# Spartan: Efficient and general-purpose zkSNARKs without trusted setup

Srinath Setty
*Microsoft Research*

**Abstract**

This paper introduces Spartan, a new family of zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) for the rank-1 constraint satisfiability (R1CS), an NP-complete language that generalizes arithmetic circuit satisfiability. A distinctive feature of Spartan is that it offers the first zkSNARKs without trusted setup (i.e., transparent zkSNARKs) for NP where verifying a proof incurs sub-linear costs—without requiring uniformity in the NP statement's structure. Furthermore, Spartan offers zkSNARKs with a time-optimal prover, a property that has remained elusive for nearly all zkSNARKs in the literature.

To achieve these results, we introduce new techniques that we compose with the sum-check protocol, a seminal interactive proof protocol: (1) *computation commitments*, a primitive to create a succinct commitment to a description of a computation; this technique is crucial for a verifier to achieve sub-linear costs after investing a one-time, public computation to preprocess a given NP statement; (2) SPARK, a cryptographic compiler to transform any existing extractable polynomial commitment scheme for multilinear polynomials to one that efficiently handles *sparse* multilinear polynomials; this technique is critical for achieving a time-optimal prover; and (3) a compact encoding of an R1CS instance as a low-degree polynomial. The end result is a public-coin succinct interactive argument of knowledge for NP (which can be viewed as *a succinct variant of the sum-check protocol*); we transform it into a zkSNARK using prior techniques. By applying SPARK to different commitment schemes, we obtain several zkSNARKs where the verifier's costs and the proof size range from $O(\log^2 n)$ to $O(\sqrt{n})$ depending on the underlying commitment scheme ($n$ denotes the size of the NP statement). These schemes do not require a trusted setup except for one that requires a universal trusted setup.

We implement Spartan as a library in about 8,000 lines of Rust. We use the library to build a transparent zkSNARK in the random oracle model where security holds under the discrete logarithm assumption. We experimentally evaluate it and compare with recent zkSNARKs for R1CS instance sizes up to $2^{20}$ constraints. Among schemes without trusted setup, Spartan offers the fastest prover with speedups of 36–152$\times$ depending on the baseline, produces proofs that are shorter by 1.2–416$\times$, and incurs the lowest verification times with speedups of 3.6–1326$\times$. When compared to the state-of-the-art zkSNARK with trusted setup, Spartan's prover is 2$\times$ faster for arbitrary R1CS instances and 16$\times$ faster for data-parallel workloads.

## 1 Introduction

We revisit the problem of designing zero-knowledge succinct non-interactive arguments of knowledge (zkSNARKs) [22, 48] for the complexity class NP: they enable a computationally-bounded prover to convince the membership of a problem instance in an NP language by producing a proof—*without* revealing anything besides the validity of the statement. Furthermore, the proof size and the verifier's costs are sub-linear in the size of the statement. We are motivated to design zkSNARKs because they enable many applications that involve various forms of delegation of computation for scalability or privacy [12, 26, 29, 31, 38, 39, 41, 46, 59, 61, 70, 73–79, 87].

Specifically, we are interested in zkSNARKs that prove the satisfiability of R1CS instances over a finite field $\mathbb{F}$ (an NP-complete language that generalizes arithmetic circuit satisfiability; see §2.1 for details): given a problem instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$, we desire a proof that demonstrates the knowledge of a witness $w$ such that $\mathtt{Sat}_{\mathrm{R1CS}}(\mathbb{x}, w) = 1$.[1] We desire zkSNARKs for R1CS because there exist efficient toolchains to transform high-level applications of interest to R1CS [13, 15, 18, 31, 60, 70, 73, 77, 83].

There are many approaches to construct such arguments in the literature, starting with the work of Kilian [58] who provided the first construction of a succinct interactive argument protocol by employing probabilistically checkable proofs (PCPs) [5–7, 42, 44, 54] in conjunction with Merkle trees [67]. Micali [68] made a similar protocol non-interactive in the random oracle model, thereby obtaining the first zkSNARK. Unfortunately, the underlying PCP machinery remains extremely expensive for the prover and the verifier—despite foundational advances [14, 19–21].

Thus, the first works with an explicit motivation to make proof systems practical [38, 74, 76, 77, 79] refine and implement interactive protocols of Ishai et al. [55] and Goldwasser et al. [49], which do not require asymptotically-efficient PCPs. The principal downside is that they achieve practicality for only a restricted class of NP statements.

Gennaro, Gentry, Parno, and Raykova (GGPR) [47] address the above issue with a new characterization of NP called *quadratic arithmetic programs (QAPs)*. By building on the work of Ishai et al. [55], Groth [50], and Lipmaa [65], GGPR construct a zkSNARK for R1CS in which the prover's running time is $O(n \log n)$, the size of a proof is $O(1)$, and the verifier incurs $O(|io|)$ computation to verify a proof, where $n$ is the size of the statement, and $io$ denotes the public input and output. Unfortunately, GGPR's zkSNARK requires a per-statement *trusted setup* that produces an $O_\lambda(n)$-sized structured common reference string and the trapdoor used in the setup process must be kept secret to ensure soundness. Relying on such a trusted setup is often infeasible, especially for applications that do not have trusted authorities. There exist several advances atop GGPR, but they retain a trusted setup [15, 18, 23, 51, 52, 70], or require interaction [75].

The above state of affairs has motivated another class of works, called *transparent* zkSNARKs, that aim to eliminate the requirement of a trusted setup. They prove security in the random oracle model, which is acceptable in practice. First, Hyrax [84] extends a line of work [38, 78–82] that refines the doubly-efficient interactive proofs (IPs) of Goldwasser et al. [49]. Second, STARK [10] and Aurora [16] build on interactive oracle proofs (IOPs) [17, 71]. Third, Ligero [3] builds on the "MPC in the head" paradigm [56]. Fourth, Bulletproofs [32] builds on the work of Bootle et al. [27].

Unfortunately, they face the following problems.

- The computational model of Hyrax [83] is layered arithmetic circuits, where the verifier's costs and the proof sizes scale linearly in the depth of the circuit. Converting an arbitrary circuit into a layered form can increase its size quadratically [49],[2] so Hyrax

[1]Although we use the word "proof", we mean proofs that are computationally sound [30].

[2]For a depth-$d$ circuit, converting to a layered form increases the circuit size by a factor of $O(d)$.

is restricted to low-depth circuits. Also, Hyrax [83] achieves sub-linear verification costs only for circuits with a uniform structure (e.g., data-parallel circuits).

- STARK [10] requires circuits with a sequence of identical sub-circuits, otherwise it does not achieve sub-linear verification costs. Any circuit can be converted to this form [13, 15], but the transformation increases circuit sizes by 10–1000$\times$, which translates to a similar factor increase in the prover's costs [83].
- Ligero [3], Bulletproofs [33], and Aurora [16] incur $O(n)$ verification costs.

Our work addresses these problems.

## 1.1 Summary of contributions

This paper presents a new family of zkSNARKs, which we call *Spartan*, for proving the satisfiability of NP statements expressed in R1CS. Spartan offers the first transparent zk-SNARK that achieves sub-linear verification costs for arbitrary NP statements.[3] Spartan also offers zkSNARKs with a time-optimal prover, a property that has remained difficult to achieve in nearly all prior zkSNARKs.

In a nutshell, Spartan introduces a new public-coin succinct interactive argument of knowledge where the verifier incurs sub-linear costs for arbitrary R1CS instances by employing *computation commitments* (which we describe below). Our argument makes a black box use of an extractable polynomial commitment scheme in conjunction with an information-theoretic protocol, so its soundness holds under the assumptions needed by the polynomial commitment scheme (there exist many polynomial commitment schemes that can be instantiated under standard cryptographic assumptions [32, 84, 86]). The interactive argument is public-coin, so we add zero-knowledge using existing compilers [84, 85, 88], which themselves build on prior theory [9, 35, 40]. We then make the resulting zero-knowledge argument of knowledge non-interactive in the random oracle model using the Fiat-Shamir transform [45]. Since our interactive argument employs a polynomial commitment scheme as a black box, we obtain a family of zkSNARKs where each variant employs a different polynomial commitment scheme.

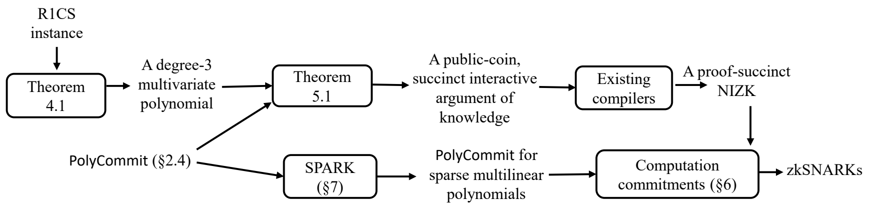
In more detail, Spartan makes the following contributions.

**(1) A new family of public-coin succinct interactive arguments of knowledge.** Our core insight is that the sum-check protocol [66], a seminal interactive proof protocol (where soundness holds unconditionally), when applied to a suitably-constructed low-degree polynomial yields a powerful—but *highly inefficient*—interactive proof protocol, but the inefficiency can be tamed with new techniques. Specifically, we introduce three techniques (Figure 1 offers a visual depiction of how these techniques work together):

(i) *Computation commitments*, a primitive for creating succinct cryptographic commitments to a mathematical description of an NP statement, which is critical for achieving sub-linear verification costs.

Achieving sub-linear verification costs appears fundamentally unrealizable because the verifier must process an NP statement for which the proof is produced before it can verify a purported proof. Our observation is that this cost can be made sub-linear

[3]To our knowledge, short PCP-based transparent zkSNARKs [58, 68] do not achieve sub-linear verification costs unless one uses uniform circuits, which is undesirable as noted above.


R1CS instance
Theorem 4.1
A degree-3 multivariate polynomial
Theorem 5.1
A public-coin, succinct interactive argument of knowledge
Existing compilers
A proof-succinct NIZK
PolyCommit (§2.4)
SPARK (§7)
PolyCommit for sparse multilinear polynomials
Computation commitments (§6)
zkSNARKs
FIGURE 1—Overview of our techniques for constructing zkSNARKs.

in the size of an NP statement by introducing a *public preprocessing step*.

Specifically, our observation is that when verifying a proof under our interactive argument, the verifier must evaluate a low-degree polynomial that encodes the NP statement, which incurs $O(n)$ costs to the verifier. Our primitive, computation commitments, enables verifiably delegating the necessary polynomial evaluations to the prover. Specifically, in Spartan, the verifier reads an R1CS instance (without the *io* component) for which the proof is produced and retains a short cryptographic commitment to a set of sparse multilinear polynomials that encode the R1CS structure. Later, when producing a proof, the prover evaluates the necessary polynomials and proves that the sparse polynomial evaluations are consistent with the commitment retained by the verifier. While the verifier incurs $O(n)$ cost to compute a computation commitment, the cost is amortized over *all* future proofs produced for all R1CS instances with the same structure. This amortization is similar to that of GGPR [47]. However, unlike GGPR's trusted setup, creating a computation commitment does not involve any secret trapdoors. Section 6 provides details.

(ii) SPARK, a cryptographic compiler to transform any existing extractable polynomial commitment scheme for multilinear polynomials to one that efficiently handles sparse multilinear polynomials. Using the compiler, we obtain schemes with time-optimal costs for both creating commitments to sparse multilinear polynomials and to produce proofs of evaluations of the committed polynomials. This compiler is crucial for achieving a time-optimal prover in Spartan. In more detail, SPARK employs an existing extractable polynomial commitment scheme as a black box, and uses it in conjunction with a special-purpose zkSNARK and a carefully-constructed circuit (that employs offline memory checking techniques [4, 24, 37, 43, 73]) to efficiently prove evaluations of sparse multilinear polynomials. Section 7 provides details.

(iii) A compact encoding of an R1CS instance as a degree-3 multivariate polynomial that can be decomposed into four multilinear polynomials. The decomposition into multilinear polynomials is critical for achieving a time-optimal prover in the sum-check protocol by employing prior ideas [78, 85]. Section 4 provides details.

**(2) An optimized implementation and experimental evaluation.** We implement Spartan as a library in about 8,000 lines of Rust. We use the library to build a transparent zkSNARK that employs an extractable polynomial commitment scheme due to Wahby et al. [84] where soundness holds under the hardness of computing discrete logarithms. Our experimental evaluation demonstrates that, among schemes without trusted setup, Spartan offers the fastest prover with speedups of 36–152× depending

| | setup | prover | proof length | verifier | computational model |
|---|---|---|---|---|---|
| GGPR [47] | private | $O(n \log n)$ | $O(1)$ | $O(1)$ | R1CS |
| Libra [85] | private$^\star$ | $O(n)$ | $O(d \log n)$ | $O(d \log n)$ | uniform circuits |
| Ligero [3] | public | $O(n \log n)$ | $O(\sqrt{n})$ | $O(n)$ | arithmetic circuits |
| Hyrax [82] | public | $O(n + m \cdot g)$ | $O(m + \sqrt{w})$ | $O(m + \sqrt{w})$ | data-parallel circuits |
| Bulletproofs [33] | public | $O(n)$ | $O(\log n)$ | $O(n)$ | arithmetic circuits |
| STARK [10] | public | $O(n \log^2 n)$ | $O(\log^2 n)$ | $O(\log^2 n)$ | uniform circuits |
| Aurora [16] | public | $O(n \log n)$ | $O(\log^2 n)$ | $O(n)$ | R1CS |
| Fractal [36] | public | $O(n \log n)$ | $O(\log^2 n)$ | $O(\log^2 n)$ | R1CS |
| Virgo [86] | public | $O(n \log n)$ | $O(d \log n)$ | $O(d \log n)$ | uniform circuits |
| SuperSonic [32] | public | $O(n \log n)$ | $O(\log n)$ | $O(\log n)$ | arithmetic circuits |
| **Variants in Spartan's family of zkSNARKs:** | | | | | |
| Spartan$_{\text{DL}}$ | public | $O(n)$ | $O(\sqrt{n})$ | $O(\sqrt{n})$ | R1CS |
| Spartan$_{\text{KE}}$ | private$^\star$ | $O(n)$ | $O(\log^2 n)$ | $O(\log^2 n)$ | R1CS |
| Spartan$_{\text{RO}}$ | public | $O(n \log n)$ | $O(\log^2 n)$ | $O(\log^2 n)$ | R1CS |
| Spartan$_{\text{CL}}$ | public | $O(n \log n)$ | $O(\log^2 n)$ | $O(\log^2 n)$ | R1CS |

FIGURE 2—A comparison of prior and recent zkSNARKs, where $n$ denotes the size of the NP statement. For Hyrax [84], we assume a layered arithmetic circuit $\mathcal{C}$ of depth $d$, width $g$, and $\beta$ copies (i.e., $n = d \cdot g \cdot \beta$); $w$ denotes the size of a witness to $\mathcal{C}$; and $m = d \cdot \log g$. Hyrax and Spartan$_{\text{DL}}$ can achieve sub-sqrt proofs at the cost of increasing $\mathcal{V}$'s time. For Libra and Virgo, we assume a depth-$d$ layered uniform circuit. The verifier incurs $O(|io|)$ additional cost in all schemes where $io$ denotes the public inputs and outputs of the NP relation being proved. Furthermore, all transparent zkSNARKs achieve non-interactivity in the random oracle model using the Fiat-Shamir heuristic [45]. Private$^\star$ means that the trusted setup is universal. Ligero, Virgo, STARK, Aurora, Fractal, and Spartan$_{\text{RO}}$ are plausibly post-quantum secure. Finally, Spartan$_{\text{CL}}$ applies SPARK to the commitment scheme of Bünz et al. [32], but the commitment scheme requires an adaptation (§5.1).

on the baseline, produces proofs that are shorter by 1.2–416×, and incurs the lowest verification times with speedups of 3.6–1326×. When compared to the state-of-the-art zkSNARK with trusted setup, Spartan's prover is 2× faster for arbitrary R1CS instances and 16× faster for data-parallel workloads.

