:

$$\mathsf{cm} := \sum_{j=1}^{m} C_j r^j,$$

which is a commitment to $M\vec{a}$ that the verifier is homomorphically deriving on its own from the commitments $C_1, \ldots, C_m$ to the columns of M.

The verifier then checks the vector $\vec{v}$ sent be the prover that is *claimed* to equal $M\vec{a}$ indeed opens the homomorphically-derived commitment cm to $M\vec{a}$, i.e.,

$$\mathsf{cm} \stackrel{?}{=} \langle \vec{v}, G \rangle .$$

This confirms that the response vector $\vec{v}$ is consistent with the committed matrix M and vector $\vec{a}$, in which case the desired evaluation $p(r)$ equals $\langle \vec{b}, \vec{v} \rangle$, and this inner product can be computed directly by the verifier.

Efficiency.

- The commitment consists of $m = \sqrt{N}$ group elements.
- The opening proof consists of an m-length vector $\vec{v}$ and an opening of the inner product.
- Verifier work includes computing two MSMs of size $\sqrt{N}$.

Comparison with sum-check-based schemes. Hyrax avoids sum-check entirely and instead leverages "multiplicative structure" in the polynomial evaluation vector $\vec{r}$. The evaluation proof is non-interactive, and consists simply of the "partial evaluation vector" $M \cdot \vec{a}$. Its commitments are sublinear in size, but verifier cost is high.

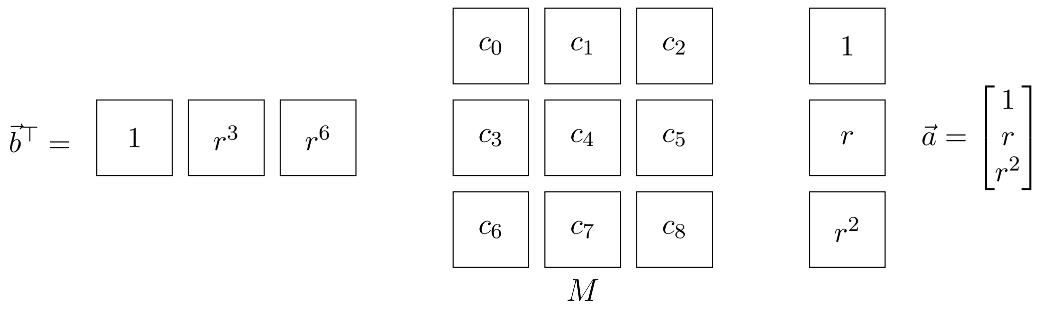
The benefits of Hyrax relative to Bulletproofs are as follows:

- The size of the commitment key is $\sqrt{N}$ rather than N group elements.
- The evaluation proofs are very fast to compute, involving no cryptographic operations. It just amounts to computing the "partial evaluation vector" $M \cdot \vec{a}$.
- Verifier time is two MSMs of size $\sqrt{N}$ rather than one of size N.

The downside is the commitment is big ($\sqrt{N}$ group elements instead of one) and the evaluation proof is big ($\sqrt{N}$ field elements instead of $2 \log N$ group elements).

6.5 Dory: The best of Bulletproofs/IPA and Hyrax

Dory [Lee21] is a polynomial commitment scheme that combines the best aspects of Hyrax and Bulletproofs/IPA. Like Bulletproofs, Dory uses just a single group element to commit to a polynomial. Like Hyrax, it supports fast evaluation proofs that exploit multiplicative structure in the evaluation vector—without relying on the sum-check protocol. It also improves over both schemes in one important respect: verifier time is logarithmic, rather than linear in the size of the committed vector.



$$p(r) = \vec{b}^\top M \vec{a}$$

Figure 2: Hyrax's view of a degree-8 polynomial evaluation query as a vector-matrix-vector product.

Commitment phase. The key idea is to commit not to the polynomial's coefficient vector directly, but to the corresponding Hyrax commitment—a vector of group elements. Dory uses a commitment scheme from AFGHO [AFG$^+$10] that supports committing to group elements using pairings. This compresses the full Hyrax commitment (which consists of $\sqrt{N}$ group elements) into a single group element in a pairing group.