**(3) A unified understanding of different strands of theory.** Spartan exposes interconnections among different lines of work on probabilistic proofs—from the perspective of zkSNARKs—including doubly-efficient IPs, MIPs, and short PCPs [72, §3.2].

**(4) Improvements in zkSNARKs with universal setup.** While our focus is transparent zkSNARKs, Spartan improves on prior zkSNARKs with universal trusted setup.

By employing a different polynomial commitment scheme [69, 87], which requires q-type, knowledge of exponent assumptions, in SPARK, Spartan offers an alternative to Libra [85]; we refer to this variant as Spartan$_{\text{KE}}$. Compared to Libra, Spartan$_{\text{KE}}$ supports arbitrary R1CS instances instead of layered arithmetic circuits. Furthermore, unlike Libra, the proof sizes and the verifier's running times in Spartan$_{\text{KE}}$ do not scale linearly with the circuit depth. Finally, Libra achieves sub-linear verification costs only for low-depth uniform circuits whereas Spartan$_{\text{KE}}$ achieves sub-linear verification costs for arbitrary R1CS instances via computation commitments.

## 1.2 Additional related work

Figure 2 compares the asymptotic costs of Spartan-based zkSNARKs with other schemes.

**Recent schemes.** Following our preprint, there are three transparent zkSNARKs: Fractal [36], SuperSonic [32], and Virgo [86]. Virgo's model of computation is same as Hyrax's, so it achieves sub-linear verification costs only for low-depth, uniform circuits.

Fractal and SuperSonic achieve sub-linear verification costs for arbitrary NP statements. In these schemes, the verifier preprocesses an NP statement—without secret trapdoors—to create a commitment to the structure of the statement. In other words, they instantiate the computation commitments primitive. Unfortunately, both schemes incur orders of magnitude higher expense than Spartan (§9).

## 2 Preliminaries

We use $\mathbb{F}$ to denote a finite field (e.g., the prime field $\mathbb{F}_p$ for a large prime $p$) and $\lambda$ to denote the security parameter. We use $\texttt{negl}(\lambda)$ to denote a negligible function in $\lambda$. Throughout the paper, the depicted asymptotics depend on $\lambda$, but we elide this for brevity. We use "PPT algorithms" to refer to probabilistic polynomial time algorithms.

### 2.1 Problem instances in R1CS

Recall that for any problem instance $\mathbb{x}$, if $\mathbb{x}$ is in an NP language $\mathcal{L}$, there exists a witness $w$ and a deterministic algorithm $\texttt{Sat}$ such that:

$$\texttt{Sat}_{\mathcal{L}}(\mathbb{x}, w) = \begin{cases} 1 & \text{if } \mathbb{x} \in \mathcal{L} \\ 0 & \text{otherwise} \end{cases}$$

Alternatively, the set of tuples of the form $\langle \mathbb{x}, w \rangle$ form a set of NP relations. The subset of those for which $\texttt{Sat}_{\mathcal{L}}(\mathbb{x}, w) = 1$ are called *satisfiable instances*, which we denote as: $\mathcal{R}_{\mathcal{L}} = \{\langle \mathbb{x}, w \rangle : \texttt{Sat}_{\mathcal{L}}(\mathbb{x}, w) = 1\}$.

As an NP-complete language, we focus on the rank-1 constraint satisfiability (R1CS). As noted earlier, R1CS is a popular target for compiler toolchains that accept applications expressed in high-level languages [70, 75, 77, 83]. R1CS is implicit in the QAPs of GGPR [47], but it is used with (and without) QAPs in subsequent works [16, 64, 75].

**Definition 2.1** (R1CS instance)**.** An R1CS instance is a tuple $(\mathbb{F}, A, B, C, io, m, n)$, where $io$ denotes the public input and output of the instance, $A, B, C \in \mathbb{F}^{m \times m}$, where $m \geq |io| + 1$ and there are at most $n$ non-zero entries in each matrix.

Note that matrices $A, B, C$ are defined to be square matrices for conceptual simplicity. Below, we use the notation $z = (x, y, z)$ (where each of $x, y, z$ is a vector over $\mathbb{F}$) to mean that $z$ is a vector that concatenates the three vectors in a natural way. WLOG, we assume that $n = O(m)$ throughout the paper.

**Definition 2.2** (R1CS)**.** An R1CS instance $(\mathbb{F}, A, B, C, io, m, n)$ is said to be *satisfiable* if there exists a witness $w \in \mathbb{F}^{m-|io|-1}$ such that $(A \cdot z) \circ (B \cdot z) = (C \cdot z)$, where $z = (io, 1, w)$, $\cdot$ is the matrix-vector product, and $\circ$ is the Hadamard (entry-wise) product.

Note that R1CS generalizes arithmetic circuit satisfiability because the entries in matrices $A, B, C$ can be used to encode addition and multiplication gates over $\mathbb{F}$. Furthermore, they can be used to encode a class of degree-2 constraints of the form $L(z) \cdot R(z) = O(z)$, where $L, R, O$ are degree-1 polynomials over variables that take values specified by

$z = (io, 1, w)$. In other words, R1CS supports arbitrary fan-in addition gates, and multiplication gates that verify arbitrary bilinear relations over the entire $z$.

**Definition 2.3.** For an R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$ and a purported witness $w \in \mathbb{F}^{m-|io|-1}$, we define:

$$\mathtt{Sat}_{\mathrm{R1CS}}(\mathbb{x}, w) = \begin{cases} 1 & (A \cdot (io, 1, w) \circ (B \cdot (io, 1, w)) = (C \cdot (io, 1, w)) \\ 0 & \text{otherwise} \end{cases}$$

The set of satisfiable R1CS instances can be denoted as:

$$\mathcal{R}_{\mathrm{R1CS}} = \{\langle(\mathbb{F}, A, B, C, io, m, n), w\rangle : \mathtt{Sat}_{\mathrm{R1CS}}((\mathbb{F}, A, B, C, io, m, n), w) = 1\}$$

**Definition 2.4.** For a given R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$, the NP statement that $\mathbb{x}$ is satisfiable (i.e., $\mathbb{x} \in \mathcal{R}_{\mathrm{R1CS}}$) is of size $O(n)$.

## 2.2 Polynomials and low-degree extensions

**Definition 2.5** (Multilinear polynomial)**.** A multivariate polynomial is called a multilinear polynomial if the degree of the polynomial in each variable is at most one.

**Definition 2.6** (Low-degree polynomial)**.** A multivariate polynomial $\mathcal{G}$ over a finite field $\mathbb{F}$ is called low-degree polynomial if the degree of $\mathcal{G}$ in each variable is exponentially smaller than $|\mathbb{F}|$.

**Low-degree extensions (LDEs).** Suppose $g : \{0,1\}^m \to \mathbb{F}$ is a function that maps $m$-bit elements into an element of $\mathbb{F}$. A *polynomial extension* of $g$ is a low-degree $m$-variate polynomial $\widetilde{g}(\cdot)$ such that $\widetilde{g}(x) = g(x)$ for all $x \in \{0,1\}^m$.

A *multilinear* polynomial extension (or simply, a multilinear extension, or MLE) is a low-degree polynomial extension where the extension is a multilinear polynomial (i.e., the degree of each variable in $\widetilde{g}(\cdot)$ is at most one). Given a function $Z : \{0,1\}^m \to \mathbb{F}$, the multilinear extension of $Z(\cdot)$ is the unique multilinear polynomial $\widetilde{Z} : \mathbb{F}^m \to \mathbb{F}$. It can be computed as follows.

$$\begin{aligned} \widetilde{Z}(x_1, \ldots, x_m) &= \sum_{e \in \{0,1\}^m} Z(e) \cdot \prod_{i=1}^{m} (x_i \cdot e_i + (1 - x_i) \cdot (1 - e_i)) \\ &= \sum_{e \in \{0,1\}^m} Z(e) \cdot \widetilde{\mathrm{eq}}(x, e) \\ &= \langle (Z(0), \ldots, Z(2^m - 1)), (\widetilde{\mathrm{eq}}(x, 0), \ldots, \widetilde{\mathrm{eq}}(x, 2^m - 1)\rangle \end{aligned}$$

Note that $\widetilde{\mathrm{eq}}(x, e) = \prod_{i=1}^{m}(e_i \cdot x_i + (1 - e_i) \cdot (1 - x_i))$, which is the MLE of the following function:

$$\mathrm{eq}(x, e) = \begin{cases} 1 & \text{if } x = e \\ 0 & \text{otherwise} \end{cases}$$

For any $r \in \mathbb{F}^m$, $\widetilde{Z}(r)$ can be computed in $O(2^m)$ operations in $\mathbb{F}$ [78, 80].

**Dense representation for multilinear polynomials.** Since the MLE of a function is unique, it offers the following method to represent any multilinear polynomial. Given a multilinear polynomial $\mathcal{G}(\cdot) : \mathbb{F}^m \to \mathbb{F}$, it can be represented uniquely by the list of evaluations of $\mathcal{G}(\cdot)$ over the Boolean hypercube $\{0,1\}^m$ (i.e., a function that maps $\{0,1\}^m \to \mathbb{F}$). We denote such a representation of $\mathcal{G}$ as $\texttt{DenseRepr}(\mathcal{G})$.

**Definition 2.7.** A multilinear polynomial $\mathcal{G} : \mathbb{F}^m \to \mathbb{F}$ is a sparse multilinear polynomial if $|\texttt{DenseRepr}(\mathcal{G})|$ is sub-linear in $O(2^m)$. Otherwise, it is a dense multilinear polynomial.

## 2.3 A polynomial commitment scheme for multilinear polynomials

We adopt our definitions from Bünz et al. [32] where they generalize the definition of Kate et al. [57] to allow interactive evaluation proofs. We also borrow their notation: in a list of arguments or returned tuples, variables before the semicolon are public and the ones after are secret; when there is no secret information, semicolon is omitted.

WLOG, below, when algorithms accept as input a multilinear polynomial, they use the dense representation of multilinear polynomials (§2.2).

A polynomial commitment scheme for multilinear polynomials is a tuple of four protocols $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$:

- $pp \leftarrow \mathsf{Setup}(1^\lambda, \mu)$: takes as input $\mu$ (the number of variables in a multilinear polynomial); produces public parameters $pp$.
- $(\mathcal{C}; \mathcal{S}) \leftarrow \mathsf{Commit}(pp; \mathcal{G})$: takes as input a $\mu$-variate multilinear polynomial over a finite field $\mathcal{G} \in \mathbb{F}[\mu]$; produces a public commitment $\mathcal{C}$ and a secret opening hint $\mathcal{S}$.
- $b \leftarrow \mathsf{Open}(pp, \mathcal{C}, \mathcal{G}, \mathcal{S})$: verifies the opening of commitment $\mathcal{C}$ to the $\mu$-variate multilinear polynomial $\mathcal{G} \in \mathbb{F}[\mu]$ with the opening hint $\mathcal{S}$; outputs a $b \in \{0,1\}$.
- $b \leftarrow \mathsf{Eval}(pp, \mathcal{C}, r, v, \mu; \mathcal{G}, \mathcal{S})$ is an interactive public-coin protocol between a PPT prover $\mathcal{P}$ and verifier $\mathcal{V}$. Both $\mathcal{V}$ and $\mathcal{P}$ hold a commitment $\mathcal{C}$, the number of variables $\mu$, a scalar $v \in \mathbb{F}$, and $r \in \mathbb{F}^\mu$. $\mathcal{P}$ additionally knows a $\mu$-variate multilinear polynomial $\mathcal{G} \in \mathbb{F}[\mu]$ and its secret opening hint $\mathcal{S}$. $\mathcal{P}$ attempts to convince $\mathcal{V}$ that $\mathcal{G}(r) = v$. At the end of the protocol, $\mathcal{V}$ outputs $b \in \{0,1\}$.

Definitions of properties of polynomial commitments as well as definitions of interactive arguments of knowledge are in an extended report [72].

## 3 The sum-check protocol: opportunities and challenges

An interactive proof is an interactive argument, where the soundness holds unconditionally. We now describe a seminal interactive proof protocol that we employ in Spartan, called the sum-check protocol [66]. Suppose there is an $\mu$-variate low-degree polynomial, $\mathcal{G} : \mathbb{F}^\mu \to \mathbb{F}$ where the degree of each variable in $\mathcal{G}$ is at most $\ell$. Suppose that a verifier $\mathcal{V}_{SC}$ is interested in checking a claim of the following form by an untrusted prover $\mathcal{P}_{SC}$:

$$T = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \dots \sum_{x_\mu \in \{0,1\}} \mathcal{G}(x_1, x_2, \dots, x_\mu)$$

Of course, given $\mathcal{G}(\cdot)$, $\mathcal{V}_{SC}$ can deterministically evaluate the above sum and verify whether the sum is $T$. But, this computation takes time exponential in $\mu$.

Lund et al. [66] describe the sum-check protocol that requires far less computation on $\mathcal{V}_{SC}$'s behalf, but provides a probabilistic guarantee. In the protocol, $\mathcal{V}_{SC}$ interacts with $\mathcal{P}_{SC}$ over a sequence of $\mu$ rounds. At the end of this interaction, $\mathcal{V}_{SC}$ outputs $b \in \{0,1\}$. The principal cost to $\mathcal{V}_{SC}$ is to evaluate $\mathcal{G}$ at a random point in its domain $r \in \mathbb{F}^{\mu}$. We denote the sum-check protocol as $b \leftarrow \langle \mathcal{P}_{SC}, \mathcal{V}_{SC}(r) \rangle(\mathcal{G}, \mu, \ell, T)$. For any $\mu$-variate polynomial $\mathcal{G}$ with degree at most $\ell$ in each variable, the following properties hold.

- **Completeness.** If $T = \sum_{x \in \{0,1\}^{\mu}} \mathcal{G}(x)$, then for a correct $\mathcal{P}_{SC}$ and for all $r \in \{0,1\}^*$, $\Pr\{\langle \mathcal{P}_{SC}(\mathcal{G}), \mathcal{V}_{SC}(r) \rangle(\mu, \ell, T) = 1\} = 1$.
- **Soundness.** If $T \neq \sum_{x \in \{0,1\}^{\mu}} \mathcal{G}(x)$, then for any $\mathcal{P}^{\star}_{SC}$ and for all $r \in \{0,1\}^*$, $\Pr_r\{\langle \mathcal{P}^{\star}_{SC}(\mathcal{G}), \mathcal{V}_{SC}(r) \rangle(\mu, \ell, T) = 1\} \leq \ell \cdot \mu / |\mathbb{F}|$.
- **Succinctness.** The communication between $\mathcal{P}_{SC}$ and $\mathcal{V}_{SC}$ is $O(\mu \cdot \ell)$ elements of $\mathbb{F}$.

**An alternate formulation.** In the rest of the paper, it is natural to view the sum-check protocol as a mechanism to reduce a claim of the form $\sum_{x \in \{0,1\}^m} \mathcal{G}(x) \stackrel{?}{=} T$ to the claim $\mathcal{G}(r) \stackrel{?}{=} e$. This is because in most cases, the verifier uses an auxiliary protocol to verify the latter claim, so this formulation makes it easy to describe our end-to-end protocols. We denote this reduction protocol with $e \leftarrow \langle \mathcal{P}_{SC}(\mathcal{G}), \mathcal{V}_{SC}(r) \rangle(\mu, \ell, T)$.

## 3.1 Challenges with using the sum-check protocol for succinct arguments

To build a succinct interactive argument of knowledge for R1CS, we need an interactive protocol for the verifier $\mathcal{V}$ to check if the prover $\mathcal{P}$ knows a witness $w$ to a given R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$ such that $\texttt{Sat}_{\texttt{R1CS}}(\mathbb{x}, w) = 1$.

At first glance, the sum-check protocol [66] seems to offer the necessary building block (it is public-coin, incurs succinct communication, etc.). However, to build a succinct interactive argument of knowledge (that can in turn be compiled into a zkSNARK), we must solve the following sub-problems:

1. **Encode R1CS instances as sum-check instances.** For any R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$, we must devise a degree-$\ell$, $\mu$-variate polynomial that sums to a specific value $T$ over $\{0,1\}^{\mu}$ *if and only if* there exists a witness $w$ such that $\texttt{Sat}_{\texttt{R1CS}}(\mathbb{x}, w) = 1$, where $\mu = O(\log m)$ and $\ell$ is a small constant (e.g., 3).
2. **Achieve communication-succinctness.** Although the sum-check protocol offers succinctness (if the first sub-problem is solved with constraints on $\mu$ and $\ell$ noted above), building a succinct interactive argument is non-trivial. This is because after the sum-check reduction, $\mathcal{V}$ must verify $\mathcal{G}(r) \stackrel{?}{=} e$. Unfortunately, $\mathcal{G}(r)$ depends on the $\mathcal{P}$'s witness $w$ to $\mathbb{x}$. Thus, a naive evaluation of $\mathcal{G}(r)$ requires $O(m)$ communication to transmit $w$. Transmitting $w$ is also incompatible with zero-knowledge.
3. **Achieve verifier-succinctness.** To compile an interactive argument to a zkSNARK, $\mathcal{V}$'s costs must be sub-linear in the size of an NP statement, but evaluating $\mathcal{G}(r)$ requires $O(n)$ computation if the statement has no structure (e.g., data-parallelism). A potential way around this fundamental issue is for $\mathcal{V}$ to preprocess the structure of the R1CS instance to accelerate all future verification of proofs for different R1CS instances with the same structure. However, to avoid any form of trusted setup, the

preprocessing must not involve secret trapdoors.

We describe prior solutions to the three sub-problems in an extended report [72].

## 4 An encoding of R1CS instances as low-degree polynomials

This section describes a compact encoding of an R1CS instance as a degree-3 multivariate polynomial. The following theorem summarizes our result, which we prove below.

**Theorem 4.1.** *For any R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$, there exists a degree-3 $\log m$-variate polynomial $\mathcal{G}$ such that $\sum_{x \in \{0,1\}^{\log m}} \mathcal{G}(x) = 0$ if and only if there exists a witness $w$ such that $\mathtt{Sat}_{R1CS}(\mathbb{x}, w) = 1$ (except for a soundness error that is negligible in $\lambda$) under the assumption that $|\mathbb{F}|$ is exponential in $\lambda$ and $m = O(\lambda)$.*

For a given R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$, let $s = \lceil \log m \rceil$. Thus, we can view matrices $A, B, C \in \mathbb{F}^{m \times m}$ as functions with the following signature: $\{0,1\}^s \times \{0,1\}^s \rightarrow \mathbb{F}$. Specifically, any entry in them can be accessed with a $2s$-bit identifier (or two $s$-bit identifiers). Furthermore, given a purported witness $w$ to $\mathbb{x}$, let $Z = (io, 1, w)$. It is natural to interpret $Z$ as a function with the following signature: $\{0,1\}^s \rightarrow \mathbb{F}$, so any element of $Z$ can be accessed with an $s$-bit identifier.

We now describe a function $F_{io}(\cdot)$ that can be used to encode $w$ such that $F_{io}(\cdot)$ exhibits a desirable behavior *if and only if* $\mathtt{Sat}_{\mathrm{R1CS}}(\mathbb{x}, w) = 1$.

$$F_{io}(x) = \left(\sum_{y \in \{0,1\}^s} A(x,y) \cdot Z(y)\right) \cdot \left(\sum_{y \in \{0,1\}^s} B(x,y) \cdot Z(y)\right) - \sum_{y \in \{0,1\}^s} C(x,y) \cdot Z(y)$$

**Lemma 4.1.** $\forall x \in \{0,1\}^s$, $F_{io}(x) = 0$ *if and only if* $\mathtt{Sat}_{R1CS}(\mathbb{x}, w) = 1$.

*Proof.* This follows from the definition of $\mathtt{Sat}_{\mathrm{R1CS}}(\mathbb{x}, w)$ (Section 2.1) and of $Z(\cdot)$. □

Unfortunately $F_{io}(\cdot)$ is a function, *not* a polynomial, so it cannot be directly used in the sum-check protocol. But, consider its polynomial extension $\widetilde{F}_{io} : \mathbb{F}^s \rightarrow \mathbb{F}$.

$$\widetilde{F}_{io}(x) = \left(\sum_{y \in \{0,1\}^s} \widetilde{A}(x,y) \cdot \widetilde{Z}(y)\right) \cdot \left(\sum_{y \in \{0,1\}^s} \widetilde{B}(x,y) \cdot \widetilde{Z}(y)\right) - \sum_{y \in \{0,1\}^s} \widetilde{C}(x,y) \cdot \widetilde{Z}(y)$$

**Lemma 4.2.** $\forall x \in \{0,1\}^s$, $\widetilde{F}_{io}(x) = 0$ *if and only if* $\mathtt{Sat}_{R1CS}(\mathbb{x}, w) = 1$.

*Proof.* For any $x \in \{0,1\}^s$, $\widetilde{F}_{io}(x) = F_{io}(x)$, so the result follows from Lemma 4.1. □

Since $\widetilde{F}_{io}(\cdot)$ is a low-degree multivariate polynomial over $\mathbb{F}$ in $s$ variables, a verifier $\mathcal{V}$ could check if $\sum_{x \in \{0,1\}^s} \widetilde{F}_{io}(x) = 0$ using the sum-check protocol with a prover $\mathcal{P}$. But, this is insufficient: $\sum_{x \in \{0,1\}^s} \widetilde{F}_{io}(x) = 0$ does not imply that $F_{io}(x)$ is zero $\forall x \in \{0,1\}^s$. This is because the $2^s$ terms in the sum might cancel each other making the final sum zero—even when some of the individual terms are not zero.

We addresses the above issue using a prior idea [8, 25, 34]. Consider:

$$Q_{io}(t) = \sum_{x \in \{0,1\}^s} \widetilde{F}_{io}(x) \cdot \widetilde{\mathrm{eq}}(t, x),$$

where $\widetilde{\mathrm{eq}}(t,x) = \prod_{i=1}^{s}(t_i \cdot x_i + (1-t_i)\cdot(1-x_i))$.

Observe that $Q_{io}(\cdot)$ is a multivariate polynomial such that $Q_{io}(t) = \widetilde{F}_{io}(t)$ for all $t \in \{0,1\}^s$. Thus, $Q_{io}(\cdot)$ is a *zero-polynomial* (i.e., it evaluates to zero for all points in its domain) *if and only if* $\widetilde{F}_{io}(\cdot)$ evaluates to zero at all points in the $s$-dimensional Boolean hypercube (and hence *if and only if* $\widetilde{F}_{io}(\cdot)$ encodes a witness $w$ such that $\mathtt{Sat}_{\mathrm{R1CS}}(\mathbb{x}, w) = 1$). To check if $Q_{io}(\cdot)$ is a zero-polynomial, it suffices to check if $Q_{io}(\tau) = 0$ where $\tau \in_R \mathbb{F}^s$. This introduces a soundness error, which we quantify below.

**Lemma 4.3.** $\Pr_{\tau}\{Q_{io}(\tau) = 0 | \exists x \in \{0,1\}^s \text{ } s.t. \text{ } \widetilde{F}_{io}(x) \neq 0\} \leq \log m/|\mathbb{F}|$

*Proof.* If $\exists x \in \{0,1\}^s$ such that $\widetilde{F}_{io}(x) \neq 0$, then $Q_{io}(t)$ is not a zero-polynomial. By the Schwartz-Zippel lemma, $Q_{io}(t) = 0$ for at most $d/|\mathbb{F}|$ values of $t$ in the domain of $Q_{io}(\cdot)$, where $d$ is the degree of $Q_{io}(\cdot)$. Here, $d = s = \log m$. □

**Proof of Theorem 4.1.** For a given R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$, define, $\mathcal{G}_{io,\tau}(x) = \widetilde{F}_{io}(x) \cdot \widetilde{\mathrm{eq}}(\tau, x)$, so $Q_{io}(\tau) = \sum_{x\in\{0,1\}^s} \mathcal{G}_{io,\tau}(x)$. Observe that $\mathcal{G}_{io,\tau}(\cdot)$ is a degree-3 $s$-variate polynomial if multilinear extensions of $A, B, C$, and $Z$ are used in $\widetilde{F}_{io}(\cdot)$. In the terminology of the sum-check protocol, $T = 0, \mu = s = \log m$, and $\ell = 3$. Furthermore, if $\tau \in_R \mathbb{F}^s$, $\sum_{x\in\{0,1\}^s} \mathcal{G}_{io,\tau}(x) = 0$ if and only $\widetilde{F_{io}}(x) = 0$ $\forall x \in \{0,1\}^s$—except for soundness error that is negligible in $\lambda$ under the assumptions noted above (lemma 4.3). This combined with lemma 4.2 implies the desired result.

## 5 A family of NIZKs with succinct proofs for R1CS

We first design an interactive argument with succinct communication costs and then compile it into a family of NIZKs in the random oracle model using prior transformations.

### 5.1 A new public-coin succinct interactive argument of knowledge

The following theorem summarizes our result in this section.

**Theorem 5.1.** *Given an extractable polynomial commitment scheme for multilinear polynomials, there exists a public-coin succinct interactive argument of knowledge where security holds under the assumptions needed for the polynomial commitment scheme and assuming $|\mathbb{F}|$ is exponential in $\lambda$ and the size parameter of R1CS instance $n = O(\lambda)$.*

To prove the above theorem, we first provide a construction of a public-coin succinct interactive argument of knowledge, and then analyze its costs and security. The proof of Theorem 4.1 established that for $\mathcal{V}$ to verify if an R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$ is satisfiable, it can check if $\sum_{x\in\{0,1\}^s} \mathcal{G}_{io,\tau}(x) = 0$. By using the sum-check protocol, we can reduce the claim about the sum to $e_x \overset{?}{=} \mathcal{G}_{io,\tau}(r_x)$ where $r_x \in \mathbb{F}^s$, so $\mathcal{V}$ needs a mechanism to evaluate $\mathcal{G}_{io,\underset{\sim}{\tau}}(r_x)$—without incurring $O(m)$ communication from $\mathcal{P}$ to $\mathcal{V}$.

Recall that $G_{io,\tau}(x) = \widetilde{F}_{io}(x) \cdot \widetilde{\mathrm{eq}}(\tau, x)$. Thus, to evaluate $G_{io,\tau}(r_x)$, $\mathcal{V}$ must evaluate $\widetilde{F}_{io}(r_x)$ and $\widetilde{eq}(\tau, r_x)$. The latter can be evaluated in $O(\log m)$ time. Furthermore, recall:

$$\widetilde{F}_{io}(r_x) = \left(\sum_{y\in\{0,1\}^s} \widetilde{A}(r_x, y) \cdot \widetilde{Z}(y)\right) \cdot \left(\sum_{y\in\{0,1\}^s} \widetilde{B}(r_x, y) \cdot \widetilde{Z}(y)\right) - \sum_{y\in\{0,1\}^s} \widetilde{C}(r_x, y) \cdot \widetilde{Z}(y)$$

To evaluate $\widetilde{F}_{io}(r_x)$, $\mathcal{V}$ needs to evaluate the following $\forall y \in \{0,1\}^s$: $\widetilde{A}(r_x, y)$, $\widetilde{B}(r_x, y)$, $\widetilde{C}(r_x, y)$, and $\widetilde{Z}(y)$. However, the evaluations of $\widetilde{Z}(y)$ for all $y \in \{0,1\}^s$ is the same as $(io, 1, w)$, so the communication from $\mathcal{P}$ to $\mathcal{V}$ is $\geq O(|w|)$. We now address this issue.