Evaluation proof. To open the commitment at a point r, the prover proves two things:

1. It knows a Hyrax commitment (i.e., the $\sqrt{N}$ group elements) that opens to the Dory commitment.

2. It knows a Hyrax evaluation proof that would cause the Hyrax verifier to accept.

Both of these are proven using a variant of Bulletproofs/IPA adapted to the setting of AFGHO-style commitments. Thanks to the homomorphic structure of these commitments, the Dory verifier runs in *logarithmic time*—a major improvement over the linear-time verifier in Bulletproofs *and* the square-root verifier time of Hyrax.

Advantages for sparse vectors. Dory is particularly well-suited to committing to large, sparse vectors. The commitment key is only of size $\sqrt{N}$, so committing to a large vector $u \in \mathbb{F}^N$ with mostly-zero entries does not inflate the commitment key too much. The number of cryptographic operations required to open a commitment—i.e., the number of scalar multiplications and pairings required to compute an evaluation proof—is also just $O(\sqrt{N})$. Meanwhile, all costs that *do* scale with the number of non-zeros (e.g., commitment time, field operations in computing evaluation proofs) benefit directly from sparsity (i.e., zeros in the committed vector do not contribute to these costs).

This of course does not mean that Dory can be used to fruitfully commit to vectors of size, say, $N = 2^{128}$. For example, in this case the $\sqrt{N}$-size commitment key would be far too big for the prover to explicitly generate and store. Dory becomes especially effective when paired with PIOP-based techniques that control the size of sparse vectors to be committed (see Section 5.2 for details). The PIOP techniques can be used to ensure N is not larger than, say, 2^{50}, and then the $O(\text{sparsity} + \sqrt{N})$ prover costs of Dory are highly attractive.

Summary. Dory achieves the small commitment size of Bulletproofs, the fast evaluation of Hyrax, and a logarithmic-time verifier, all while gracefully handling sparse inputs. It exemplifies how the *combination* of the sum-check protocol and vector-matrix-vector structure in polynomial evaluation queries can lead to commitment schemes with exactly the properties needed to turn sparse-sum-check-based PIOPs like Twist and Shout (Sections 5.1 and 5.3) into efficient SNARKs.

It is worth mentioning that, after committing to a polynomial of size N, computing a Dory evaluation proof requires the prover to perform $O(\sqrt{N})$ pairings (i.e., bilinear map evaluations) and scalar multiplications. Each pairing operation and scalar multiplication is concretely expensive, however the number of such operations performed by the Dory prover grows sublinearly with N. Thus, for small values of N, these $O(\sqrt{N})$ pairings and scalar multiplications can dominate prover time (relative to the rest of the work performed prover in the SNARK that invokes Dory). But for large values of N, these $O(\sqrt{N})$ pairings and scalar multiplications are a tiny fraction of overall prover work (which typically grows linearly with N). For example, in the Jolt zkVM described in Section 7 below, Dory evaluation proofs are under 10% of total prover time so long as the VM execution being proven consists of at least tens of millions of CPU cycles.

7 Case Study: zkVMs and Jolt

A “zero-knowledge virtual machine” (zkVM) is a general-purpose SNARK engine that proves correct execution of a low-level program—typically written in assembly or bytecode for some fixed instruction set architecture (ISA).[10] Rather than writing circuits manually, users just write code in any high-level programming language that can be compiled down to bytecode for the VM, and receive proofs of correct execution.

This approach has many advantages, including:

- **Ease of use.** Developers can write normal code instead of hand-authoring arithmetic circuits.
- **Pre-existing tooling.** Developing an optimized and bug-free compiler from high-level programming languages to lower-level representations (whether circuits or VM bytecode) is an enormous undertaking. A zkVM targeting a standard ISA like RISC-V can reuse existing compilers that have already been subjected to substantial scrutiny and optimization.