Our solution is a combination of three protocols: the sum-check protocol, a randomized mini protocol, and a polynomial commitment scheme. Our first observation is that the structure of the individual terms in $F_{x,y}(\cdot)$ evaluated at $r_x$ are in a form suitable for the application of a second instance of the sum-check protocol. Specifically, let $\widetilde{F}_{io}(r_x) = \overline{A}(r_x) \cdot \overline{B}(r_x) - \overline{C}(r_x)$, where

$$\begin{aligned}
\overline{A}(r_x) &= \sum_{y \in \{0,1\}^s} \widetilde{A}(r_x, y) \cdot \widetilde{Z}(y) \\
\overline{B}(r_x) &= \sum_{y \in \{0,1\}^s} \widetilde{B}(r_x, y) \cdot \widetilde{Z}(y) \\
\overline{C}(r_x) &= \sum_{y \in \{0,1\}^s} \widetilde{C}(r_x, y) \cdot \widetilde{Z}(y)
\end{aligned}$$

This observation opens up the following solution: the prover can make three separate claims to $\mathcal{V}$, say that $\overline{A}(r_x) = v_A$, $\overline{B}(r_x) = v_B$, and $\overline{C}(r_x) = v_C$. Then, $\mathcal{V}$ can evaluate:

$$\mathcal{G}_{io,\tau}(r_x) = (v_A \cdot v_B - v_C) \cdot \widetilde{\text{eq}}(r_x, \tau),$$

which in turn enables $\mathcal{V}$ to verify $\mathcal{G}_{io,\tau}(r_x) \stackrel{?}{=} e_x$. Of course, $\mathcal{V}$ must still verify three new claims from $\mathcal{P}$: $\overline{A}(r_x) \stackrel{?}{=} v_A$, $\overline{B}(r_x) \stackrel{?}{=} v_B$, and $\overline{C}(r_x) \stackrel{?}{=} v_C$. To do so, $\mathcal{V}$ and $\mathcal{P}$ can run three independent instances of the sum-check protocol to verify these claims. Instead, we use a prior idea [35, 84] to combine three claims into a single claim:

- $\mathcal{V}$ samples $r_A, r_B, r_C \in_R \mathbb{F}$ and computes $c = r_A \cdot v_A + r_B \cdot v_B + r_C \cdot v_C$.
- $\mathcal{V}$ uses the sum-check protocol with $\mathcal{P}$ to verify $r_A \cdot \overline{A}(r_x) + r_B \cdot \overline{B}(r_x) + r_C \cdot \overline{C}(r_x) \stackrel{?}{=} c$. In more detail, let $L(r_x) = r_A \cdot \overline{A}(r_x) + r_B \cdot \overline{B}(r_x) + r_C \cdot \overline{C}(r_x)$.

$$\begin{aligned}
L(r_x) &= \sum_{y \in \{0,1\}^s} r_A \cdot \widetilde{A}(r_x, y) \cdot \widetilde{Z}(y) + r_B \cdot \widetilde{B}(r_x, y) \cdot \widetilde{Z}(y) + r_C \cdot \widetilde{C}(r_x, y) \cdot \widetilde{Z}(y) \\
&= \sum_{y \in \{0,1\}^s} M_{r_x}(y)
\end{aligned}$$

$M_{r_x}(y)$ is an $s$-variate polynomial with degree at most 2 in each variable. In the terminology of the sum-check protocol, $\mu = s$, $\ell = 2$, and $T = c$.

**Lemma 5.1.** $\Pr_{r_A, r_B, r_C}\{r_A \cdot \overline{A}(r_x) + r_B \cdot \overline{B}(r_x) + r_C \cdot \overline{C}(r_x) = c | \overline{A}(r_x) \neq v_A \vee \overline{B}(r_x) \neq v_B \vee \overline{C}(r_x) \neq v_C\} \leq 1/|\mathbb{F}|$, *where* $c = r_A \cdot v_A + r_y \cdot v_B + r_C \cdot v_C$.

*Proof.* The LHS is a polynomial in $r_A, r_B, r_C$ of total degree 1; the same holds for the RHS. So, the desired result follows from the Schwartz-Zippel lemma. $\square$

$\mathcal{V}$ is not out of the woods. At the end of the second instance of the sum-check protocol, $\mathcal{V}$ must evaluate $M_{r_x}(r_y)$ for $r_y \in \mathbb{F}^s$:

$$\begin{aligned} M_{r_x}(r_y) &= r_A \cdot \widetilde{A}(r_x, r_y) \cdot \widetilde{Z}(r_y) + r_B \cdot \widetilde{B}(r_x, r_y) \cdot \widetilde{Z}(r_y) + r_C \cdot \widetilde{C}(r_x, r_y) \cdot \widetilde{Z}(r_y) \\ &= (r_A \cdot \widetilde{A}(r_x, r_y) + r_B \cdot \widetilde{C}(r_x, r_y) + r_C \cdot \widetilde{C}(r_x, r_y)) \cdot \widetilde{Z}(r_y) \end{aligned}$$

Observe that the only term in $M_{r_x}(r_y)$ that depends on the prover's witness is $\widetilde{Z}(r_y)$. This is because all other terms in the above expression can be computed locally by $\mathcal{V}$ using $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$ in $O(n)$ time (Section 6 discusses how to reduce the cost of those evaluations to be sub-linear in $n$). Our second observation is that to evaluate $\widetilde{Z}(r_y)$ without incurring $O(|w|)$ communication from $\mathcal{P}$ to $\mathcal{V}$, we can employ an extractable polynomial commitment scheme for multilinear polynomials (§2.3). A similar observation was made by Zhang et al. [87] in a different context.

In more detail, $\mathcal{P}$ sends a commitment to $\widetilde{w}(\cdot)$ (i.e., a multilinear extension of its purported witness) to $\mathcal{V}$ *before* the first instance of the sum-check protocol begins using an extractable polynomial commitment scheme for multilinear polynomials. To evaluate $\widetilde{Z}(r_y)$, $\mathcal{V}$ does the following. WLOG, assume $|w| = |io| + 1$. Thus, by the closed form expression of multilinear polynomial evaluations, we have:

$$\widetilde{Z}(r_y) = (1 - r_y[0]) \cdot \widetilde{w}(r_y[1..]) + r_y[0] \cdot \widetilde{(io, 1)}(r_y[1..]),$$

where $r_y[1..]$ refers to a slice of $r_y$ that excludes the the first element.

**Putting things together.** We assume that there exists an extractable polynomial commitment scheme for multilinear polynomials $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$.

- $pp \leftarrow \mathsf{Setup}(1^\lambda)$: Invoke $pp \leftarrow \mathsf{PC.Setup}(1^\lambda, \log m)$; output $pp$.
- $b \leftarrow \langle \mathcal{P}(w), \mathcal{V}(r) \rangle (\mathbb{F}, A, B, C, io, m, n)$:
  1. $\mathcal{P} : (\mathcal{C}, \mathcal{S}) \leftarrow \mathsf{PC.Commit}(pp, \widetilde{w})$ and send $\mathcal{C}$ to $\mathcal{V}$.
  2. $\mathcal{V} : \tau \in_R \mathbb{F}^{\log m}$ and send $\tau$ to $\mathcal{P}$.
  3. Let $T_1 = 0$, $\mu_1 = \log m$, $\ell_1 = 3$.
  4. $\mathcal{V}$ : Sample $r_x \in_R \mathbb{F}^{\mu_1}$
  5. **Sum-check#1.** $e_x \leftarrow \langle \mathcal{P}_{SC}(\mathcal{G}_{io,\tau}), \mathcal{V}_{SC}(r_x) \rangle (\mu_1, \ell_1, T_1)$
  6. $\mathcal{P}$: Compute $v_A = \overline{A}(r_x), v_B = \overline{B}(r_x), v_C = \overline{C}(r_x)$; send $(v_A, v_B, v_C)$ to $\mathcal{V}$.
  7. $\mathcal{V}$ : Abort with $b = 0$ if $e_x \neq (v_A \cdot v_B - v_C) \cdot \widetilde{\mathrm{eq}}(r_x, \tau)$.
  8. $\mathcal{V}$: Sample $r_A, r_B, r_C \in_R \mathbb{F}$ and send $(r_A, r_B, r_C)$ to $\mathcal{P}$.
  9. Let $T_2 = r_A \cdot v_A + r_B \cdot v_B + r_C \cdot v_C$, $\mu_2 = \log m$, $\ell_2 = 2$.
  10. $\mathcal{V}$ : Sample $r_y \in_R \mathbb{F}^{\mu_2}$
  11. **Sum-check#2.** $e_y \leftarrow \langle \mathcal{P}_{SC}(M_{r_x}), \mathcal{V}_{SC}(r_y) \rangle (\mu_2, \ell_2, T_2)$
  12. $\mathcal{P}$: $v \leftarrow \widetilde{w}(r_y[1..])$ and send $v$ to $\mathcal{V}$.
  13. $b_e \leftarrow \langle \mathcal{P}_{\mathsf{PC.Eval}}(\widetilde{w}, \mathcal{S}), \mathcal{V}_{\mathsf{PC.Eval}}(r) \rangle (pp, \mathcal{C}, r_y, v, \mu_2)$

```
14. $\mathcal{V}$: Abort with $b = 0$ if $b_e == 0$.
15. $\mathcal{V} : v_Z \leftarrow (1 - r_y[0]) \cdot \widetilde{w}(r_y[1..]) + r_y[0] \cdot \widetilde{(io, 1)}(r_y[1..])$
16. $\mathcal{V} : v_1 \leftarrow \widetilde{A}(r_x, r_y)$, $v_2 \leftarrow \widetilde{B}(r_x, r_y)$, $v_3 \leftarrow \widetilde{C}(r_x, r_y)$
17. $\mathcal{V}$ : Abort with $b = 0$ if $e_y \neq (r_A \cdot v_1 + r_B \cdot v_2 + r_C \cdot v_3) \cdot v_Z$.
18. $\mathcal{V}$ : Output $b = 1$.
```

**Choice of a polynomial commitment scheme.** There exist many extractable polynomial commitment schemes for multilinear polynomials [69, 84, 86, 87] that suffice for our purposes. The particular choice impacts the costs of our protocol as well as assumptions, so we review prior commitment schemes' costs and assumptions. An additional choice here is the scheme of Bünz et al [32] instantiated with class groups, but it requires a modification for our setting where we represent multilinear polynomials using their evaluations over a Boolean hypercube (§2.2,2.3).

| prior scheme | setup | $\mathcal{P}_{\mathsf{Eval}}$ | $\lvert\mathcal{C}\rvert$ | communication | $\mathcal{V}_{\mathsf{Eval}}$ | assumption |
|---|---|---|---|---|---|---|
| Hyrax-PC [82] | public | $O(\Gamma)$ | $O(\sqrt{\Gamma})$ | $O(\log\Gamma)$ | $O(\sqrt{\Gamma})$ | DLOG |
| vSQL-VPD [87] | private | $O(\Gamma)$ | $O(1)$ | $O(\log\Gamma)$ | $O(\log\Gamma)$ | q-PKE |
| Virgo-VPD [86] | public | $O(\Gamma\log\Gamma)$ | $O(1)$ | $O(\log^2\Gamma)$ | $O(\log^2\Gamma)$ | CRHF |

FIGURE 3—A comparison of candidate extractable polynomial commitment schemes for multilinear polynomials. Here, $\Gamma = 2^\mu$ where $\mu$ is the number of variables in the multilinear polynomial. Hyrax-PC refers to the scheme of Wahby et al. [84], which also supports shorter commitments at the cost of increasing the verifier's time. vSQL-VPD refers to the zero-knowledge variant [88] of the scheme of Zhang et al. [87]. Virgo-VPD refers to the scheme of Zhang et al. [86]. The communication column refers to the amount of communication required in the interactive argument for PC.Eval.

**Analysis of costs.** Note that the polynomials over which the sum-check protocol is run in our interactive argument decompose into several multilinear polynomials (four in the first sum-check protocol and two in the second sum-check protocol), so by employing prior ideas [78, 82, 85] to implement a linear-time prover for the sum-check protocol, the costs of our interactive argument are as follows.

- $\mathcal{P}$ incurs: (1) $O(n)$ costs to participate in the sum-check instances; (2) the cost of PC.Commit and PC.Eval for a $\log m$-variate multilinear polynomial $\widetilde{w}(\cdot)$.
- $\mathcal{V}$ incurs: (1) $O(\log m)$ costs for the sum-check instances; (2) the cost of PC.Eval for a $\log m$-variate multilinear polynomial; and (3) $O(n)$ costs to evaluate $\widetilde{A}(\cdot), \widetilde{B}(\cdot), \widetilde{C}(\cdot)$.
- The amount of communication is: (1) $O(\log m)$ in the sum-check instances; (2) the size of the commitment to $\widetilde{w}(\cdot)$ and the communication in PC.Eval for $\widetilde{w}(\cdot)$.

**Proof of Theorem 5.1.** The desired completeness of our interactive argument of knowledge follows from the completeness of the sum-check protocol and of the underlying polynomial commitment scheme. Furthermore, in all the four candidate constructions for polynomial commitment schemes, the communication from $\mathcal{P}$ to $\mathcal{V}$ is sub-linear in $m$ (Figure 4), which satisfies succinctness. Thus, we are left with proving witness-extended emulation, which we prove in the full version of the paper [72].

| PC choice | setup | prover | communication | verifier | assumption |
|---|---|---|---|---|---|
| Hyrax-PC [82] | public | $O(n)$ | $O(\sqrt{m})$ | $O(n + \sqrt{m})$ | DLOG |
| vSQL-VPD [87] | private | $O(n)$ | $O(\log m)$ | $O(n + \log m)$ | q-PKE |
| Virgo-VPD [86] | public | $O(n + m \log m)$ | $O(\log^2 m)$ | $O(n + \log^2 m)$ | CRHF |

FIGURE 4—Costs of our public-coin succinct interactive argument of knowledge instantiated with different polynomial commitment schemes. The depicted costs are for an R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$.