A common misconception is that zkVMs must be slower than “circuit-SAT SNARKs”—i.e., SNARKs applied directly to hand-constructed circuits. This seems intuitive: after all, a zkVM prover is simulating a virtual CPU, often with far more internal steps than the original computation would require if circuitized carefully. After all, a circuit—like a hardware ASIC—is tailored to a single task, while a CPU pays for its flexibility by doing all the extra work of fetch–decode–execute every cycle.

In fact, the opposite is often true. This is because all that extra work is highly structured and repeated identically across cycles. VMs fetch, decode, and execute the same few instructions millions or billions or trillions of times. They read and write every cycle from the same small register set. The constraint systems that enforce correct execution are short, fixed, and reused across cycles. And values arising inside the VM (registers, addresses, immediates) are small integers, because native data types on standard VMs are only 32 or 64 bits in size.

[10]Zero-knowledge here is often a misnomer, as many zkVMs are non-zero-knowledge SNARKs for VM execution.

This all means the SNARK prover doesn't need to pay "full price" for proving that work was done correctly. Sophisticated applications of the sum-check protocol exploit the repeated structure to minimize the overhead of proving relative to native execution (i.e., simply running the CPU with no proof of corretness).

Jolt [AST24] is a zkVM for the standard RISC-V architecture that exploits this repeated structure to the fullest (specifically, Jolt targets RV64IMAC, which means RISC-V with 64-bit registers, and the Multiplication extension, among others). Each cycle of the VM is SNARKed as follows:

- **Fetch:** The VM reads from the program bytecode to determine which instruction to execute. Jolt uses Shout (viewed as a memory-checker for read-only memory) to prove this read was answered correctly.
- **Decode and execute:** The VM reads up to two registers, applies the appropriate instruction to the returned values, and writes the result to a designated output register. Jolt uses Twist to prove these register reads and writes were correctly performed. It uses Shout (viewed as a batch-evaluation argument) to check that instruction execution was correct.
- **RAM access:** If the instruction is a Load or Store, the VM reads or writes to main memory (RAM). Jolt uses a second instance of Twist to prove memory accesses are processed correctly.
- **Constraint checking:** Jolt enforces global correctness by checking that the transition constraints for the VM are satisfied. It uses Spartan to prove satisfaction of a small set of roughly 20 constraints, repeated identically in every cycle—a structure that Jolt's variant of Spartan is optimized to exploit.

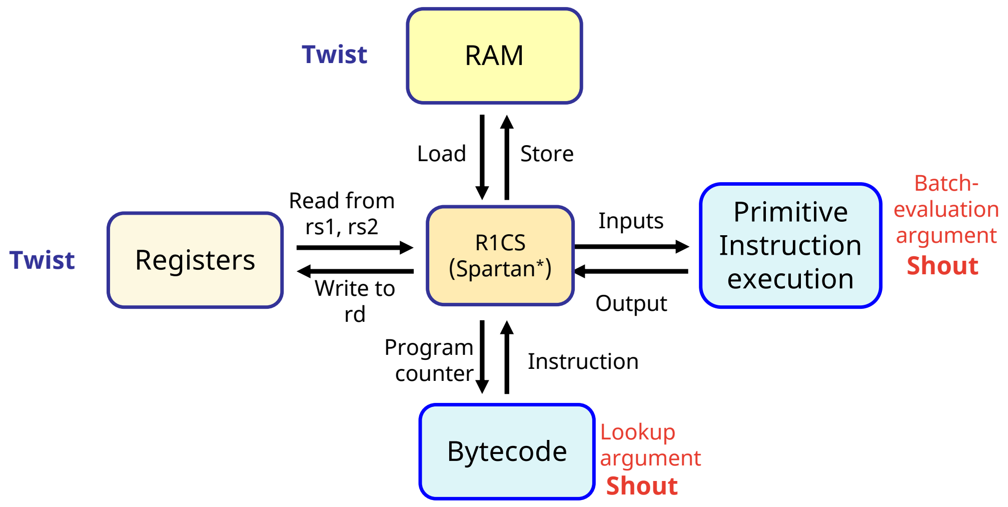


Figure 3: Overview of the Jolt zkVM architecture and proof structure. *Jolt uses a variant of Spartan optimized for repeated constraints.

Performance. For real-world programs, Jolt produces SNARKs with proof sizes around 50 KB and prover throughput in the MHz range on commodity CPUs (e.g., over 500,000 RV64IMAC cycles per second on a high-end MacBook, and over 1.5 million on a 32-core machine [GMT+25]). The prover's speed depends primarily on the number of VM cycles, with little sensitivity to the specific computation being executed.

These performance levels are on par with, and often better than, popular SNARKs applied to hand-optimized arithmetic circuits. But unlike circuit-SAT SNARKs, Jolt is general-purpose, flexible, and easier to use. It exemplifies how sum-check-based SNARKs can leverage structure to achieve both speed and simplicity.

Importantly, Jolt's performance was not an empirical surprise—it was predicted in advance by a theoretical accounting of the prover's workload [ST25]. The Jolt prover performs roughly 500 field multiplications (over a 256-bit field) per RV64IMAC cycle.[11] Each field multiplication costs around 100 CPU cycles under conservative assumptions [Xop], implying an expected prover cost of 50,000 CPU cycles per VM cycle. On a single-threaded 4 GHz core, this translates to an expected throughput of about 80,000 VM cycles per second.[12] Scaling across 16 threads yields a projected throughput of about 1 million VM cycles per second—closely matching observed performance.

Of course, real hardware is complex, and substantial engineering effort was required to reach this level of performance. Nonetheless, theoretical grounding played a crucial role. Without precise operation counts, performance tuning becomes guesswork: protocol designers don't know which parts of the system are slower than they should be, because they don't know what performance to expect. And while empirical benchmarking is valuable, comparisons that ignore implementation maturity or compare to suboptimal baselines mislead. In short, accurate operation-level cost analysis is essential for understanding tradeoffs and guiding practical SNARK design.

8 Takeaways and Future Directions

The most performant SNARKs today share a common blueprint: they leverage the sum-check protocol to shift work to the prover without relying on cryptography, and minimize commitment costs (and speed up sum-check itself) through techniques like sparse sum-checks, small-value preservation, and virtual polynomials. The hierarchy of prover speeds is as follows:

- Slowest: no use of sum-check at all.
- Faster: uses sum-check, but fails to exploit structure in the computation, e.g., using generic circuits, perhaps requiring the prover to commit to random values (i.e., no small-value preservation), etc.
- Fastest: sum-check paired with awareness of repeated structure—batch-evaluation arguments, small-value preservation, sparse sums, virtual polynomials.

Below we summarize core design lessons, followed by open directions for the field.

Key lessons.

- **Committing to data is expensive.** Every value the prover commits to must later be proven correct. Reducing the number and size of committed objects is essential for fast proving.

[11]This estimate excludes costs such as Dory evaluation proofs, which can be a significant portion of total prover time for small executions (e.g., under tens of millions of cycles), but become negligible for larger workloads.

[12]Some hardware platforms, such as GPUs, may be more limited by memory bandwidth than compute when running sum-check-based SNARK provers. While bottlenecks differ, throughput typically remains within the same order of magnitude.

- **Sum-check offloads work without cryptography.** It replaces expensive commitments with interaction and randomness.
- **Interaction is a resource.** Removing interaction twice—first to obtain a PCP, then again via Fiat–Shamir—is wasteful. If you're going to use Fiat–Shamir, apply it once, to a good interactive protocol.
- **Exploit structure in the computation.** Sparse sums, small values, and repeated patterns (as in VMs) are not just artifacts—they're opportunities. If you don't exploit them fully, you're leaving both performance and simplicity on the table.
- **Many PCSes are sum-checks in disguise.** Bulletproofs/IPA is sum-check over inner products, applied homomorphically when one vector is hidden in the exponent. Hyrax avoids sum-check, relying instead on vector-matrix-vector encodings. Dory combines both approaches, mitigating Bulletproofs/IPA's downsides and achieving sublinear commitment key size and sublinear cryptographic work during evaluation proof computation.

Open directions.