### 5.2 A family of NIZKs with succinct proofs for R1CS

The interactive argument from the prior subsection is public coin, so we add zero-knowledge using prior techniques [9, 40]. There are two compilers that are particularly efficient: (1) the one employed by Hyrax [84], which relies on a zero-knowledge argument protocol for proving dot-product relationships and other relationships in zero-knowledge (e.g., products); and (2) the compiler employed by Libra [85] and Virgo [86], which relies on an extractable polynomial commitment scheme. This transformation does not change asymptotics of $\mathcal{P}$, $\mathcal{V}$, or of the amount of communication (Figure 4).

Finally, since our protocol is public coin, it can be made non-interactive in the random oracle model using the Fiat-Shamir transform [45], thereby obtaining a family of NIZKs with succinct proofs for R1CS.

## 6 Computation commitments: zkSNARKs for R1CS from NIZK

The previous section constructed a family of NIZKs but not zkSNARKs. This is because the verifier incurs costs linear in the size of the R1CS instance to evaluate $\widetilde{A}, \widetilde{B}, \widetilde{C}$ at $(r_x, r_y)$. We now discuss how to achieve sub-linear verification costs.

At first blush, this appears impossible: The verifier incurs $O(n)$ costs to evaluate $\widetilde{A}, \widetilde{B}, \widetilde{C}$ at $(r_x, r_y)$ (step 16,§5.1), which is time-optimal [78, 80] if $\mathbb{x}$ has no structure (e.g., uniformity). We get around this impossibility by introducing a preprocessing step for $\mathcal{V}$. In an offline phase, $\mathcal{V}$ with access to non-*io* portions of an R1CS instance $\mathbb{x} = (\mathbb{F}, A, B, C, io, m, n)$ executes the following, where $pp_{cc} \leftarrow \mathsf{PC.Setup}(1^\lambda, 2\log m)$ and $\mathsf{PC}$ is an extractable polynomial commitment scheme for multilinear polynomials.

$\underline{\mathsf{Encode}(pp_{cc}, (A, B, C))}$:

- $(\mathcal{C}_A, \mathcal{S}_A) \leftarrow \mathsf{PC.Commit}(pp_{cc}, \widetilde{A})$
- $(\mathcal{C}_B, \mathcal{S}_B) \leftarrow \mathsf{PC.Commit}(pp_{cc}, \widetilde{B})$
- $(\mathcal{C}_C, \mathcal{S}_C) \leftarrow \mathsf{PC.Commit}(pp_{cc}, \widetilde{C})$
- Output $(\mathcal{C}_A, \mathcal{C}_B, \mathcal{C}_C)$

$\mathcal{V}$ retains commitments output by $\mathsf{Encode}$ (which need not hide the underlying polynomials, so in practice $\mathcal{S}_A = \mathcal{S}_B = \mathcal{S}_C = \perp$). The interactive argument proceeds as in the prior section except that at step 16, instead of $\mathcal{V}$ evaluating $\widetilde{A}, \widetilde{B}, \widetilde{C}$, we have:

- $\mathcal{P} : v_1 \leftarrow \widetilde{A}(r_x, r_y)$, $v_2 \leftarrow \widetilde{B}(r_x, r_y)$, $v_3 \leftarrow \widetilde{C}(r_x, r_y)$. Send $(v_1, v_2, v_3)$ to $\mathcal{V}$.

- $b_1 \leftarrow \langle \mathcal{P}_{\mathsf{PC.Eval}}(\widetilde{A}, \perp), \mathcal{V}_{\mathsf{PC.Eval}}(r) \rangle (pp_{cc}, \mathcal{C}_A, (r_x, r_y), v_1, 2\log m)$
- $b_2 \leftarrow \langle \mathcal{P}_{\mathsf{PC.Eval}}(\widetilde{B}, \perp), \mathcal{V}_{\mathsf{PC.Eval}}(r) \rangle (pp_{cc}, \mathcal{C}_B, (r_x, r_y), v_2, 2\log m)$
- $b_3 \leftarrow \langle \mathcal{P}_{\mathsf{PC.Eval}}(\widetilde{C}, \perp), \mathcal{V}_{\mathsf{PC.Eval}}(r) \rangle (pp_{cc}, \mathcal{C}_C, (r_x, r_y), v_3, 2\log m)$
- $\mathcal{V}$: Abort with $b = 0$ if $b_1 = 0 \vee b_2 = 0 \vee b_3 = 0$.

**Lemma 6.1.** *The interactive argument from Section 5.1 where step 16 is replaced with the above protocol is a public-coin succinct interactive argument of knowledge assuming* PC *is an extractable polynomial commitment scheme for multilinear polynomials.*

*Proof.* The result follows from the knowledge soundness property satisfied by PC scheme used in the Encode algorithm. □

If $\mathcal{V}$'s costs to verify the three evaluations and the added communication are sub-linear in $O(n)$, the modified interactive argument leads to a zkSNARK (if we add zero-knowledge and non-interactivity as before).

Unfortunately, existing polynomial commitment schemes do not satisfy the desired efficiency properties: (1) to participate in Eval for any of $\widetilde{A}, \widetilde{B}, \widetilde{C}$, $\mathcal{P}$ incurs at least quadratic costs i.e., $O(m^2)$; and (2) in some schemes (e.g., Hyrax-PC), the modified interactive argument does not offer improved asymptotics for the verifier.

The next section describes a scheme that meets our efficiency requirements and leads to asymptotics noted in Figure 2.

## 7 The SPARK compiler

This section describes SPARK, a new cryptographic compiler to transform an existing extractable polynomial commitment scheme for dense multilinear polynomials to one that can efficiently handle sparse multilinear polynomials.

For ease of exposition, we focus on describing SPARK that applies to $2\log m$-variate sparse polynomials $\widetilde{A}, \widetilde{B}, \widetilde{C}$ (where their dense representation is of size $\leq n$) from Section 5.1, but our result generalizes to other sparse multilinear polynomials.

### 7.1 SPARK-naive: A straw-man solution

To present our solution, we describe a straw-man that helps introduce the necessary building blocks as well as articulate difficulties addressed by SPARK. We recall Hyrax [84], a zkSNARK that achieves sub-linear verification costs for uniform circuits, specifically data-parallel circuits. The prover's costs in Hyrax can be made linear in the circuit size using subsequent ideas [85]. Furthermore, the verifier's costs are $O(d \log n + e)$ where $d$ is the depth of the circuit and $e$ is the cost to the verifier to participate in PC.Eval to evaluate a $\log |w|$-variate multilinear polynomial where $w$ is a witness to the circuit.

**Details.** Let $M$ denote one of $\{A, B, C\}$ and let $s = \log m$, so $\mu = 2s$. Recall the closed-form expression for multilinear polynomial evaluations at $r \in \mathbb{F}^{\mu}$.

$$\widetilde{M}(r) = \sum_{i \in \{0,1\}^{\mu} \,::\, M(i) \neq 0} M(i) \cdot \widetilde{\mathrm{eq}}(i, r) \tag{1}$$

The above sum has at most $n$ terms since $M(i) \neq 0$ for at most $n$ values of $i$. Also, each entry in the sum can be computed with $O(\mu)$ multiplications. Consider the following circuit to evaluate $\widetilde{M}(r)$.

A $O(\log \mu)$-depth circuit with $O(n \cdot \mu)$ gates that:

- Takes as witness the list of $n$ tuples of the form $(i, M(i)) :: M(i) \neq 0$, where each $i$ is represented with a vector of $\mu$ elements of $\mathbb{F}$, so each entry in the list is $\mu + 1$ elements of $\mathbb{F}$ (in other words, the witness is a $\log(n \cdot (\mu + 1))$-variate multilinear polynomial whose dense representation is the above list of tuples);
- Takes as public input $r \in \mathbb{F}^{\mu}$;
- Asserts that in each of the $n$ tuples, the first $\mu$ elements are either 0 or 1.
- Computes $v \leftarrow \widetilde{M}(r)$ using Equation 1;
- Outputs $v$

Note that the above circuit is uniform: there are $n$ identical copies of a sub-circuit, where each sub-circuit computes $O(\mu)$ multiplications; the outputs of these sub-circuits is fed into a binary tree of addition gates to compute the final sum. Furthermore, there is no sharing of witness elements across data-parallel units, so it truly data-parallel.

**Construction.** Given an extractable polynomial commitment scheme $\mathsf{PC}$ for multilinear polynomials, we build a scheme for sparse multilinear polynomials as follows.

$\mathsf{PC}^{\text{naive}}$:

- $pp \leftarrow \mathsf{Setup}(1^{\lambda}, \mu, n)$: $\mathsf{PC.Setup}(1^{\lambda}, \log((\mu + 1) \cdot n))$
- $(\mathcal{C}; \mathcal{S}) \leftarrow \mathsf{Commit}(pp; \widetilde{M})$: $\mathsf{PC.Commit}(pp, \mathcal{D})$, where $\mathcal{D}$ is the unique $\log((\mu + 1) \cdot n)$-variate multilinear polynomial whose dense representation is the list of tuples $(i, M(i)) ::$ $M(i) \neq 0$ and each entry is $(\mu + 1)$ elements of $\mathbb{F}$.
- $b \leftarrow \mathsf{Open}(pp, \mathcal{C}, \widetilde{M}, \mathcal{S})$: $\mathsf{PC.Open}(pp, \mathcal{C}, \mathcal{D}, \mathcal{S})$, where $\mathcal{D}$ is defined as above.
- $b \leftarrow \mathsf{Eval}(pp, \mathcal{C}, r, v, \mu, n; \widetilde{M}, \mathcal{S})$: $\mathcal{P}$ and $\mathcal{V}$ use Hyrax to verify the claim that $\widetilde{M}(r) = v$ using the circuit described above.

**Analysis of costs.** Recall that computing $\widetilde{M}(r)$ for $M \in \{A, B, C\}$ and $r \in \mathbb{F}^{\mu}$ takes $O(n)$ costs. The principal downside of $\mathsf{PC}^{\text{naive}}$ is it imposes an asymptotic overhead over its underlying commitment scheme for dense multilinear polynomials.

For example, with Hyrax-PC as the underlying commitment scheme, the prover with $\mathsf{PC}^{\text{naive}}$ incurs $O(n \log n)$ costs to prove an evaluation of a committed sparse multilinear polynomial. This is because the prover must prove the satisfiability of a circuit of size $O(n \cdot \mu)$ as well as prove the evaluations of a constant number of $(\log(n \cdot (\mu + 1)))$-variate multilinear polynomials. This slowdown is also significant in practice (§8).

**Lemma 7.1.** *$\mathsf{PC}^{naive}$ is a polynomial commitment scheme for multilinear polynomials with the costs noted above.*

*Proof.* Completeness follows from the completeness of PC and Hyrax. Binding follows from the uniqueness of the dense representation of a sparse multilinear polynomial. Knowledge soundness follows from the witness-extended emulation offered by Hyrax and PC.Eval. The claimed prover's slowdown follows from the costs of Hyrax and PC applied to a constant number of $(\log(n \cdot (\mu + 1)))$-variate multilinear polynomials. $\square$

## 7.2 Eliminating asymptotic overheads by leveraging memory checking

We now improve on the straw-man scheme by devising an $O(n)$-sized circuit for sparse polynomial evaluation. Naturally, the size of the witness to the circuit is also of size $O(n)$. This allows SPARK to achieve a linear-time prover if the underlying polynomial commitment scheme offers linear-time costs for the prover [69, 84]. More generally, when transforming an existing polynomial commitment scheme that meets our requirements (§2.3), SPARK does not add asymptotic overheads to the prover for proving the evaluations of committed sparse multilinear polynomials.

Observe that for $M \in \{A, B, C\}$, $M \in \mathbb{F}^{m \times m}$ and any $r \in \mathbb{F}^{\mu}$, we can rewrite the evaluation of $\widetilde{M}(r)$ as follows. In our context $\mu = 2 \log m$, interpret $r$ as a tuple $(r_x, r_y)$ where $r_x, r_y \in \mathbb{F}^s$ and $s = \log m = \mu/2$. Thus, we can rewrite Equation 1 as:

$$\widetilde{M}(r_x, r_y) = \sum_{(i,j) \in (\{0,1\}^s, \{0,1\}^s) \,::\, M(i,j) \neq 0} M(i,j) \cdot \widetilde{\text{eq}}(i, r_x) \cdot \widetilde{\text{eq}}(j, r_y)$$

In our context, the above sum still contains $n$ terms. Also, computing each entry in the sum still requires $(\mu + 1)$ multiplications over $\mathbb{F}$. However, it is possible to compute a table of evaluations of $\widetilde{\text{eq}}(i, r_x)$ for all $i \in \{0,1\}^s$ in $O(2^s) = O(m)$ time. Similarly, it is possible to compute evaluations of $\widetilde{\text{eq}}(j, r_y)$ for all $j \in \{0,1\}^s$ in $O(m)$ time.

Unfortunately, this observation is insufficient: even though these tables can be computed in $O(m)$ time, the sum is taken over the list of $(i,j) \in (\{0,1\}^s, \{0,1\}^s)$ where $M(i,j) \neq 0$ and for an arbitrary $2s$-variate sparse multilinear polynomial, such a list has no structure, so computing the sum requires $n$ random accesses into two tables each with $m$ entries. We could attempt to build a circuit that supports RAM operations. Unfortunately, existing techniques to encode RAM in circuits incur a logarithmic blowup or constants that in practice are larger than a logarithmic blowup.

For $m$ RAM operations over a memory of size $m$,

- Pantry [31], using Merkle trees, trees [24, 67], offers a circuit of size $O(m \log m)$.
- Buffet [83], using permutation networks [13], offers a circuit of size $O(m \log m)$ with constants smaller than the ones in Pantry.
- vRAM [89] offers an $O(m)$-sized circuit with a constant of $\log |\mathbb{F}|$ (to encode consistency checks over a memory transcript), so, in practice, this does not improve on the straw-man. Other downsides: (1) it only supports 32-bit sized memory cells, whereas we need a memory over elements of $\mathbb{F}$; (2) nearly all of the circuit's non-deterministic witness must be committed by $\mathcal{P}$ during circuit evaluation.