1. **Post-quantum and smaller-field SNARKs.** Twist, Shout, and Jolt currently rely on elliptic curve cryptography over 256-bit fields. An open challenge is to transition to 128-bit fields and post-quantum security in the simplest and most performant way possible. (Recent research has substantially mitigated the performance hit of using 256-bit fields [Gru24, BDDT25], but 128-bit fields would still be faster).

   Lattice-based commitment schemes are promising but still maturing. Hash-based schemes are also attractive, and despite substantial progress [DP23, DP24, ST25], integrating them with sparse sum-checks still raises both engineering and theoretical challenges.

2. **Formal verification and correctness.** SNARKs—especially zkVMs—are on the path to becoming critical cryptographic infrastructure, just like digital signatures and encryption. But they're vastly more complex, and implementations today contain bugs and subtle correctness issues. Formal verification is essential. Progress is underway, but much remains to be done.

3. **Beyond the VM abstraction.** Despite major improvements to zkVM performance, the fastest SNARKs for fixed circuits still outperform general-purpose zkVMs.[13] Should we be combining the two? What's the right tradeoff between flexibility, usability, and raw performance? Are there principled alternatives to today's "precompiles" (optimized circuits and associated SNARKs that are "glued in" to the VM runtime)? This remains an open and active area of research.

The core design principles for simple and performant SNARKs are becoming clear. The open problems above are challenging but tractable. With the right focus—on performance, simplicity, and correctness—the coming years should see sum-check-based SNARKs transition to foundational infrastructure for the digital world.

[13]While Shout, a batch evaluation argument, outperforms even the fastest circuit-based SNARKs for proving primitive instruction execution, full zkVMs like Jolt can still be slower overall than the fastest SNARKs for fixed circuits. This is because zkVMs must prove the full "fetch-decode-execute" logic for each CPU cycle. In such systems, overheads from the "fetch-decode" stages can outweigh the gains from using Shout for instruction execution.

Acknowledgments. I thank Arpita Patra and Chaya Ganesh for inviting me to contribute to a special issue of the *Journal of the Indian Institute of Science*, which prompted the writing of this survey. I am grateful to Zach Destefano and Kabir Peshawaria for valuable conversations and detailed comments on an earlier version of the manuscript. Thanks also to the entire Jolt team for their tireless engineering efforts and for embodying the design principles described in this survey, alongside many long hours and much fun.

Disclosures. Justin Thaler is a Research Partner at a16z crypto and is an investor in various blockchain-based platforms, as well as in the crypto ecosystem more broadly (for general a16z disclosures, see `https://www.a16z.com/disclosures/`.)

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*, volume 15604 of *Lecture Notes in Computer Science*, pages 214–243. Springer, 2025.

[AFG$^+$10] Masayuki Abe, Georg Fuchsbauer, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Structure-preserving signatures and commitments to group elements. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*, pages 209–236. Springer, 2010.

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthu Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[AS24] Arasu Arun and Srinath Setty. Nebula: Efficient read-write memory and switchboard circuits for folding schemes. Cryptology ePrint Archive, 2024.

[AST24] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2024.

[BBB$^+$18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[BBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, 2018.

[BCC$^+$16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[BCF$^+$24] Martijn Brehm, Binyi Chen, Ben Fisch, Nicolas Resch, Ron D Rothblum, and Hadas Zeilberger. Blaze: Fast SNARKs from interleaved RAA codes. Cryptology ePrint Archive, 2024.

[BCS21] Jonathan Bootle, Alessandro Chiesa, and Katerina Sotiraki. Sumcheck arguments and their applications. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part I 41*, pages 742–773. Springer, 2021.

[BDDT25] Suyash Bagad, Quang Dao, Yuval Domb, and Justin Thaler. Speeding up sum-check proving. Cryptology ePrint Archive, Paper 2025/1117, 2025.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017(1050), 2017.

[Blo25] Remco Bloemen. Zero-knowledge and WHIR. X (formerly Twitter), status update, August 2025. https://x.com/recmo/status/1960980010455519622.

[BTVW14] Andrew J. Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. Verifiable computation using multiple provers. ePrint Report 2014/846, 2014.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS)*, 2012.

[Dia25] Benjamin E. Diamond. Zero-knowledge polynomial commitment in binary fields. Cryptology ePrint Archive, Paper 2025/1015, 2025.

[DP23] Benjamin E. Diamond and Jim Posen. Succinct arguments over towers of binary fields. Cryptology ePrint Archive, Paper 2023/1784, 2023. https://eprint.iacr.org/2023/1784.

[DP24] Benjamin E Diamond and Jim Posen. Polylogarithmic proofs for multilinears over binary towers. Cryptology ePrint Archive, 2024.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986.

[GH98] Oded Goldreich and Johan Håstad. On the complexity of interactive proofs with bounded communication. *Inf. Process. Lett.*, 67(4):205–214, 1998.

[GKM$^+$18] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-snarks. In *CRYPTO 2018 (LNCS 10993)*, pages 698–728. Springer, 2018.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 113–122. ACM, 2008. Journal version in Journal of the ACM (JACM), 2015.

[GLS$^+$23] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. In *Annual International Cryptology Conference*, pages 193–226. Springer, 2023.

[GMT$^+$25] Markos Georghiades, Andrew Milson, Justin Thaler, Andrew Tretyakov, Julius Zhang, and Michael Zhu. 64-bit proving for Jolt, without a slowdown. https://a16zcrypto.com/posts/article/64-bit-proving-jolt/, 2025. a16z crypto blog post (October 15, 2025).

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[Gru24] Angus Gruen. Some improvements for the piop for zerocheck. Cryptology ePrint Archive, 2024.

[GVW02] Oded Goldreich, Salil Vadhan, and Avi Wigderson. On interactive proofs with a laconic prover. *computational complexity*, 11(1):1–53, 2002.

[GW20] Ariel Gabizon and Zachary Williamson. Proposal: The TurboPlonk program syntax for specifying SNARK programs, 2020.

[KS22] Abhiram Kothapalli and Srinath Setty. SuperNova: Proving universal machine executions without universal circuits. Cryptology ePrint Archive, Paper 2022/1758, 2022.

[KS24a] Abhiram Kothapalli and Srinath Setty. HyperNova: Recursive arguments for customizable constraint systems. In *Annual International Cryptology Conference*, pages 345–379, 2024.

[KS24b] Abhiram Kothapalli and Srinath Setty. NeutronNova: Folding everything that reduces to zero-check. Cryptology ePrint Archive, Paper 2024/1606, 2024.

[Lee21] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *Theory of Cryptography Conference*, pages 1–34. Springer, 2021.

[LFKN90] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, October 1990.

[Lip89] Richard J. Lipton. New directions in testing. In Joan Feigenbaum and Michael Merritt, editors, *Distributed Computing And Cryptography, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, October 4-6, 1989*, volume 2 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 191–202. DIMACS/AMS, 1989.

[NS25] Wilson Nguyen and Srinath Setty. Neo: Lattice-based folding scheme for ccs over small fields and pay-per-bit commitments. *Cryptology ePrint Archive*, 2025.

[NTZ25] Vineet Nair, Justin Thaler, and Michael Zhu. Proving CPU executions in small space. Cryptology ePrint Archive, Paper 2025/611, 2025.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[Sha92] Adi Shamir. IP=PSPACE. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.

[ST25] Srinath Setty and Justin Thaler. Twist and shout: Faster memory checking arguments via one-hot addressing and increments. Cryptology ePrint Archive, Paper 2025/105, 2025.

[STW24] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with Lasso. Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), 2024.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2013.

[WTS$^+$18] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[Xop] XopMC. secp256k1-x64-avx: very fast (not secure) implementation of arithmetic on curve secp256k1 on x86_64. https://github.com/XopMC/secp256k1-x64-avx. README benchmark: Montgomery mul = 49 cycles/op on Intel Core i7-6700 (Skylake); 93 cycles/op on Intel Core i3-2328M (Sandy Bridge).

[XZZ$^+$19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2019.

[ZGK$^+$17] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2017.