Our solution specializes and improves upon a recent implementation of offline memory checking techniques [24] in Spice [73], which builds circuits to encode operations on persistent storage with serializable transactions. The storage abstraction can be used as a memory abstraction where for $m$ operations, the circuit is of size $O(m)$, but the constants are worse than those of VRAM: $\geq 1000$ (to encode an elliptic-curve based multiset collision-resistant hash function for each memory operation). We get around this issue by designing an offline memory checking primitive via a new randomized check that only uses public coins. Furthermore, unlike a vRAM-based solution, most of the non-deterministic witness needed by the circuit can be created by PC.Commit (i.e., by the Encode algorithm in the context of computation commitments).

### 7.2.1 An $O(n)$-sized circuit for evaluating $\widetilde{M}$

We now describe an $O(n)$-sized circuit to compute an evaluation of $\widetilde{M}$. We prove that the circuit indeed computes the correct evaluation of the sparse polynomial in lemma 7.5. In the description of the circuit, we assume hash functions $H$ and $\mathcal{H}$, which are defined below (Equations 2 and 3). Before we describe the circuit for polynomial evaluation, we specify an encoding of sparse polynomials that our circuit leverages.

**Encoding sparse polynomials.** Given a sparse polynomial $\widetilde{M}$ (e.g., $\widetilde{M} \in \{\widetilde{A}, \widetilde{B}, \widetilde{C}\}$), we encode it using three vectors of size $n$ as follows. Since $\widetilde{M}$ is represented by $n$ tuples of the form $(i, j, M(i,j))$, where each tuple has 3 elements of $\mathbb{F}$ such that $M(i,j) \neq 0$. Note that this encoding differs from the encoding in the straw-man where each $i$ and $j$ were encoded using a vector of $s$ elements of $\{0,1\} \in \mathbb{F}$. The encoding here essentially packs $s$ bits in $i$ (or $j$) into a single element of $\mathbb{F}$ in the obvious way, which works because $s < \log |\mathbb{F}|$. In some canonical order, let $row, col, val$ be three vectors that encode the above $n$ tuples such that for $k \in [0, n-1]$ $row(k) = i, col(k) = j, val(k) = M(i,j)$.

**Encoding metadata for memory checking: "Memory in the head".** The circuit below takes as witness additional metadata about $\widetilde{M}$ (besides $row, col, val$ introduced above). This metadata accelerates memory checking during the evaluation of $\widetilde{M}(r)$.

The metadata is in the form of six vectors: $\textit{read-ts}_{row} \in \mathbb{F}^n, \textit{write-ts}_{row} \in \mathbb{F}^n$, $\textit{audit-ts}_{row} \in \mathbb{F}^m$, $\textit{read-ts}_{col} \in \mathbb{F}^n, \textit{write-ts}_{col} \in \mathbb{F}^n$, and $\textit{audit-ts}_{col} \in \mathbb{F}^m$. We specify how these are computed below with pseudocode. Note that computing this metadata only needs the following parameters: memory size (which is determined by $2^s = m$) and the sequence of addresses at which the memory is accessed (which are provided by $row$ and $col$). In a nutshell, $\textit{read-ts}_{row}$ and $\textit{write-ts}_{row}$ denote the timestamps associated with read and write operations, and $\textit{audit-ts}_{row}$ denotes the final timestamps of memory cells in the offline memory checking primitive [24, §4.1] for the address sequence specified by $row$ over a memory of size $m = O(2^s)$. Similarly, $\textit{read-ts}_{col}, \textit{write-ts}_{col}$, and $\textit{audit-ts}_{col} \in \mathbb{F}^m$ denote timestamps for the address sequence specified by $col$. They are computed as follows ($vec!$ uses Rust notation).

```
MemoryInTheHead($m, n, addrs$):
  • $\textit{read-ts} \leftarrow vec![n ; 0]$; $\textit{write-ts} \leftarrow vec![n ; 0]$; $\textit{audit-ts} \leftarrow vec![m ; 0]$; $ts \leftarrow 0$
  • for $i$ in $(0..addrs.len())$:
      • $addr \leftarrow addrs[i]$
      • $\textit{r-ts} \leftarrow \textit{audit-ts}[i]$
      • $ts \leftarrow max(ts, \textit{r-ts}) + 1$
```

    - *read-ts*$[i] \leftarrow$ *r-ts*
    - *write-ts*$[i] \leftarrow ts$
    - *audit-ts*$[addr] \leftarrow ts$
- return $(\textit{read-ts}, \textit{write-ts}, \textit{audit-ts})$

**Circuit description.** The circuit below evaluates a sparse polynomial using the encoding and preprocessed metadata described above. It relies multiset hash functions, which we now define. Unlike ECC-based multiset hash functions in Spice [73], we employ a public-coin hash function that verifies the desired multiset relationship. Specifically, we define two hash functions: (1) $h_\gamma : \mathbb{F}^3 \to \mathbb{F}$; and (2) $\mathcal{H}_\gamma : \mathbb{F}^* \to \mathbb{F}$, where $\mathbb{F}^*$ denotes a multiset with elements from $\mathbb{F}$ and $\gamma \in_R \mathbb{F}$.

$$h_\gamma(a, v, t) = a \cdot \gamma^2 + v \cdot \gamma + t \tag{2}$$

$$\mathcal{H}_\gamma(\mathcal{M}) = \Pi_{e \in \mathcal{M}}(e - \gamma) \tag{3}$$

Given $(A, V, T) \in (\mathbb{F}^\ell, \mathbb{F}^\ell, \mathbb{F}^\ell)$ for $\ell > 0$, we define a map $H_\gamma : (\mathbb{F}^\ell, \mathbb{F}^\ell, \mathbb{F}^\ell) \to \mathbb{F}^\ell$:

$$H_\gamma(A, V, T) = [h_\gamma(A[0], V[0], T[0]), \ldots, h_\gamma(A[\ell - 1], V[\ell - 1], T[\ell - 1])]$$

We capture the soundness errors of these hash functions in lemma 7.2 and lemma 7.3.

An $O(n)$-sized, $O(\log n)$-depth circuit ($\mathit{Circuit}_{\text{eval-opt}}$).

- Takes as witness the following lists (Hyrax can accept witness in separate lists).
    1. a succinct description of $\widetilde{M}$: three lists *row*, *col*, *val*, where each list has $n$ entries.
    2. two lists $e_{row}, e_{col}$, where each list contains $n$ elements of $\mathbb{F}$.
    3. six lists: *read-ts*$_{row}$,*read-ts*$_{col}$,*write-ts*$_{row}$, *write-ts*$_{col}$,*audit-ts*$_{row}$, and *audit-ts*$_{col}$. The first four are of size $n$ and the last two are of size $m$; each entry is an element of $\mathbb{F}$.
    4. two challenges $\gamma_1, \gamma_2 \in \mathbb{F}$.
- Takes as public input $r = (r_x, r_y) \in \mathbb{F}^\mu$;
- Output $\widetilde{M}(r)$ using $v \leftarrow \sum_{k=0}^{n-1} val[k] \cdot e_{row}[k] \cdot e_{col}[k]$.
- Memory checking for $e_{row}$:
    - $mem_{row} \leftarrow [\widetilde{\text{eq}}(0, r_x), \ldots, \widetilde{\text{eq}}(m-1, r_x)] \in \mathbb{F}^m$
    - $Init_{row} \leftarrow H_{\gamma_1}([0, \ldots, m-1], mem_{row}, [0, \ldots, 0]) \in \mathbb{F}^m$
    - $RS_{row} \leftarrow H_{\gamma_1}(row, e_{row}, \textit{read-ts}_{row}) \in \mathbb{F}^n$
    - $WS_{row} \leftarrow H_{\gamma_1}(row, e_{row}, \textit{write-ts}_{row}) \in \mathbb{F}^n$
    - $Audit_{row} \leftarrow H_{\gamma_1}([0, \ldots, m-1], mem_{row}, \textit{audit-ts}_{row}) \in \mathbb{F}^m$
    - Assert $\mathcal{H}_{\gamma_2}(Init_{row}) \cdot \mathcal{H}_{\gamma_2}(WS_{row}) = \mathcal{H}_{\gamma_2}(RS_{row}) \cdot \mathcal{H}_{\gamma_2}(Audit_{row})$
- Memory checking for $e_{col}$:

- $mem_{col} \leftarrow [\widetilde{\text{eq}}(0, r_y), \ldots, \widetilde{\text{eq}}(m-1, r_y)] \in \mathbb{F}^m$
- Let $Init_{col} \leftarrow H_{\gamma_1}([0, \ldots, m-1], mem_{col}, [0, \ldots, 0]) \in \mathbb{F}^m$
- Let $RS_{col} \leftarrow H_{\gamma_1}(col, e_{col}, \textit{read-ts}_{col}) \in \mathbb{F}^n$
- Let $WS_{col} \leftarrow H_{\gamma_1}(col, e_{col}, \textit{write-ts}_{col}) \in \mathbb{F}^n$
- Let $Audit_{col} \leftarrow H_{\gamma_1}([0, \ldots, m-1], mem_{col}, \textit{audit-ts}_{col}) \in \mathbb{F}^m$
- Assert $\mathcal{H}_{\gamma_2}(Init_{col}) \cdot \mathcal{H}_{\gamma_2}(WS_{col}) = \mathcal{H}_{\gamma_2}(RS_{col}) \cdot \mathcal{H}_{\gamma_2}(Audit_{col})$

**Lemma 7.2.** *For any two pairs $(a_1, v_1, t_1) \in \mathbb{F}^3$ and $(a_2, v_2, t_2) \in \mathbb{F}^3$, $\Pr_\gamma\{h_\gamma(a_1, v_1, t_1) = h_\gamma(a_2, v_2, t_2) | (a_1, v_1, t_1) \neq (a_2, v_2, t_2)\} \leq 3/|\mathbb{F}|$.*

*Proof.* This follows from the Schwartz-Zippel lemma. □

**Lemma 7.3.** *For any $\ell > 0$, $(A_1, V_1, T_1) \in (\mathbb{F}^\ell, \mathbb{F}^\ell, \mathbb{F}^\ell)$ and $(A_2, V_2, T_2) \in (\mathbb{F}^\ell, \mathbb{F}^\ell, \mathbb{F}^\ell)$ $\Pr_\gamma\{\exists i :: H_\gamma(A_1, V_1, T_1)[i] = H_\gamma(A_2, V_2, T_2)[i] | (A_1, V_1, T_1) \neq (A_2, V_2, T_2)\} \leq 3 \cdot \ell/|\mathbb{F}|$.*

*Proof.* This follows from a standard union bound with the result of the lemma 7.2. □

**Lemma 7.4.** *For any two multisets $\mathcal{M}_1, \mathcal{M}_2$ of size $\ell$ over $\mathbb{F}$,*

$$\Pr_\gamma\{\mathcal{H}_\gamma(\mathcal{M}_1) = \mathcal{H}_\gamma(\mathcal{M}_2) | \mathcal{M}_1 \neq \mathcal{M}_2\} \leq \ell/|\mathbb{F}|$$

*Proof.* This follows from the Schwartz-Zippel lemma. □

**Lemma 7.5.** *Assuming that $|\mathbb{F}|$ is exponential in $\lambda$ and $n = O(\lambda)$, for any $2\log m$-variate multilinear polynomial $\widetilde{M}$ whose dense representation is of size at most $n$ and for any given $e_{row}, e_{col} \in \mathbb{F}^n$,*

$$\Pr_{\gamma_1,\gamma_2}\{Circuit_{\textit{eval-opt}}(w, (\gamma_1, \gamma_2), r) = v | \widetilde{M}(r) \neq v\} \leq \texttt{negl}(\lambda),$$

*where $w = (row, col, val, e_{row}, e_{col}, \mathsf{MemoryInTheHead}(m, n, row), \mathsf{MemoryInTheHead}(m, n, col))$ and $(row, col, val)$ denotes the dense representation of $\widetilde{M}$.*

*Proof.* This follows from the soundness of the memory checking primitive [24] and the collision-resistance of the underlying hash functions used (lemmas 7.4 and 7.3). □

### 7.2.2 Construction of a polynomial commitment scheme

Given an extractable polynomial commitment scheme PC for multilinear polynomials, we build a scheme for sparse multilinear polynomials as follows.

Note that our focus is on designing a polynomial commitment scheme for efficiently realizing computation commitments (§6). For this purpose, the Spartan verifier runs the Commit algorithm (of the sparse polynomial commitment scheme) as part of the Encode algorithm, so unlike the general setup of polynomial commitments, the entity creating a commitment is the verifier itself (not an untrusted entity). As a result, the additional memory-checking metadata about the sparse polynomial as part of Commit is created by the verifier, so we do not need to verify that the timestamps are well-formed according to its specification in the MemoryInTheHead procedure as required by lemma 7.5. This is only an optimization and not a limitation. In the general setting where $\mathsf{Commit}$ (of the sparse polynomial commitment scheme) is run by an untrusted entity, we can require it to additionally produce a proof that proves that timestamps are well-formed. In the description below, given our focus on computation commitments, we omit those proofs.

<u>$\mathsf{PC}^{\text{SPARK}}$:</u>

- $pp \leftarrow \mathsf{Setup}(1^\lambda, \mu, n)$: $(\mathsf{PC.Setup}(1^\lambda, \mu)), \mathsf{PC.Setup}(1^\lambda, \log(n)))$
- $(\mathcal{C}; \mathcal{S}) \leftarrow \mathsf{Commit}(pp; \widetilde{M})$:
  - Let $(pp_m, pp_n) \leftarrow pp$
  - Let $(row, col, val)$ denote the dense representation of $\widetilde{M}$ as described in text.
    - $(\mathcal{C}_{row}, \mathcal{S}_{row}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{row})$
    - $(\mathcal{C}_{col}, \mathcal{S}_{col}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{col})$
    - $(\mathcal{C}_{val}, \mathcal{S}_{val}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{val})$
  - Let $(read\text{-}ts_{row}, write\text{-}ts_{row}, audit\text{-}ts_{row}) \leftarrow \mathsf{MemoryInTheHead}(2^{\mu/2}, n, row)$
    - $(\mathcal{C}_{read\text{-}ts_{row}}, \mathcal{S}_{read\text{-}ts_{row}}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{read\text{-}ts}_{row})$
    - $(\mathcal{C}_{write\text{-}ts_{row}}, \mathcal{S}_{write\text{-}ts_{row}}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{write\text{-}ts}_{row})$
    - $(\mathcal{C}_{audit\text{-}ts_{row}}, \mathcal{S}_{audit\text{-}ts_{row}}) \leftarrow \mathsf{PC.Commit}(pp_m, \widetilde{audit\text{-}ts}_{row})$
  - Let $(read\text{-}ts_{col}, write\text{-}ts_{col}, audit\text{-}ts_{col}) \leftarrow \mathsf{MemoryInTheHead}(2^{\mu/2}, n, col)$
    - $(\mathcal{C}_{read\text{-}ts_{col}}, \mathcal{S}_{read\text{-}ts_{col}}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{read\text{-}ts}_{col})$
    - $(\mathcal{C}_{write\text{-}ts_{col}}, \mathcal{S}_{write\text{-}ts_{col}}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{write\text{-}ts}_{col})$
    - $(\mathcal{C}_{audit\text{-}ts_{col}}, \mathcal{S}_{audit\text{-}ts_{col}}) \leftarrow \mathsf{PC.Commit}(pp_m, \widetilde{audit\text{-}ts}_{col})$
  - Let $\mathcal{C} \leftarrow (\mathcal{C}_{row}, \mathcal{C}_{col}, \mathcal{C}_{val}, \mathcal{C}_{read\text{-}ts_{row}}, \mathcal{C}_{write\text{-}ts_{row}}, \mathcal{C}_{audit\text{-}ts_{row}}, \mathcal{C}_{read\text{-}ts_{col}}, \mathcal{C}_{write\text{-}ts_{col}}, \mathcal{C}_{audit\text{-}ts_{col}})$
  - Let $\mathcal{S} \leftarrow (\mathcal{S}_{row}, \mathcal{S}_{col}, \mathcal{S}_{val}, \mathcal{S}_{read\text{-}ts_{row}}, \mathcal{S}_{write\text{-}ts_{row}}, \mathcal{S}_{audit\text{-}ts_{row}}, \mathcal{S}_{read\text{-}ts_{col}}, \mathcal{S}_{write\text{-}ts_{col}}, \mathcal{S}_{audit\text{-}ts_{col}})$
  - Output $(\mathcal{C}, \mathcal{S})$
- $b \leftarrow \mathsf{Open}(pp, \mathcal{C}, \widetilde{M}, \mathcal{S})$:
  - Let $(pp_m, pp_n) \leftarrow pp$.
  - Let $row, col, val$ denote dense representation of $\widetilde{M}$ as defined above.
  - Output $\mathsf{PC.Open}(pp_n, \mathcal{C}.\mathcal{C}_{row}, \widetilde{row}, \mathcal{S}.\mathcal{S}_{row}) \wedge \mathsf{PC.Open}(pp_n, \mathcal{C}.\mathcal{C}_{col}, \widetilde{col}, \mathcal{S}.\mathcal{S}_{col}) \wedge \mathsf{PC.Open}(pp_n, \mathcal{C}, \mathcal{C}_{val}, \widetilde{val}, \mathcal{S}.\mathcal{S}_{val})$
- $b \leftarrow \mathsf{Eval}(pp, \mathcal{C}, r, v, \mu, n; \widetilde{M}, \mathcal{S})$:
  - Let $(pp_m, pp_n) \leftarrow pp$ and let $(r_x, r_y) = r$, where $r_x, r_y \in \mathbb{F}^{\mu/2}$.
  - Let $row, col, val$ denote dense representation of $\widetilde{M}$ as defined above.
  - $\mathcal{P}$ :
    - Compute $e_{row}$ and $e_{col}$ with $2n$ lookups over a table of size $m = 2^{\mu/2}$. That is, $e_{row} = [\widetilde{\text{eq}}(row(0), r_x), \ldots, \widetilde{\text{eq}}(row(n-1), r_x)]$; let $e_{col} = [\widetilde{\text{eq}}(col(0), r_y), \ldots, \widetilde{\text{eq}}(col(n-1), r_y)]$.

– $(\mathcal{C}_{e_{row}}, \mathcal{S}_{e_{row}}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{e_{row}})$; send $\mathcal{C}_{e_{row}}$ to $\mathcal{V}$.
– $(\mathcal{C}_{e_{col}}, \mathcal{S}_{e_{col}}) \leftarrow \mathsf{PC.Commit}(pp_n, \widetilde{e_{col}})$; send $\mathcal{C}_{e_{col}}$ to $\mathcal{V}$.

- $\mathcal{V} : (\gamma_1, \gamma_2) \in_R \mathbb{F}^2$. Send $(\gamma_1, \gamma_2)$ to $\mathcal{P}$.
- $\mathcal{P}$ and $\mathcal{V}$ use Hyrax (with PC as the extractable polynomial commitment scheme) to verify the claim that $\widetilde{M}(r) = v$ using $\mathit{Circuit}_{\text{eval-opt}}$.

**Analysis of costs.** $\mathit{Circuit}_{\text{eval-opt}}$ is uniform because computing $\mathcal{H}$ using a binary tree of multiplications [78] constitutes nearly all of the work in the above circuit. Figure 5 depicts the costs of $\mathsf{PC}^{\text{SPARK}}$ with different choices for PC.

| PC choice | setup | $\mathcal{P}_{\mathsf{Eval}}$ | $\lvert\mathcal{C}\rvert$ | communication | $\mathcal{V}_{\mathsf{Eval}}$ |
|---|---|---|---|---|---|
| Hyrax-PC [82] | public | $O(n)$ | $O(\sqrt{n})$ | $O(\log^2 n)$ | $O(\sqrt{n})$ |
| vSQL-VPD [87] | private$^{\star}$ | $O(n)$ | $O(1)$ | $O(\log^2 n)$ | $O(\log^2 n)$ |
| Virgo-VPD [86] | public | $O(n \log n)$ | $O(1)$ | $O(\log^2 n)$ | $O(\log^2 n)$ |

FIGURE 5—Costs of $\mathsf{PC}^{\text{SPARK}}$ with different choices for PC. Here, $n$ is number of entries in the dense representation of the multilinear polynomial.

**Lemma 7.6.** *Assuming that $\mathsf{PC}^{\text{SPARK}}$.Commit is run by an honest entity, then $\mathsf{PC}^{\text{SPARK}}$ is a polynomial commitment scheme for multilinear polynomials with the costs noted.*

*Proof.* Completeness follows from the completeness of PC, Hyrax, and $\mathit{Circuit}_{\text{eval-opt}}$. Binding follows from the uniqueness of the dense representation of the sparse multilinear polynomial as $(\mathit{row}, \mathit{col}, \mathit{val})$. Knowledge soundness follows from the witness-extended emulation offered by Hyrax and PC, and from the negligible soundness error of $\mathit{Circuit}_{\text{eval-opt}}$ (lemma 7.5). Finally, the claimed costs follow from the cost model of Hyrax and of PC applied to a constant number of $O(\log n)$-variate multilinear polynomials. □

### 7.2.3 Optimizations

We now describe many optimizations to SPARK to reduce constants.

1. Instead of using Hyrax as a black box, we tailor it for $\mathit{Circuit}_{\text{eval-opt}}$ using prior ideas [78]. This reduces overall costs significantly. We also do not need Hyrax's zero-knowledge compiler for computation commitments.
2. For computation commitments, we build a single circuit that produces evaluations of $\widetilde{A}, \widetilde{B}, \widetilde{C}$ at $(r_x, r_y)$. This enables reusing parts of the memory checking circuit (related to the state of the memory) across evaluations.
3. In our particular context, we can set $\forall 0 \le i < n$: $\mathit{write\text{-}ts}_{row}[i] = \mathit{read\text{-}ts}_{row}[i] + 1$ and $\mathit{write\text{-}ts}_{col}[i] = \mathit{read\text{-}ts}_{read}[i] + 1$. This is because unlike the traditional setting of offline memory checking, the read timestamps are not untrusted. This avoids having to commit to $\widetilde{\mathit{write\text{-}ts}}_{row}$ and $\widetilde{\mathit{write\text{-}ts}}_{col}$.
4. During $\mathsf{PC}^{\text{SPARK}}$.Eval, at the witness layer in Hyrax, $\mathcal{V}$ needs to evaluate a number of multilinear polynomials at either $r_{row}, r_{col} \in \mathbb{F}^{\log n}$ or $r_{mem} \in \mathbb{F}^{\log m}$. We avoid having to commit to them by leveraging their succinct representations.

- $\mathcal{V}$ can compute $\widetilde{mem}_{row}(r_{row})$ and $\widetilde{mem}_{col}(r_{col})$ in $O(\log m)$ as follows:

$$\widetilde{mem}_{row}(r_{row}) \leftarrow \widetilde{\mathrm{eq}}(r_{row}, r_x)$$

$$\widetilde{mem}_{col}(r_{col}) \leftarrow \widetilde{\mathrm{eq}}(r_{col}, r_y)$$

- We leverage the following facts: (1) $\widetilde{(0, 1, \ldots, m-1)}(r_{mem}) = \sum_{i=0}^{\log m} 2^i \cdot r_{mem}[i]$; (2) $\widetilde{(0, 0, \ldots, 0)}(r_{mem}) = 0$.

5. It is possible to combine $k$ $\mu$-variate multilinear polynomials into a single multilinear polynomial over $\mu + \log k$ variables. We employ this technique to reduce the number of committed multilinear polynomials from 23 to 3.

## 8 Implementation and optimizations

We implement Spartan as a modular library in about 8,000 lines of Rust including optimizations listed throughout the paper as well as optimizations from prior work [78, 80, 82, 84, 85]. We find that the prover under SPARK outperforms the prover under SPARK-naive by $> 10\times$ for R1CS instances with $2^{20}$ constraints. We also implement SPARK with and without our optimizations. At $2^{20}$ constraints, our optimizations reduce proof lengths from 3.1 MB to 138.4 KB, a improvement of $23\times$; our optimizations also improve prover and verification times by about $10\times$.

In the next section, we present results from SPARK instantiated with Hyrax-PC [84] i.e., we evaluate a zkSNARK whose security holds under the discrete logarithm problem. For curve arithmetic, we use `curve25519-dalek` [1], which offers an efficient implementation of a prime-order Ristretto group [2, 53] called `ristretto255`. The scalar arithmetic in the library is however slow since it represents the underlying scalar elements as byte strings for fast curve arithmetic. To cope with this, we optimize the underlying scalar arithmetic by $\approx 10\times$ by adapting other code [28].

## 9 Experimental evaluation

This section experimentally evaluates our implementation of Spartan and compares it with state-of-the-art zkSNARKs and proof-succinct NIZKs.

### 9.1 Metrics, methodology, and testbed

Our principal evaluation metrics are: (1) $\mathcal{P}$'s costs to produce a proof; (2) $\mathcal{V}$'s costs to preprocess an R1CS instance; (3) $\mathcal{V}$'s costs to verify a proof; and (4) the size of a proof. We measure $\mathcal{P}$'s and $\mathcal{V}$'s costs using a real-time clock and the size of proofs in bytes by serializing proof data structures. For Spartan, we use `cargo bench` to run experiments, and for baselines, we use profilers provided with their code.

We experiment with Spartan and several baselines (listed below) using a Microsoft Surface Laptop 3 on a single CPU core of Intel Core i7-1065G7 running Ubuntu 20.04 atop Windows 10. We report results from a single-threaded configuration since not all our baselines leverage multiple cores. As with prior work [16], we vary the size of the R1CS instance by varying the number of constraints and variables $m$ and maintain the ratio $n/m$ to approximately 1. In all Spartan experiments $|io| = 10$.

**Baselines.** We compare Spartan with the following zkSNARKs and NIZKs.

1. Groth16 [51], the most efficient zkSNARK with trusted setup based on GGPR [47].
2. Ligero [3], a prior proof-succinct NIZK with a light-weight prover.
3. Hyrax [84], a prior transparent zkSNARK that achieves sub-linear verification costs for data-parallel computations.
4. Aurora [16], a prior proof-succinct NIZK.
5. Fractal [36], a recent transparent zkSNARK that instantiates computation commitments to achieve sub-linear verification costs.

We provide a comparison with additional baselines in an extended report [72].

**Methodology and parameters.** For Spartan$_{\text{DL}}$, we report results from two variants: SpartanSNARK (which incurs sub-linear verification) and SpartanNIZK (which incurs linear-time verification). This is because several baselines offer only a linear-time verifier. Also, for data-parallel workloads, the NIZK variant depicts the performance that SpartanSNARK can achieve for the prover and proof sizes since SpartanSNARK can amortize the costs of computation commitments across data-parallel units.

For Groth16, we benchmark its implementation from `libsnark` with `bn128` curve [64].

For Hyrax, we use its reference implementation with curve25519 [62]. To compare Spartan with Hyrax, we transform R1CS instances to depth-1 arithmetic circuits where the circuit evaluates constraints in the R1CS instance, and outputs a vector of zeros when all constraints are satisfied. For an arbitrary R1CS instance, this circuit has no structure, and hence Hyrax incurs linear-time verification costs.

For Ligero, Aurora, and Fractal, we use their implementations from `libiop` with a prime field of size $\approx 2^{256}$ [63]. The implementations of Aurora and Fractal support two sets of parameters: proven and non-proven (also known as heuristic). The default choice in their code is the heuristic parameters, which rely on non-standard conjectures related to Reed-Solomon codes (e.g., in the FRI protocol) for soundness [10, Appendix B]. Concretely, the heuristic parameters use $\approx$10$\times$ fewer query repetitions of FRI compared to the proven parameters. As expected, the heuristic versions achieve $\approx$10$\times$ lower verification costs and proof sizes than the corresponding provable versions. Note that very recent work makes progress toward proving some of these heuristics [11].

## 9.2 Performance results

**Prover.** Figure 6 depicts the prover's costs under Spartan and its baselines. Spartan outperforms all its baselines. When compared to the most closely related system, SpartanSNARK is 36$\times$ faster than Fractal at $2^{18}$ constraints.[4] When we compare Ligero, Aurora, and Hyrax with SpartanNIZK (since all of them are proof-succinct NIZKs and incur linear-time verification costs), SpartanNIZK is 24$\times$ faster than Ligero, 152$\times$ faster than Aurora, and 99$\times$ faster than Hyrax at $2^{20}$ instance sizes. Finally, compared to Groth16, SpartanSNARK is 2$\times$ faster and SpartanNIZK is 16$\times$ faster for $2^{20}$ constraints.

**Proof sizes.** Figure 7 depicts proof sizes under Spartan and its baselines. Although SpartanSNARK's proofs are asymptotically larger than Fractal (Figure 2), SpartanSNARK

[4]Unfortunately, we could not run Fractal at $2^{19}$ or $2^{20}$ constraints because it crashes by running out of memory.

| | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ | $2^{15}$ | $2^{16}$ | $2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Groth16 | 0.17 | 0.26 | 0.46 | 0.85 | 1.5 | 2.8 | 5.4 | 10.1 | 23.2 | 44.7 | 76.2 |
| Hyrax | 1.2 | 1.7 | 2.8 | 4.1 | 7.2 | 13.9 | 22.5 | 44.6 | 90 | 181 | 447 |
| Ligero | 0.2 | 0.3 | 0.6 | 1.2 | 2.3 | 3.4 | 6.7 | 13.7 | 27.2 | 56.3 | 112 |
| Ligero-heuristic | 0.16 | 0.3 | 0.6 | 1.3 | 2.4 | 3.5 | 6.6 | 13 | 25 | 51.3 | 101 |
| Aurora | 0.7 | 1.2 | 2.5 | 4.6 | 9 | 17.3 | 36 | 69 | 140 | 282 | 688 |
| Aurora-heuristic | 0.4 | 0.7 | 1.4 | 3.2 | 5.8 | 12.2 | 24.8 | 52.2 | 108 | 224 | 509 |
| Fractal | 1 | 1.8 | 3.6 | 6.7 | 15 | 30 | 61 | 125 | 337 | – | – |
| Fractal-heuristic | 0.8 | 1.5 | 3 | 6.5 | 14 | 29 | 60 | 125 | 342 | – | – |
| SpartanNIZK | 0.02 | 0.03 | 0.04 | 0.06 | 0.1 | 0.17 | 0.33 | 0.57 | 1.1 | 2.14 | 4.5 |
| SpartanSNARK | 0.07 | 0.13 | 0.21 | 0.39 | 0.79 | 1.3 | 2.6 | 4.9 | 9.2 | 18.5 | 36.3 |

FIGURE 6—Prover's performance (in seconds) for varying R1CS instance sizes under different schemes.

offers $\approx 23\times$ shorter proofs at $2^{18}$ constraints. When we compare the proof-succinct NIZKs, SpartanNIZK offers proofs that are 1.2–416$\times$ shorter than its baselines. All transparent zkSNARKs produce orders of magnitude longer proofs than Groth16.

| | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ | $2^{15}$ | $2^{16}$ | $2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Hyrax | 13.7 | 15.7 | 16.8 | 19.9 | 21 | 26.3 | 27.5 | 37 | 38.2 | 56.4 | 57.5 |
| Ligero | 546 | 628 | 1M | 1.2M | 2M | 3M | 5M | 5M | 10M | 10M | 20M |
| Ligero-heuristic | 559 | 620 | 1M | 1.1M | 2M | 3M | 5M | 5M | 10M | 10M | 20M |
| Aurora | 447 | 510 | 610 | 717 | 810 | 931 | 1M | 1.1M | 1.3M | 1.5M | 1.6M |
| Aurora-heuristic | 53 | 58 | 70 | 75 | 82 | 95 | 101 | 111 | 121 | 129 | 141 |
| Fractal | 1.1M | 1.2M | 1.4M | 1.5M | 1.7M | 1.8M | 2M | 2.1M | 2.3M | – | – |
| Fractal-heuristic | 125 | 136 | 148 | 163 | 177 | 189 | 206 | 219 | 234 | – | – |
| SpartanNIZK | 9.3 | 10 | 11.7 | 12.5 | 15.2 | 16 | 20.7 | 21.5 | 30.3 | 31.1 | 48 |
| SpartanSNARK | 32 | 37 | 41.7 | 48 | 54 | 63 | 71.6 | 85 | 98 | 120 | 142 |

FIGURE 7—Proof sizes in KBs for various zkSNARKs. Entries with "M" are in megabytes. The proof sizes under Groth16 [51] is 128 bytes for all instance sizes.

**Verifier.** Figure 8 depicts the verifier times under different schemes. Groth16 offers the fastest verifier, but it requires a trusted setup. Among schemes without trusted setup, Spartan offers the fastest verifier. Specifically, SpartanSNARK's verifier is $3.6\times$ faster than Fractal (at the largest instance size Fractal can run), and at $2^{20}$ constraints, it is $1326\times$ faster than Aurora, $383\times$ faster than Ligero, and $80\times$ faster than Hyrax. This type of performance is expected because Aurora, Ligero, and Hyrax incur linear costs for the verifier whereas SpartanSNARK (and Fractal) incur sub-linear verification costs due to the use of computation commitments, which requires preprocessing the non-*io* component of an R1CS instance (we quantify the costs of that process below). Among proof-succinct NIZKs, SpartanNIZK is $22\times$ faster than Hyrax, $363\times$ faster than Aurora, and $105\times$ faster than Ligero at $2^{20}$ constraints.

**Encoder.** Figure 9 depicts the cost to the verifier to preprocess an R1CS instance (without the *io* component) under SpartanSNARK, Fractal [36], and Groth16 [51]. We do not depict other baselines because they do not require any preprocessing. SpartanSNARK's encoder is up to $52\times$ faster than Fractal's encoder and about $4.7\times$ faster than the trusted setup for Groth16 at the largest instance sizes.

| | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ | $2^{15}$ | $2^{16}$ | $2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Hyrax | 206 | 231 | 253 | 257 | 331 | 473 | 594 | 926 | 1.6s | 3.1s | 8.1s |
| Ligero | 52 | 100 | 183 | 398 | 823 | 1.2s | 2.2s | 4.8s | 9.5s | 19s | 38.5s |
| Ligero-heuristic | 53 | 99 | 176 | 446 | 822 | 1.2s | 2.3s | 4.3s | 8.3s | 16.6s | 34.6s |
| Aurora | 221 | 351 | 694 | 1.1s | 2.1s | 4.1s | 8.3s | 14.7s | 30s | 56s | 133s |
| Aurora-heuristic | 16 | 25 | 47 | 86 | 166 | 359 | 597 | 1.2s | 2.4s | 5.3s | 10s |
| Fractal | 147 | 138 | 165 | 172 | 174 | 195 | 195 | 198 | 204 | – | – |
| Fractal-heuristic | 11 | 8.5 | 10 | 13 | 14 | 16 | 14 | 15 | 16 | – | – |
| SpartanNIZK | 5 | 6 | 7.4 | 9.2 | 12.4 | 17.5 | 28 | 49 | 88.4 | 188.9 | 366 |
| SpartanSNARK | 9.6 | 11.4 | 13.9 | 16.4 | 21 | 25 | 34.3 | 42 | 55.9 | 70.8 | 100.3 |

FIGURE 8—Verifier's performance (in ms) under different schemes. Entries with "s" are in seconds. The verifier under Groth16 [51] takes $\approx 2$ ms at all instance sizes.

| | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ | $2^{15}$ | $2^{16}$ | $2^{17}$ | $2^{18}$ | $2^{19}$ | $2^{20}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Groth16 | 0.13 | 0.23 | 0.4 | 0.75 | 1.5 | 2.8 | 5.3 | 10.9 | 21.4 | 48.4 | 71.9 |
| Fractal | 0.3 | 0.6 | 1.3 | 2.6 | 6 | 12.7 | 26.8 | 56 | 120 | 389 | – |
| Fractal-heuristic | 0.3 | 0.6 | 1.2 | 2.6 | 5.9 | 12.5 | 26.1 | 55 | 119 | 358 | – |
| SpartanSNARK | 0.04 | 0.06 | 0.12 | 0.19 | 0.4 | 0.7 | 1.4 | 2.2 | 4.5 | 7.4 | 15.1 |

FIGURE 9—Encoder's performance (in seconds) for varying R1CS instance sizes under different schemes.

## Acknowledgments

Comments from Sebastian Angel, Melissa Chase, Ben Fisch, Esha Ghosh, Abhiram Kothapalli, Satya Lokam, Bryan Parno, Ioanna Tzialla, Ramarathnam Venkatesan, and the CRYPTO reviewers helped improve this paper. Special thanks to Justin Thaler, Riad Wahby, and Michael Walfish for their detailed attention and thorough comments, which helped clarify several aspects of this work. We thank Jonathan Lee for insightful discussions on various topics covered in this work.

## References

[1] A pure-Rust implementation of group operations on Ristretto and Curve25519. `https://github.com/dalek-cryptography/curve25519-dalek`.

[2] The Ristretto group. `https://ristretto.group/`.

[3] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *CCS*, 2017.

[4] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *SIGMOD*, 2017.

[5] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3), May 1998.

[6] S. Arora and S. Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, Jan. 1998.

[7] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, 1991.

[8] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 2(4), Dec. 1992.

[9] M. Ben-Or, O. Goldreich, S. Goldwasser, J. Håstad, J. Kilian, S. Micali, and P. Rogaway. Everything provable is provable in zero-knowledge. In *CRYPTO*, pages 37–56, 1988.

[10] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. Scalable, transparent, and post-quantum secure computational integrity. ePrint Report 2018/046, 2018.

[11] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf. Proximity gaps for reed-solomon codes. Cryptology ePrint Archive, Report 2020/654, 2020.

[12] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *S&P*, 2014.

[13] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems: Extended abstract. In *ITCS*, 2013.

[14] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete efficiency of probabilistically-checkable proofs. In *STOC*, pages 585–594, 2013.

[15] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, Aug. 2013.

[16] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, 2019.

[17] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive Oracle Proofs. In *TCC*, 2016.

[18] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, 2014.

[19] E. Ben-Sasson, O. Goldreich, P. Harsha, M. Sudan, and S. Vadhan. Short PCPs verifiable in polylogarithmic time. In *Computational Complexity*, 2005.

[20] E. Ben-Sasson and M. Sudan. Simple PCPs with poly-log rate and query complexity. In *STOC*, pages 266–275, 2005.

[21] E. Ben-Sasson and M. Sudan. Short PCPs with polylog query complexity. *SIAM J. Comput.*, 38(2):551–607, May 2008.

[22] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, 2012.

[23] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, 2013.

[24] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *FOCS*, 1991.

[25] A. J. Blumberg, J. Thaler, V. Vu, and M. Walfish. Verifiable computation using multiple provers. ePrint Report 2014/846, 2014.

[26] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. ePrint Report 2019/188, 2019.

[27] J. Bootle, A. Cerulli, P. Chaidos, J. Groth, and C. Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *EUROCRYPT*, 2016.

[28] S. Bowe. A BLS12-381 implementation. `https://github.com/zkcrypto/bls12_381`.

[29] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu. Zexe: Enabling decentralized private computation. ePrint Report 2018/962, 2018.

[30] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, Oct. 1988.

[31] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, 2013.

[32] B. Bunz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. ePrint Report 2019/1229, 2019.

[33] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *S&P*, 2018.

[34] M. Campanelli, D. Fiore, and A. Querol. LegoSNARK: modular design and composition of succinct zero-knowledge proofs. ePrint Report 2019/142, 2019.

[35] A. Chiesa, M. A. Forbes, and N. Spooner. A zero knowledge sumcheck and its applications. *CoRR*, abs/1704.02086, 2017.

[36] A. Chiesa, D. Ojha, and N. Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. ePrint Report 2019/1076, 2019.

[37] D. Clarke, S. Devadas, M. V. Dijk, B. Gassend, G. Edward, and S. Mit. Incremental multiset hash functions and their application to memory integrity checking. In *ASIACRYPT*, 2003.

[38] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, 2012.

[39] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *S&P*, May 2015.

[40] R. Cramer and I. Damgård. Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free? In *CRYPTO*, pages 424–441, 1998.

[41] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, and B. Parno. Cinderella: Turning shabby X.509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *S&P*, 2016.

[42] I. Dinur. The PCP theorem by gap amplification. *J. ACM*, 54(3), June 2007.

[43] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *TCC*, 2009.

[44] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Interactive proofs and the hardness of approximating cliques. *J. ACM*, 43(2):268–292, Mar. 1996.

[45] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[46] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *CCS*, 2016.

[47] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[48] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108, 2011.

[49] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, 2008.

[50] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, 2010.

[51] J. Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.

[52] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In *CRYPTO*, 2018.

[53] M. Hamburg. Decaf: Eliminating cofactors through point compression. In *CRYPTO*, 2015.

[54] J. Håstad. Some optimal inapproximability results. In *STOC*, pages 1–10, 1997.

[55] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Computational Complexity*, 2007.

[56] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In *STOC*, pages 21–30, 2007.

[57] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, pages 177–194, 2010.

[58] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, 1992.

[59] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *S&P*, 2016.

[60] A. Kosba, C. Papamanthou, and E. Shi. xJsnark: A framework for efficient verifiable computation. In *S&P*, 2018.

[61] J. Lee, K. Nikitin, and S. Setty. Replicated state machines without replicated execution. In *S&P*, 2020.

[62] libfennel. Hyrax reference implementation. `https://github.com/hyraxZK/fennel`.

[63] libiop. A C++ library for IOP-based zkSNARK. `https://github.com/scipr-lab/libiop`.

[64] libsnark. A C++ library for zkSNARK proofs.
`https://github.com/scipr-lab/libsnark`.

[65] H. Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In *TCC*, 2012.

[66] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *FOCS*, Oct. 1990.

[67] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1988.

[68] S. Micali. CS proofs. In *FOCS*, 1994.

[69] C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *TCC*, 2013.

[70] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *S&P*, May 2013.

[71] O. Reingold, G. N. Rothblum, and R. D. Rothblum. Constant-round interactive proofs for delegating computation. In *STOC*, pages 49–62, 2016.

[72] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. ePrint Report 2019/550, 2019.

[73] S. Setty, S. Angel, T. Gupta, and J. Lee. Proving the correct execution of concurrent services in zero-knowledge. In *OSDI*, Oct. 2018.

[74] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, May 2011.

[75] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, Apr. 2013.

[76] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, Feb. 2012.

[77] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, Aug. 2012.

[78] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, 2013.

[79] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *HotCloud*, 2012.

[80] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for verifiable computation. In *S&P*, 2013.

[81] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish. Verifiable ASICs. In *S&P*, 2016.

[82] R. S. Wahby, Y. Ji, A. J. Blumberg, A. Shelat, J. Thaler, M. Walfish, and T. Wies. Full accounting for verifiable outsourcing. In *CCS*, 2017.

[83] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[84] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *S&P*, 2018.

[85] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. ePrint Report 2019/317, 2019.

[86] J. Zhang, T. Xie, Y. Zhang, and D. Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *S&P*, 2020.

[87] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *S&P*, 2017.

[88] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. A zero-knowledge version of vSQL. ePrint Report 2017/1146, 2017.

[89] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *S&P*, 2018.